USENIX Association

Proceedings of the 17th USENIX Conference on File and Storage Technologies

February 25–28, 2019
Boston, MA, USA

Conference Organizers

Program Co-Chairs
Arif Merchant, *Google*
Hakim Weatherspoon, *Cornell University*

Program Committee
Nitin Agrawal, *Samsung Research*
Mahesh Balakrishnan, *Yale University and Facebook*
André Brinkmann, *Johannes Gutenberg-University Mainz*
Vijay Chidambaram, *The University of Texas at Austin*
Angela Demke Brown, *University of Toronto*
Peter Desnoyers, *Northeastern University*
Ashvin Goel, *University of Toronto*
Ajay Gulati, *ZeroStack*
Haryadi Gunawi, *University of Chicago*
Tim Harris, *Amazon*
Cheng Huang, *Microsoft Research and Azure*
Bill Jannen, *Williams College*
Rob Johnson, *VMware Research Group*
Kim Keeton, *Hewlett Packard Enterprise*
Geoff Kuenning, *Harvey Mudd College*
Sungjin Lee, *DGIST (Daegu Gyeongbuk Institute of Science and Technology)*
Peter Macko, *NetApp*
Umesh Maheshwari, *Nimble Storage (an HPE company)*
Arif Merchant, *Google*
Onur Mutlu, *ETH Zurich and Carnegie Mellon University*
Sam H. Noh, *UNIST (Ulsan National Institute of Science and Technology)*
Florentina Popovici, *Google*
Raju Rangaswami, *Florida International University*
Ken Salem, *University of Waterloo*
Jiri Schindler, *Tranquil Data*
Bianca Schroeder, *University of Toronto*
Keith A Smith, *NetApp*
Ioan Stefanovici, *Microsoft Research*
Swaminathan Sundararaman, *ParallelM*
Nisha Talagala, *ParallelM*
Vasily Tarasov, *IBM Research*
Joseph Tucek, *Amazon*
Carl Waldspurger, *Carl Waldspurger Consulting*
Andrew Warfield, *Amazon*

Work-in-Progress/Posters Co-Chairs
Bill Jannen, *Williams College*
Vasily Tarasov, *IBM Research*

Test of Time Awards Committee
Jiri Schindler, *Tranquil Data*
Eno Thereska, *Amazon*
Erez Zadok, *Stony Brook University*

Tutorial Coordinators
John Strunk, *Red Hat*
Eno Thereska, *Amazon*

Steering Committee
Nitin Agrawal, *Samsung Research*
Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*
Angela Demke Brown, *University of Toronto*
Greg Ganger, *Carnegie Mellon University*
Casey Henderson, *USENIX Association*
Kimberly Keeton, *HP Labs*
Geoff Kuenning, *Harvey Mudd College*
Florentina Popovici, *Google*
Raju Rangaswami, *Florida International University*
Erik Riedel
Jiri Schindler, *Tranquil Data*
Bianca Schroeder, *University of Toronto*
Keith A. Smith, *NetApp*
Eno Thereska, *Amazon*
Carl Waldspurger, *Carl Waldspurger Consulting*
Ric Wheeler, *Facebook*
Erez Zadok, *Stony Brook University*

External Reviewers

Mingzhe Hao
Huaicheng Li
Jake Wires
Ali Razeen
Mihir Nanavati
James Kelley
Kishore Kasi Udayashankar
Jasmina Malicevic
Peter Corbett

Message from the FAST '19 Program Co-Chairs

Welcome to the 17th USENIX Conference on File and Storage Technologies, FAST '19. This year's conference continues the tradition of bringing together researchers and practitioners from both industry and academia for a program of innovative and rigorous storage-related research. We are pleased to present a diverse set of papers on topics such as persistent memory systems, deduplication, erasure coding and reliability, and traditional file systems. Submissions to the conference came from 20 countries on 4 continents, from authors representing academia, industry, and the open source community.

FAST '19 received 145 submissions—a record high number. Of these, we accepted 26 papers, for an acceptance rate of 18%. The Program Committee used a two-round online review process and then met in person to select the final program. In the first round, each paper received at least three reviews. For the second round, 96 papers received at least two more reviews. The Program Committee discussed 58 papers in an all-day meeting on December 3, 2018, at Google in Sunnyvale, CA. We used Eddie Kohler's excellent HotCRP software to manage all stages of the review process, from submission to author notification.

As in the previous years, we included a category of short papers. Short papers provide a vehicle for presenting completed research that does not require a full-length paper to describe and evaluate. We received 27 short paper submissions, of which 2 were accepted. Also in line with previous years, we included a category of deployed-systems papers, which address experience with the practical design, implementation, analysis or deployment of large-scale, operational systems. We received 12 deployed-systems submissions, of which we accepted 3.

We wish to thank the many people who contributed to this conference. First and foremost, we are grateful to all the authors who submitted their work to FAST '19. We would also like to thank the attendees of FAST '19 and the future readers of these papers. Together with the authors, you form the FAST community and make storage research vibrant and exciting. We extend our thanks to the entire USENIX staff, especially Casey Henderson, Jasmine Murcia, Jessica Kim, Michele Nelson, and Arnold Gatilao, who have provided outstanding support throughout the planning and organizing of this conference with the highest degree of professionalism and friendliness. Most importantly, their behind-the-scenes work makes this conference actually happen. We would like to thank the Poster and Work-in-Progress session Chairs, Bill Jannen and Vasily Tarasov. Our thanks go also to the members of the FAST Steering Committee who provided invaluable advice and feedback, and to our Steering Committee Liaison, Keith Smith, for his guidance and encouragement on many issues, large and small, over the past year.

Finally, we wish to thank our Program Committee for their many hours of hard work reviewing and discussing the submissions, some of whom traveled halfway across the world for the one-day in-person PC meeting. In total, they wrote 637 thoughtful and meticulous reviews. HotCRP recorded over 447,194 words in reviews and comments (excluding HotCRP boilerplate; 496,030 when included). The reviewers' evaluations, and their thorough and conscientious deliberations at the PC meeting, contributed significantly to the quality of our decisions.

We look forward to an interesting and enjoyable conference!

Arif Merchant, *Google*
Hakim Weatherspoon, *Cornell University*
FAST '19 Program Co-Chairs

FAST ’19: 17th USENIX Conference on File and Storage Technologies
February 25–28, 2019
Boston, MA, USA

Persistent Memory Systems

Reaping the performance of fast NVM storage with uDepot 1
Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas, *IBM Research*

Optimizing Systems for Byte-Addressable NVM by Reducing Bit Flipping 17
Daniel Bittman, Darrell D. E. Long, Peter Alvaro, and Ethan L. Miller, *UC Santa Cruz*

Write-Optimized Dynamic Hashing for Persistent Memory 31
Moohyeon Nam, *UNIST* (*Ulsan National Institute of Science and Technology*)*;* Hokeun Cha, *Sungkyunkwan University;* Young-ri Choi and Sam H. Noh, *UNIST* (*Ulsan National Institute of Science and Technology*)*;* Beomseok Nam, *Sungkyunkwan University*

Software Wear Management for Persistent Memories 45
Vaibhav Gogte, *University of Michigan;* William Wang and Stephan Diestelhorst, *ARM;* Aasheesh Kolli, *Pennsylvania State University and VMware Research;* Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch, *University of Michigan*

File Systems

Storage Gardening: Using a Virtualization Layer for Efficient Defragmentation in the WAFL File System 65
Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra, *NetApp*

Pay Migration Tax to Homeland: Anchor-based Scalable Reference Counting for Multicores 79
Seokyong Jung, Jongbin Kim, Minsoo Ryu, Sooyong Kang, and Hyungsoo Jung, *Hanyang University*

Speculative Encryption on GPU Applied to Cryptographic File Systems 93
Vandeir Eduardo, *Federal University of Paraná and University of Blumenau*; Luis C. Erpen de Bona and Wagner M. Nunan Zola, *Federal University of Paraná*

Deduplication

Sketching Volume Capacities in Deduplicated Storage 107
Danny Harnik and Moshik Hershcovitch, *IBM Research;* Yosef Shatsky, *IBM Systems;* Amir Epstein, *Citi Innovation Lab TLV;* Ronen Kat, *IBM Research*

Finesse: Fine-Grained Feature Locality based Fast Resemblance Detection for Post-Deduplication Delta Compression 121
Yucheng Zhang, *Hubei University of Technology;* Wen Xia, *Harbin Institute of Technology, Shenzhen & Peng Cheng Laboratory;* Dan Feng, *WNLO, School of Computer, Huazhong University of Science and Technology;* Hong Jiang, *University of Texas at Arlington;* Yu Hua and Qiang Wang, *WNLO, School of Computer, Huazhong University of Science and Technology*

Sliding Look-Back Window Assisted Data Chunk Rewriting for Improving Deduplication Restore Performance . . 129
Zhichao Cao, *University of Minnesota;* Shiyong Liu, *Ocean University of China;* Fenggang Wu, *University of Minnesota;* Guohua Wang, *South China University of Technology;* Bingzhe Li and David H.C. Du, *University of Minnesota*

Storage Potpourri

DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching 143
Zaoxing Liu and Zhihao Bai, *Johns Hopkins University;* Zhenming Liu, *College of William and Mary;* Xiaozhou Li, *Celer Network;* Changhoon Kim, *Barefoot Networks;* Vladimir Braverman and Xin Jin, *Johns Hopkins University;* Ion Stoica, *UC Berkeley*

GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction 159
Ting Yao, *Huazhong University of Science and Technology and Temple University;* Jiguang Wan, *Huazhong University of Science and Technology;* Ping Huang, *Temple University;* Yiwen Zhang, Zhiwen Liu, and Changsheng Xie, *Huazhong University of Science and Technology;* Xubin He, *Temple University*

Speicher: Securing LSM-based Key-Value Stores using Shielded Execution 173
Maurice Bailleu, Jörg Thalheim, and Pramod Bhatotia, *The University of Edinburgh;* Christof Fetzer, *TU Dresden;* Michio Honda, *NEC Labs;* Kapil Vaswani, *Microsoft Research*

NVM File and Storage Systems

SLM-DB: Single-Level Key-Value Store with Persistent Memory 191
Olzhas Kaiyrakhmet and Songyi Lee, *UNIST;* Beomseok Nam, *Sungkyunkwan University;* Sam H. Noh and Young-ri Choi, *UNIST*

Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks 207
Shengan Zheng, *Shanghai Jiao Tong University;* Morteza Hoseinzadeh and Steven Swanson, *University of California, San Diego*

Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks 221
Jian Yang, Joseph Izraelevitz, and Steven Swanson, *UC San Diego*

Big Systems

INSTalytics: Cluster Filesystem Co-design for Big-data Analytics 235
Muthian Sivathanu, Midhul Vuppalapati, Bhargav Gulavani, Kaushik Rajan, and Jyoti Leeka, *Microsoft Research India;* Jayashree Mohan, *Univ. of Texas Austin;* Piyus Kedia, *IIIT Delhi*

GraphOne: A Data Store for Real-time Analytics on Evolving Graphs 249
Pradeep Kumar and H. Howie Huang, *George Washington University*

Automatic, Application-Aware I/O Forwarding Resource Allocation 265
Xu Ji, *Tsinghua University; National Supercomputing Center in Wuxi;* Bin Yang and Tianyu Zhang, *National Supercomputing Center in Wuxi; Shandong University;* Xiaosong Ma, *Qatar Computing Research Institute, HBKU;* Xiupeng Zhu, *National Supercomputing Center in Wuxi; Shandong University;* Xiyang Wang, *National Supercomputing Center in Wuxi;* Nosayba El-Sayed, *Emory University;* Jidong Zhai, *Tsinghua University;* Weiguo Liu, *National Supercomputing Center in Wuxi; Shandong University;* Wei Xue, *Tsinghua University; National Supercomputing Center in Wuxi*

Flash and Emerging Storage Systems

Design Tradeoffs for SSD Reliability 281
Bryan S. Kim, *Seoul National University;* Jongmoo Choi, *Dankook University;* Sang Lyul Min, *Seoul National University*

Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts 295
Taejin Kim and Duwon Hong, *Seoul National University;* Sangwook Shane Hahn, *Western Digital;* Myoungjun Chun, *Seoul National University;* Sungjin Lee, *DGIST;* Jooyoung Hwang and Jongyoul Lee, *Samsung Electronics;* Jihong Kim, *Seoul National University*

Large-Scale Graph Processing on Emerging Storage Devices 309
Nima Elyasi, *The Pennsylvania State University;* Changho Choi, *Samsung Semiconductor Inc.;* Anand Sivasubramaniam, *The Pennsylvania State University*

(continued on next page)

Erasure Coding and Reliability

Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques. 317
Tianli Zhou and Chao Tian, *Texas A&M University*

OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems 331
Xiaolu Li, Runhui Li, and Patrick P. C. Lee, *The Chinese University of Hong Kong;* Yuchong Hu, *Huazhong University of Science and Technology*

Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity 345
Saurabh Kadekodi, K. V. Rashmi, and Gregory R. Ganger, *Carnegie Mellon University*

ScaleCheck: A Single-Machine Approach for Discovering Scalability Bugs in Large Distributed Systems 359
Cesar A. Stuardo, *University of Chicago;* Tanakorn Leesatapornwongsa, *Samsung Research America;* Riza O. Suminto, Huan Ke, and Jeffrey F. Lukman, *University of Chicago;* Wei-Chiu Chuang, *Cloudera;* Shan Lu and Haryadi S. Gunawi, *University of Chicago*

Reaping the performance of fast NVM storage with uDepot

*Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas**
IBM Research, Zurich
{kou, nio, iko}@zurich.ibm.com

Abstract

Many applications require low-latency key-value storage, a requirement that is typically satisfied using key-value stores backed by DRAM. Recently, however, storage devices built on novel NVM technologies offer unprecedented performance compared to conventional SSDs. A key-value store that could deliver the performance of these devices would offer many opportunities to accelerate applications and reduce costs. Nevertheless, existing key-value stores, built for slower SSDs or HDDs, cannot fully exploit such devices.

In this paper, we present uDepot, a key-value store built bottom-up to deliver the performance of fast NVM block-based devices. uDepot is carefully crafted to avoid inefficiencies, uses a two-level indexing structure that dynamically adjusts its DRAM footprint to match the inserted items, and employs a novel task-based IO run-time system to maximize performance, enabling applications to use fast NVM devices at their full potential. As an embedded store, uDepot's performance nearly matches the raw performance of fast NVM devices both in terms of throughput and latency, while being scalable across multiple devices and cores. As a server, uDepot significantly outperforms state-of-the-art stores that target SSDs under the YCSB benchmark. Finally, using a Memcache service on top of uDepot we demonstrate that data services built on NVM storage devices can offer equivalent performance to their DRAM-based counterparts at a much lower cost. Indeed, using uDepot we have built a cloud Memcache service that is currently available as an experimental offering in the public cloud.

1 Introduction

Advancements in non-volatile memory (NVM) technologies enable a new class of block-based storage devices with unprecedented performance. These devices, which we refer to as Fast NVMe Devices (FNDs), achieve hundreds of thousands of IO operations per second (IOPS) as well as low latency, and constitute a discrete point in the performance/cost tradeoff spectrum between DRAM and conventional SSDs. To illustrate the difference, the latency of fetching a 4 KiB block in conventional NVMe Flash SSD is 80 µs, while in FNDs the same operation takes 7 µs (Optane drive [88]) or 12 µs (Z-SSD [48, 74]). To put this in perspective, a common round-trip latency of a TCP packet over 10 Gigabit Ethernet is 25 µs-50 µs, which means that using FNDs in commodity datacenters results in storage no longer being the bottleneck.

Hence, FNDs act as a counterweight to the prevalent architectural trend of data stores placing all data in main memory [26, 35, 72, 73, 78]. Specifically, many key-value (KV) stores place all their data in DRAM [21, 25, 44, 52, 57, 59, 68, 73] to meet application performance requirements. An FND-based KV store offers an attractive alternative to DRAM-based systems in terms of cost and capacity scalability.[1] We expect that many applications, for which conventional SSDs are not performant enough, can now satisfy their performance requirements using KV stores built on FNDs. In fact, since for many common setups FNDs shift the bottleneck from storage to the network, it is possible for FND-based KV stores to provide equivalent performance to that of their DRAM-based counterparts.

Existing KV stores, however, cannot use FNDs to their full potential. First, KV stores that place all their data in DRAM require OS paging to transparently use FNDs, which results in poor performance [33]. Second, KV stores that place their data in storage devices [8, 24, 31, 50], even those that specifically target conventional SSDs [3, 19, 20, 58, 60, 84, 87, 91], are designed with different requirements in mind: slower devices, smaller capacity, and/or no need to scale over multiple devices and cores. As Barroso et al. [7] point out, most existing systems under-perform in the face of IO operations that take a few microseconds.

Motivated by the above, we present uDepot, a KV store designed from the ground up to deliver the performance of FNDs. The core of uDepot is an embedded store that can

*Now at Google.

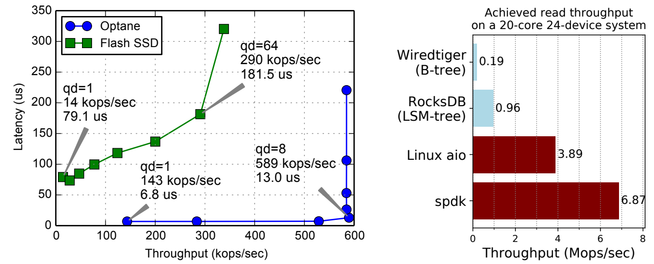
[1] At the time of writing: DRAM costs about $10/GiB, an Optane NVMe drive $1.25/GiB, and a commodity Flash NVMe drive $0.4/GiB.

be used by applications as a library. Using this embedded store we build two network services: a distributed KV store using a custom network protocol, and a distributed cache that implements the Memcache [64] protocol, which can be used as a drop-in replacement for memcached [65], a widely used [5, 70] DRAM-based cache.

By design, uDepot is *lean*: it provides streamlined functions for efficient data access, optimizing for performance instead of richer functionality (e.g., range queries). uDepot is *efficient* in that it: i) achieves low latency, ii) provides high throughput per core, iii) scales its performance with the number of drives and cores, iv) enforces low bounds to end–to-end IO amplification, in terms of bytes and number of operations, and, finally, v) achieves a high utilization of storage capacity. This requires multiple optimizations throughout the system, but two aspects are especially important. First, efficiently accessing FNDs. Most existing KV stores use synchronous IO that severely degrades performance because it relies on kernel-scheduled threads to handle concurrency. Instead, uDepot employs asynchronous IO and, if possible, directly accesses the storage device from user-space. To this end, uDepot is built on TRT, a task runtime for IO at the microsecond scale that uses user-space collaborative scheduling. Second, uDepot uses a high-performance DRAM index structure that is able to match the performance of FNDs while keeping its memory footprint small. (A small memory footprint leads to efficient capacity utilization because less DRAM is needed to index the same storage capacity.) uDepot's index structure is resizable, adapting its memory consumption to the number of items stored. Resizing does not require any IO operations, and is performed incrementally so that it causes minimal disruption.

In summary, our contributions are: **1)** uDepot, a KV store that delivers the performance of FNDs, offering low latency, high throughput, scalability, and efficient use of CPUs, memory, and storage. **2)** TRT, a task run-time system suitable for IO at the microsecond scale, which acts as a substrate for uDepot. TRT provides a programmer-friendly framework for writing applications that fully exploit fast storage. **3)** uDepot's index data structure that enables it to meet its performance goals while being space efficient, dynamically resizing to match the number of KV pairs stored. **4)** An experimental evaluation demonstrating that uDepot matches the performance of FNDs, which, to our knowledge, no existing system can. Indeed, uDepot vastly outperforms SSD-optimized stores by up to $\times 14.7$ but also matches the performance of a DRAM-backed memcached server allowing it to be used as a Memcache replacement to dramatically reduce cost. A cloud Memcache service built using uDepot is available as an experimental offering in the public cloud [39].

The rest of the paper is organized as follows. We motivate our work in §2, discuss TRT in §3, and present and evaluate uDepot in §4 and §5, respectively. In §6 we discuss related work and conclude in §7.



(a) Latency and throughput of 4 KiB random reads on two NVMe devices: a NVMe Flash SSD, and an Optane, as we vary the queue depth (operations in flight) in powers of two using SPDK's perf benchmark [86].

(b) Aggregate throughput of random 4 KiB reads using different IO facilities (aio, spdk) and storage engines (WiredTiger, RocksDB).

Figure 1

2 Background and Motivation

In 2010, arguing for an in-memory KV store, Ousterhout et al. predicted that "Within 5–10 years, assuming continued improvements in DRAM technology, it will be possible to build RAM-Clouds with capacities of 1–10 Petabytes at a cost less than $5/GB" [72]. Since then, researchers have been conducting an "arms race" to maximize performance for in-memory KV stores [10, 21, 44, 52, 57, 68, 73]. In contrast to the above prediction, however, DRAM scaling is approaching physical limits [69] and DRAM is becoming more expensive [22, 40]. Hence, as capacity demands increase, memory KV stores rely on scaling out to achieve the required storage capacity by adding more servers. Naturally, this is inefficient and comes at a high cost as the rest of the node (CPUs, storage) remains underutilized and resources required to support the additional nodes need to increase proportionally as well (space, power supplies, cooling). In addition, while the performance of memory KV stores is impressive, many depend on high-performance or specialized networking (e.g., RDMA, FPGAs) and cannot be deployed in commodity datacenter infrastructures such as the ones offered by many public cloud providers.

Fast NVMe devices (FNDs) that were released recently offer a cost-effective alternative to DRAM, with significantly better performance than conventional SSDs (Fig. 1a). Specifically, the Optane drive, based on 3D XPoint (3DXP),[2] delivers a throughput close to 0.6 Mops/s, and achieves read access latencies of 7 µs, an order of magnitude lower than conventional SSDs, which have latencies of 80 µs or higher [34]. Furthermore, Samsung announced availability of Z-SSD, a new device [89] that utilizes Z-NAND [74] and has similar performance characteristics to Optane, achieving read access latencies of 12 µs. Hence, a KV store effectively using FNDs offers an attractive alternative to its

[2] 3DXP is also used to build devices accessible as memory that offer even lower latencies. Our work focuses on IO devices because they are widely available and the most cost effective option.

DRAM counterparts. This is especially true in environments with commodity networking (e.g., 10 Gbit/s Ethernet) where FNDs shift the bottleneck from the storage to the network, and the full performance of DRAM KV stores cannot be obtained over the network.

Existing KV stores are built with slower devices in mind and fail to deliver the performance of FNDs. As a motivating example, we consider a multi-core and multi-device system aimed at minimizing cost with 20 cores and 24 NVMe drives, and compare the performance of the devices against the performance of two ubiquitous storage engines: RocksDB and WiredTiger. These engines epitomize modern KV store designs, using LSM- and B-trees. We measure device performance with microbenchmarks using the Linux asynchronous IO facility (aio) and SPDK, a library for directly accessing devices from user-space. For the two KV stores, we load 50M items of 4 KiB and measure the throughput of random `GET` operations using their accompanying microbenchmarks while setting appropriate cache sizes so that requests are directed to the devices. Even after tuning RocksDB and WiredTiger to the best of our ability, we were not able to exceed 1 Mops/s and 120 Kops/s, respectively. On the other hand, the storage devices themselves can provide 3.89 Mops/s using asynchronous IO and 6.87 Mops/s using user-space IO (SPDK). (More details about this experiment and how uDepot performs in the same setup can be found in §5.)

Overall, these stores underutilize the devices and even though experts can probably tune them to improve their performance, there are fundamental issues with their design. First, these systems, built for slower devices, use synchronous IO which is highly problematic for IO at the microsecond scale [7]. Second, they use LSM- or B-trees which are known to cause significant IO amplification. In the previous experiment, for example, RocksDB IO amplification was $\times 3$ and WiredTiger's $\times 3.5$. Third, they cache data in DRAM which requires additional synchronization but also limits scalability due to memory requirements, and finally they offer many additional features (e.g., transactions) which may have a toll on performance.

uDepot follows a different path: it is built bottom-up to deliver the performance of FNDs (e.g., by eliminating IO amplification), offers only the basic operations of a KV store, does not cache data, and uses asynchronous IO via TRT, which we describe next.

3 TRT: a task run-time system for fast IO

Broadly speaking, there are three ways to access storage: synchronous IO, asynchronous IO, and user-space IO. The majority of existing applications access storage via synchronous systems calls (e.g., `pread`, `pwrite`). As it is already well established for networking [45], synchronous IO does not scale because handling concurrent requests requires one thread for each, leading to context switches that degrade performance when the number of in-flight requests is higher than the number of cores. Hence, as with network programming, utilizing the performance of fast IO devices requires utilizing asynchronous IO [7]. For example, Linux AIO [43], allows multiple IO requests (and their completions) to be issued (and received) in batches from a single thread. Performing asynchronous IO in itself, however, is not enough to fully reap the performance of FNDs. A set of new principles have emerged for building applications that efficiently access fast IO devices. These principles include removing the kernel from the datapath, favouring polling over interrupts, and minimizing, if not precluding, cross-core communication [9, 75]. While the above techniques initially targeted mostly fast networks, they also apply to storage [47, 94]. In contrast to Linux AIO that is a kernel facility, user-space IO frameworks such as SPDK [85], allow maximizing performance by avoiding context switches, data copying, and scheduling overheads. On the other hand, it is not always possible to use them because they require direct (and in many cases unsafe) access to the device and many environments (e.g., cloud VMs) do not (yet) support them.

Hence, an efficient KV store (or a similar application) needs to access both the network and the storage asynchronously, potentially using user-space IO if available to maximize performance. Existing frameworks, such as `libevent` [55], are ill-suited for this use-case because they assume a single endpoint for the application to check for events (e.g., the `epoll_wait` [46] system call). When combining both access to the storage and network, multiple event (and event completion) endpoints that need to be checked might exist. For example, it might be that `epoll_wait` is used for network sockets, and `io_getevents` [42] or SPDK's completion processing call is used for storage. Furthermore, many of these frameworks are based on callbacks which can be troublesome to use due to the so-called "stack ripping" problem [1, 49].

To enable efficient, yet programmer-friendly, access to FNDs, we developed TRT, a Task-based Run-Time system, where tasks are collaboratively scheduled (i.e., no preemption) and each has its own stack. TRT spawns a number of threads (typically one per core) and executes a user-space scheduler on each. The scheduler executes in its own stack. Switching between the scheduler and tasks is lightweight, consisting of saving and restoring a number of registers without involving the kernel. In collaborative scheduling, tasks voluntarily switch to the scheduler via executing proper calls. An example of such a call to the scheduler is `yield` that defers execution to the next task. There are also calls to spawn tasks, and synchronization calls: waiting and notifying. The synchronization interface is based on `Futures` [29, 30]. Because TRT tries to avoid cross-core communication as much as possible, it provides two variants for the synchronization primitives: intra- and inter-core. Intra-core primitives are more efficient because they do not require syn-

chronization to protect against concurrent access as long as critical sections do not include commands that switch to the scheduler.

Based on the above primitives, TRT provides an infrastructure for asynchronous IO. In a typical scenario, each network connection would be served by a different TRT task. To enable different IO backends and facilities, each IO backend implements a poller task that is responsible for polling for events and notifying tasks to handle these events. To avoid cross-core communication, each core runs its own poller instance. As a result, tasks cannot move across core when they have pending IO operations. Poller tasks are scheduled by the scheduler as any other task.

TRT currently supports four backends: Linux AIO, SPDK (single device and RAID-0 multi-device configurations), and Epoll, with backends for RDMA and DPDK in development. Each backend provides a low-level interface that allows tasks to issue requests and wait for results, and, built on top of that, a high-level interface for writing code resembling its synchronous counterpart. For example, a `trt::spdk::read()` call will issue a read command to SPDK device queues, and call the TRT scheduler to suspend task execution until notified by the poller that processes SPDK completions.

To avoid synchronization, pollers of all backends running on different cores use separate endpoints: Linux AIO pollers use different IO contexts, SPDK pollers use different device queues, and Epoll pollers use a different control file-descriptor.

4 uDepot

uDepot supports `GET`, `PUT`, and `DELETE` operations (§4.5) on variable-sized keys and values. The maximum key and value sizes are 64 KiB and 4 GiB, respectively, with no minimum size for either. uDepot directly operates on the device and does its own (log-structured) space management (§4.1), instead of depending on a filesystem. To minimize IO amplification, uDepot uses a two-level hash table in DRAM as an index structure (§4.2) which allows implementing KV operations with a single IO operation (if no hash collision exists), but lacks support for efficient range queries. The index structure can utilize PBs of storage while still remaining memory efficient by adapting its size to the number of KV entries stored at run-time (resizing). Resizing (§4.3) causes minimal disruption because it is incremental and does not incur IO. uDepot does not cache data and is persistent (§4.4): when a `PUT` (or `DELETE`) operation returns, the data are stored in the device (not in OS cache) and will be recovered in case of a crash. uDepot supports multiple IO backends (§4.6), allowing users to maximize performance depending on their setup. uDepot can currently be used in three ways: as an embedded store linked to the application, as a distributed store over the network (§4.7), or as a cache that implements the Memcache protocol [64] (§4.8).

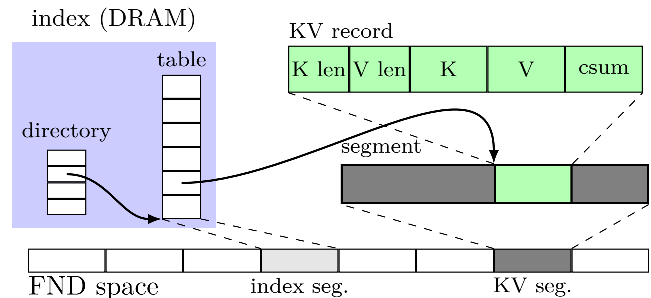


Figure 2: uDepot maintains its index structure (directory and tables) in DRAM. The FND space is split into segments of two types: index segments for flushing index tables, and KV segments for storing KV records.

4.1 Storage device space management

uDepot manages device space using a log-structured approach [67, 79], i.e., space is allocated sequentially and garbage collection (GC) deals with fragmentation. We use this approach for three reasons. First, it achieves good performance on idiosyncratic storage like NAND Flash. Second, it is more efficient than traditional allocation methods even for non-idiosyncratic storage like DRAM [80]. Third, an important use case for uDepot is caching, and there are a number of optimization opportunities when co-designing GC and caches [81, 84]. Allocation is implemented via a user-space port of the log-structured allocator of SALSA [41]. Device space is split into segments (default size: 1 GiB), which are in turn split into grains (typically sized equal to the blocks of the IO device). There are two types of segments: KV segments for storing KV records, and index segments for flushing the index structure to speed up startup (§4.4). uDepot calls SALSA to (sequentially) allocate and release grains. SALSA performs GC and upcalls uDepot to relocate specific grains to free segments [41]. SALSA's GC [76] is a generalized variant of the greedy [12] and circular buffer (CB) [79] algorithms, which augments a greedy policy with the aging factor of the CB.

4.2 Index data structure

uDepot's index is an in-memory two-level mapping directory for mapping keys to record locations in storage (Fig. 2). The directory is implemented as an atomic pointer to a read-only array of pointers to hash tables.

Hash table Each hash table implements a modified hopscotch [37] algorithm, where an entry is stored within a range of consecutive locations, which we call *neighborhood*.[3] Effectively, hopscotch works similarly to linear probe, but bounds probe distance within the neighborhood. If an entry hashes to index `i` in the hash table array, and H is the

[3]The original paper [37] also uses the term "virtual" bucket.

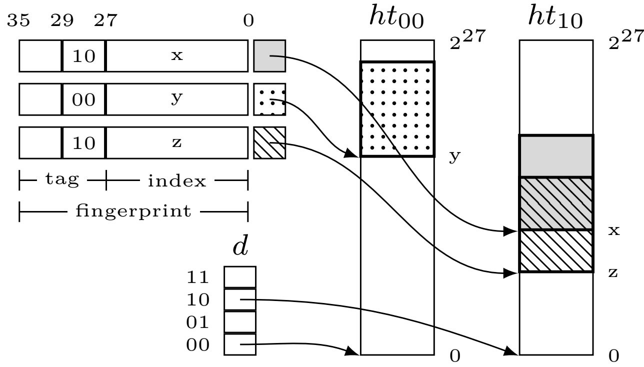


Figure 3: How key fingerprints are used to determine the neighborhood for a key. d is a directory with 4 tables, where only two are shown (ht_{00} and ht_{10}).

neighborhood size (default: 32), then the entry can be stored in any of the H valid entries starting from `i`. In the subsequent paragraphs, we refer to `i` as *neighborhood index*. We choose hopscotch because of its high occupancy, cache efficient accesses, bounded lookup performance – even in high occupancy, and simple concurrency control [21].

We make two modifications to the original algorithm. First, we use a power of two number of entries, indexing the hash table similarly to set-associative caches [38]: we calculate the neighborhood index using the least-significant bits (LSB) of a fingerprint computed from the key. This allows efficiently reconstructing the original fingerprint during resize without needing to fully store it or perform IO to fetch the key and recompute it.

Second, we do not maintain a bitmap per neighborhood, nor a linked-list of entries per neighborhood, that the original algorithm suggests [37]. The latter would increase the memory requirements by 50% for the default configuration (8B entries, and neighborhood size of 32, 4B per entry). A linked list would at least double the memory requirement (assuming 8B pointers and singly or doubly linked list); let alone increase in complexity. Instead of using a bitmap or a list, we perform a linear probe directly on the entries both for lookup and insert.

Synchronization We use an array of locks for concurrency control. These locks protect different regions (*lock regions*) of the hash table, with a region being strictly larger than the neighborhood size (8192 entries by default). A lock is acquired based on the neighborhood's region; if a neighborhood spans two regions, a second lock is acquired in order. (The last neighborhoods do not wrap-around to the beginning of the table so lock order is maintained.) Moreover, to avoid inserts spanning more than two lock regions, we do not displace entries further than two regions apart. Hence, operations take two locks at maximum, and, assuming good key distribution, there is negligible lock contention.

Hash table entry Each hash table entry consists of 8 bytes:

```
struct HashEntry {
  uint64_t neigh_off:5;  // neighborhood offset
  uint64_t key_fp_tag:8; // fingerprint MSBs
  uint64_t kv_size:11;   // KV size (grains)
  uint64_t pba:40;       // storage addr. (grains)
};
```

The `pba` field contains the grain offset on storage where the KV pair resides. To allow utilization of large-capacity devices we use 40 bits for this field, thus able to index petabytes of storage (e.g., 4 PiB for 4 KiB grains). The `pba` value of all 1s indicates an invalid (free) entry.

We use 11 bits to store the size of the KV pair in grains (`kv_size`). This allows issuing a single IO read for `GET`s to KV pairs of up-to 8 MiB when using 4 KiB grains. KV pairs larger than that require a second operation. A valid entry with a KV size of 0 indicates a deleted entry.

The remaining 13 bits are used as follows. The in-memory index operates on a fingerprint of 35 bits, which are the LSBs of a 64 bit cityhash [14] hash of the key (Fig. 3). We divide the fingerprint into a *index* (27 bits) and a *tag* (8 bits). The index is used to index the hash table, allowing for a maximum of 2^{27} entries per table (the default). Reconstructing the fingerprint from a table location requires: i) the offset of the entry within the neighborhood, and ii) the fingerprint tag. We store both on the entry: 8 bits for the tag (`key_fp_tag`), and 5 bits to allow for 32 entries in a neighborhood (`neigh_off`). Hence, if an entry has location λ in the table, then its neighborhood index is $\lambda - \texttt{neigh_off}$, and its fingerprint is $\texttt{key_fp_tag} : (\lambda - \texttt{neigh_off})$.

Capacity utilization Effectively utilizing storage capacity requires being able to address it (`pba` field), but also having enough table entries. Using the LSBs of the tag (8 bits in total) to index the directory, uDepot's index allows for 2^8 tables, each with 2^{27} entries for a total of 2^{35} entries. At the cost of increased collisions, we can further increase the directory by also using up to 5 LSBs from the fingerprint to index it, allowing for 2^{13} tables. We can use up to the 5 neighborhood bits this way because the existing hopscotch collision mechanisms will end up filling positions in the table where no neighborhood starts. If we consider KV pairs with an average size of 1 KiB, this allows utilizing up to 1 PiB ($2^{35+5} \cdot 2^{10}$) of storage. Based on the expected workload and available capacity, users can maximize utilization by configuring the table size parameters accordingly.

Operations For *lookups*, a key fingerprint is generated. We use the fingerprint tag LSBs to index the directory and find the table for this key (if the fingerprint tag is not enough we also use the fingerprint LSBs as described above). Next, we index the table with the fingerprint index to find the neighborhood (also see: Fig. 3). A linear probe is then performed in the neighborhood, and the entries for which the

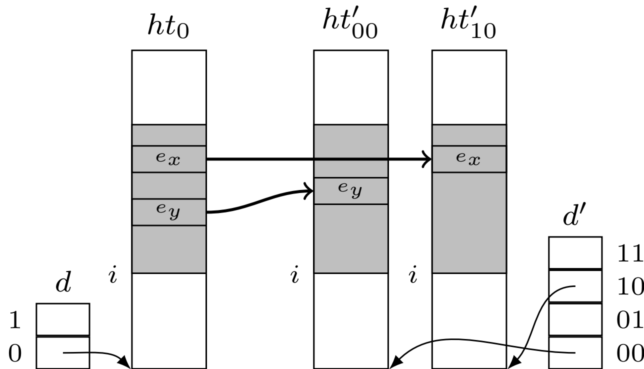


Figure 4: Incremental resizing example, transitioning from a directory with two hash tables (d) to a directory with four (d'). During resizing, insertions copy data from the lock region of ht_0 that contains the neighborhood for the inserted entry, to the same lock regions across two hash tables (ht'_{00}, ht'_{10}).

fingerprint tag (`key_fp_tag`) matches, if any, are returned.

For *inserts*, the hash table and neighborhood are located as described for the lookup. Then a linear probe is performed on the neighborhood and if no existing entry matches the fingerprint tag (`key_fp_tag`), then insert returns the first free entry, if one exists. The user may then fill the entry. If no free entry exists, then the hash table performs a series of displace attempts until a free entry can be found within the neighborhood. If this fails, an error is returned, at which point the caller usually triggers a resize operation. If matching entries exist, then insert returns them. The caller decides whether to update an entry in-place or continue the search for a free entry where they left off.

4.3 Resize operation

The optimal size of the index data structure depends on the number of KV records. Setting the size of the index data structure too low limits the number of records that can be handled. Setting the size too high could waste a significant amount of memory. For example, assuming an average KV record size of 1 KiB, a dataset of 1 PiB would require around 8 TB of memory.

uDepot avoids this issue by dynamically adapting the index data structure to the workload. The resize operation is fast, because it does not require any IO to the device, and causes minimal disruption to normal operations because it is executed incrementally.

The directory grows in powers of two, so that at any point the index holds $n * 2^m$ entries, where m is the number of grow operations, and n is the number of entries in each hash table. We only need the fingerprint to determine the new locations, so no IO operations are required to move hash entries to their new locations. A naive approach would be to move all entries at once, however, it would result in significant delays to user requests. Instead, we use an incremental approach (Fig. 4). During the resize phase, both the new and the old structures are maintained. We migrate entries from the old to the new structure at the granularity of the lock regions. A "migration" bit per lock indicates whether the region has already migrated. An atomic "resize" counter keeps track of whether the total resize operation has concluded, and is initialized to the total number of locks.

Migration is triggered by an insertion operation that fails to find a free entry. The first such failure triggers a resize operation, and sets up a new shadow directory. Subsequent insertion operations migrate all the entries under the locks they hold (one or two) to the new structure, setting the "migration" bit for each lock, and decrementing the "resize" counter (by one or two). Hash tables are pre-allocated during the resize operation in a separate thread to avoid delays. When all entries are migrated from the old to the new structure ("resize" count is zero), the memory of the old structure is released. During the resize operation, lookups need to check either the new or the old structure, depending on the lookup region's "migration" status.

4.4 Metadata and persistence

uDepot maintains metadata at three different levels: per device, per segment, and per KV record. At the device level the uDepot configuration is stored together with a unique seed and a checksum. At each segment's header, its configuration is stored (owning allocator, segment geometry, etc.) together with a timestamp and checksum that matches the device metadata. At the KV record (Fig. 2), uDepot prepends to each KV pair 6B of metadata containing the key size (2B) in bytes, and value size (4B) in bytes, and appends (to avoid the torn page problem) a 2B checksum matching the segment metadata (not computed over the data). The device and segment metadata require 128B and 64B, respectively, are stored in grain aligned locations and their overhead is negligible. The main overhead is due to the per KV metadata which depends on the average key-value size; for a 1 KiB average size the overhead amounts to 0.8%.

To speed up startup, in-memory index tables are flushed to persistent storage, but they are not guaranteed to be up-to-date: the persistent source of truth is the log. Flushing to storage occurs in normal shutdown, but also periodically to speed recovery. Upon initialization, uDepot iterates index segments, restores the index tables, and reconstructs the directory. If uDepot was cleanly shut down (we check this using checksums and unique session identifiers), the index is up to date. Otherwise, uDepot reconstructs the index from KV records found in KV segments. KV records for the same key (new values or tombstones) are disambiguated using segment version information. Because we are not reading data (only keys and metadata) during recovery, starting up after a crash typically takes a few seconds.

4.5 KV operations

For GET, a 64 bit hash of the key is computed and locking of the associated hash table region is performed. A lookup (see §4.2) is performed, returning zero or more matching hash entries. After the lookup, the table's region is unlocked. If no matching entry is found, the key does not exist. Otherwise, the KV record is fetched from storage for each matching entry; either a full key match is found and the value is returned, or the key does exist.

For PUT, we first write a KV record in the log out-of-place. Subsequently, we perform an operation similar to GET (key hash, lock, etc.) to determine whether the key already exists, using the insert (see §4.2) hash table function. If not, we insert a new entry to the hopscotch table if a free entry exists – if no free entry exists, then we trigger a resize operation. If a key already exists, we invalidate the grains of the previous entry, and update the table entry in-place with the new location (pba) and size of the KV record. Note that, also like GET, read IOs to matching hash table entries are performed without holding the table region lock. Unlike GET, though, PUT re-acquires the lock if the record is found, and repeats the lookup to detect concurrent mutation(s) on the same key: if such a concurrent mutation is detected, then the operation that updated the hash table entry first, wins. If the PUT fails, then it invalidates the grains it wrote before the lookup, and returns an appropriate error. PUT updates existing entries by default, but provides an optional argument where the user can choose instead to perform a PUT (i) only if the key exists, or (ii) only if the key does not exist.

DELETE is almost identical to PUT, other than it writes a tombstone entry instead of the KV record. Tombstone entries are used to identify deleted entries on a restore from the log, and are recycled during GC.

4.6 IO backends

uDepot bypasses the page cache and accesses the storage directly (O_DIRECT) by default. This prevents uncontrolled memory consumption, but also avoids scalability problems caused by concurrently accessing the page cache from multiple cores [96]. uDepot supports accessing storage both via synchronous IO and via asynchronous IO. Synchronous IO is implemented by the uDepot Linux backend (called so because scheduling is left to Linux). Despite its poor performance, this backend allows uDepot to be used by existing applications without modifications. For example, we have implemented a uDepot JNI interface that uses this backend. Its implementation is simple, since most operations directly translate to system calls. For asynchronous and user-space IO, uDepot uses TRT, and can use either SPDK or the kernel Linux AIO facility.

4.7 uDepot server

Embedded uDepot provides two interfaces to users: one where operations take arbitrary (contiguous) user buffers, and one where operations take a data structure that holds a linked list of buffers allocated from uDepot. The former interface, which internally is implemented using the latter, is simpler but is inherently inefficient. One of the problems is that for many IO backends it requires a data copy between IO buffers and the user-provided buffers. For instance, performing direct IO requires aligned buffers, while SPDK requires buffers allocated via its run-time system. Our server uses the second interface so that it can perform IO directly from (to) the receive (send) buffers. The server is implemented using TRT and uses the epoll backend for networking. First, a task for accepting new network connections is spawned. This task registers with the poller, and is notified when a new connection is requested. When this happens, the task will check if it should accept the new connection and spawn a new task on a (randomly chosen) TRT thread. The task will register with the local poller to be notified when there are incoming data for its connection. The connection task handles incoming requests by issuing IO operations to the storage backend (either Linux AIO or SPDK). After issuing an IO request, the task defers its execution and the scheduler runs another task. The storage poller is responsible for waking up the deferred task when the IO completion is available. The task will then send the proper reply and wait for a new request.

4.8 Memcache server

uDepot also implements the Memcache protocol [64], widely used to accelerate object retrieval from slower data stores (e.g., databases). The standard implementation of Memcache is in DRAM [65], but implementations for SSDs also exist [27, 61].

uDepot Memcache is implemented similarly to the uDepot server (§4.7): it avoids data copies, uses the epoll backend for networking and either the AIO or the SPDK backend for access to storage. Memcache specific KV metadata (e.g., expiration time, flags, etc.) are appended at the end of the value. Expiration is implemented in a lazy fashion: it is checked when a lookup is performed (either for a Memcache GET or a STORE command).

uDepot Memcache exploits synergies in the cache eviction and the space management GC design space: a merged cache eviction and GC process is implemented that reduces the GC cleanup overhead to zero in terms of IO amplification. Specifically, a GC LRU-policy is employed at the segment level (§4.1): on a cache hit the segment containing the KV is updated as the most recently accessed; when running low on free segments the least recently used one is chosen for cleanup, its valid KV entries (both expired and unexpired) are invalidated (i.e., evicted) in the uDepot directory, and the segment is now free to be re-filled, with-

out performing any relocation IO. This scheme allows us to maintain a steady performance even in the presence of sustained random updates, and also to reduce the overprovisioning at the space management level (SALSA) to a bare minimum (enough spare segments to accommodate the supported write-streams) thus maximizing capacity utilization at the space management level. A drawback of this scheme is potentially reduced cache hit ratio [81, 93]; we think this is a good tradeoff to make since the cache hit ratio is amortized by having a larger caching capacity due to the reduced overprovisioning. The uDepot memcache server is the basis of an experimental cloud memcache service currently available in the public cloud [39].

4.9 Implementation notes

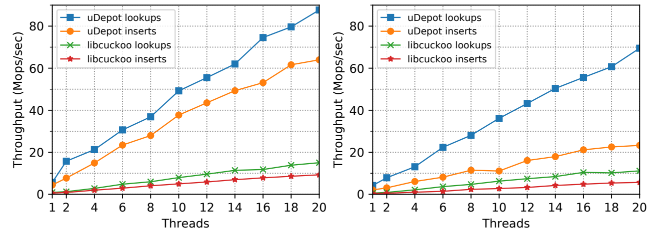
uDepot is implemented in C++11. It is worth noting that uDepot's performance requires many optimizations: we eliminate heap allocations from the data path using core-local slab allocators, we use huge pages, we favor static over dynamic polymorphism, we avoid copies using scatter-gather IO and placing data from the network at the proper location of IO buffers, we use batching, etc.

5 Evaluation

We perform our experiments on a machine with two 10-core Xeon CPUs (configured to operate at their maximum frequency: 2.2GHz), 125 GiB RAM, and running a 4.14 Linux kernel (including support for KPTI [16] – a mitigation for CPU security problems that increases context switch overhead). The machine has 26 NVMe drives: 2 Intel Optanes (P4800X 375GB HHHL PCIe), and 24 Intel Flash SSDs (P3600 400GB 2.5in PCIe).

5.1 Index structure

We start by evaluating the performance of our index structure both in the absence and presence of resize operations. We use 512 MiB (2^{26} entries) hash tables with 8192 locks per table. Our experiment consists of inserting a number of random keys, and then performing random lookups on those keys. We consider two cases: i) inserting 50M ($5 \cdot 10^7$) items where no resize happens, and ii) inserting 1B (10^9) items where four grow operations happen. We compare against libcuckoo [53, 54], a-state-of-the-art hash table implementation by running its accompanying benchmarking tool (`universal_benchmark`), configuring an initial capacity of $2^{26}/2^{30}$ for our 50M/1B runs. Results are shown in Fig. 5. For 50M items, our implementation achieves 87.7 million lookups and 64 million insertions per second, $\times 5.8$ and $\times 6.9$ better than libcuckoo, respectively. For 1B items, the insertion rate drops to 23.3 Mops/sec due to the resizing



(a) throughput: 50M items (no grow) (b) throughput: 1B items (4 grows)

percentile	lookup/50M	lookup/1B	insert/50M	insert/1B
50%	0.2 µs	0.3 µs	0.2 µs	0.4 µs
99%	1.1 µs	1.2 µs	0.6 µs	1.0 µs
99.9%	1.9 µs	2.0 µs	1.6 µs	9.2 µs
99.99%	11.0 µs	8.9 µs	7.5 µs	1168.0 µs

(c) Operation latencies

Figure 5: Mapping structure performance results.

operations. To better understand the cost of resizing, we perform another run where we sample latencies. Fig. 5c shows the resulting median and tail latencies. The latency of insert operations needing to copy items is seen in the 99.99% percentile, where latency is 1.17 ms. Note that this is a worse case scenario, where only insertions and no lookups are performed. It is possible to reduce the latency of these slow insertions by increasing the number of locks, at the cost of additional memory.

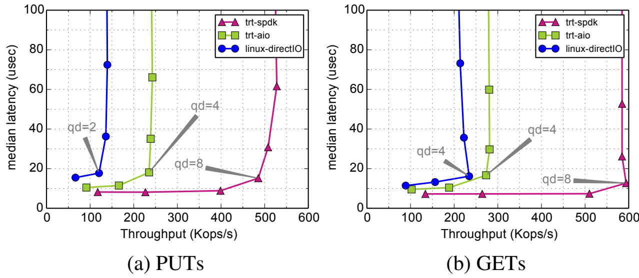
5.2 Embedded uDepot

Next, we examine the performance of uDepot as an embedded store. Our goal is to evaluate uDepot's ability to utilize FNDs, and compare the performance of the three different IO backends: syncronous IO using threads (`linux-directIO`), TRT using Linux asynchronous IO (`trt-aio`), and TRT using SPDK (`trt-spdk`). We are interested in two properties: *efficiency* and *scalability*. For the first, we restrain the application to use 1 core and 1 drive (§5.2.1). For the second, we use 24 drives and 20 cores (§5.2.2).

We use a custom microbenchmark to generate load for uDepot. We annotate the microbenchmark to sample the execution time for the operations performed, which we use to compute the median latency. In the following experiments, we use random keys of 8-32 bytes and values of 4K bytes. We perform 50M random `PUT`s, and 50M random `GET`s on the inserted keys.

5.2.1 Efficiency (one drive, one core)

We evaluate the efficiency of uDepot and its IO backends by using *one* core to drive *one* Optane drive. We compare uDepot's performance to the raw performance achievable by the device.

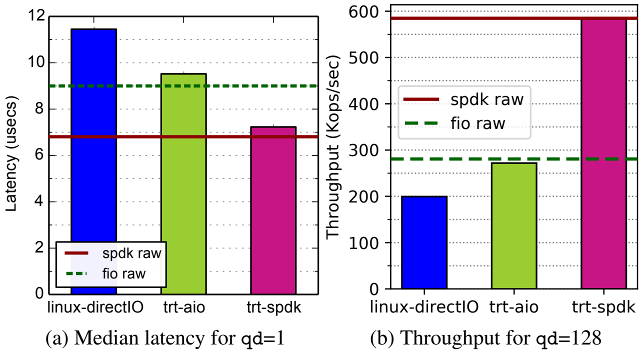


(a) PUTs

(b) GETs

Figure 6: uDepot running on a single core/single device setup. Median latency and throughput for a uniform random workload of 4K values for different IO backends and different queue depths.

We bind all threads on a single core (one that is on the same NUMA node as the drive). We apply the workload described in §5.2 for queue depths (qd) of $1, 2, 4, \ldots, 128$ and for the three different IO backends. For synchronous IO (`linux-directIO`) we spawn a number of threads equal to the qd. For TRT backends we spawn a single thread and a number of tasks equal to the qd. Both `linux-directIO` and `trt-aio` use direct IO to bypass the page cache.



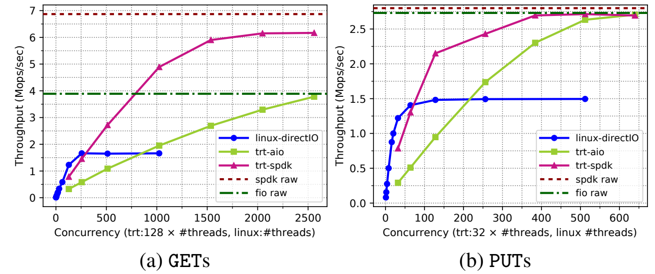
(a) Median latency for qd=1

(b) Throughput for qd=128

Figure 7: uDepot running on a single core/single device setup under a uniform random workload of GET operations for 4K values.

Results are shown in Fig. 6b for GETs and Fig. 6a for PUTs. The `linux-directIO` backend performs the worst. To a large extent, this is because it uses one thread per in-flight request, resulting in frequent context switches by the OS to allow all these threads to run on a single core. `trt-aio` improves performance by using TRT's tasks to perform asynchronous IO and perform a single system call for multiple operations. Finally, `trt-spdk` exhibits (as expected) the best performance as it avoids switching to the kernel.

We consider the better performing GET operations to compare uDepot against the device performance. We focus on latency with a single request in flight (qd = 1), and throughput at a high queue depth (qd = 128). Fig. 7a shows the median latency achieved for qd = 1 for each backend. The figure includes two lines depicting the raw performance



(a) GETs

(b) PUTs

Figure 8: Aggregate GET/PUT throughput of uDepot backends when using 24 NVMe drives for different concurrencies.

of the device under a similar workload obtained using appropriate benchmarks for each IO facility. That is, one core, one device, 4KiB random READ operations at qd = 1 across the whole device which was randomly written (preconditioned). `fio raw` shows the latency achieved by `fio` [23] with the `libaio` (i.e., Linux AIO) backend, while for `spdk raw` we use SPDK's `perf` utility [86]. uDepot under `trt-spdk` achieves a latency of 7.2 µs which is very close the latency of the raw device using SPDK (6.8 µs). The `trt-aio` backend achieves a latency of 9.5 µs with the corresponding raw device number using fio being 9 µs. An initial implementation of the `trt-aio` backend that used the `io_getevents()` system call to receive IO completions, resulted in a higher latency (close to 12 µs). We improved performance by implementing this functionality in user-space [17, 28, 77]. fio's latency remained unchanged when using this technique (fio option `userspace_reap`). Fig. 7b shows the throughput achieved by each backend at high (128) queue depth. `linux-directIO` achieves 200 kops/s, `trt-aio` 272 kops/s, and `trt-spdk` 585 kops/s. As before, `fio raw` and `spdk raw` show the device performance under a similar workload (4KiB random READs, qd=128) as reported by `fio` and SPDK's `perf`. Overall, uDepot performance is very close to the device performance.

5.2.2 Scalability (24 drives, 20 cores)

Next, we examine how well uDepot scales when using multiple drives and multiple cores, and how the different IO backends behave under these circumstances.

To maximize aggregate throughput, we use the 24 Flash-based NVMe drives in the system, and all of its 20 cores. (Even though these drives are not FNDs, we use a large number of them to achieve a high aggregate throughput and examine uDepot's scalability.) For the uDepot IO backends that operate on a block device (`linux-directIO` and `trt-aio`), we create a software RAID-0 device that combines the 24 drives into one using the Linux `md` driver. For the `trt-spdk` backend we use the RAID-0 uDepot SPDK backend. We use the workload described in §5.2, and take measurements for different numbers of concurrent requests.

For `linux-directIO` we use one thread per request, up to 1024 threads. For TRT backends, we use 128 TRT tasks per thread for GETs and 32 TRT tasks for PUTs (we use different numbers for different operations because they are saturated at different queue depths). We vary the number of threads from 1 up to 20.

Results are presented in Fig. 8. We also include two lines depicting the maximum aggregate throughput achieved on the same drives by SPDK `perf` and `fio` using the `libaio` (Linux AIO) backend. We focus on GETs, because that's the most challenging workload. The `linux-directIO` backend initially has better throughput as it uses more cores. For example, for a concurrency of 256, it uses 256 threads, and subsequently all the cores of the machine; for the TRT backends, the same concurrency uses 2 threads (128 tasks per thread), and subsequently 2 out of the 20 cores of the machine. Its performance, however, is capped at 1.66 Mops/s. The `trt-aio` backend achieves a maximum throughput of 3.78 Mops/s, which is very close to the performance achieved by `fio`: 3.89 Mops/s. Finally, `trt-spdk` achieves 6.17 Mops/s which is about 90% of the raw SPDK performance (6.87 Mops/s). We use normal SSDs to reach a larger throughput than the one that we could using Optane drives due to limited PCIe slots on our server. Because we measure throughput, these results can be generalized to FNDs with the difference being that it would require fewer drives to reach the achieved throughput. Moreover, the raw SPDK performance measured (6.87 Mops/s) is close to the throughput that the IO subsystem of our server can deliver: 6.91 Mops/s. The latter number is the throughput achieved by the SPDK benchmark when using uninitialized drives that return zeroes *without* accessing Flash. The PCIe bandwidth of our server is 30.8 GB/s (or 7.7 Mops/s for 4 KiB), which is consistent with our results if we consider PCIe and other overheads.

Overall, both uDepot backends (`trt-aio`, `trt-spdk`) perform very close in terms of efficiency and scalability to what the device can provide for each different IO facility. In contrast, using blocking system calls (`linux-directIO`) and multiple threads has significant performance limitations both in terms of throughput and latency.

5.3 uDepot server / YCSB

In this section we evaluate the performance of the uDepot server against two NoSQL stores: Aerospike [2] and ScyllaDB [82]. Even though uDepot has (by design) less functionality than these systems, we select them because they are NVMe-optimized and offer, to the best of our knowledge, the best options for exploiting FNDs today.

To facilitate a fair comparison, we use the YCSB [15] benchmark, and run the following workloads: A (update heavy: 50/50), B (read mostly: 95/5), C (read only), D (read latest), and F (read-modify-write), with 10M records and the

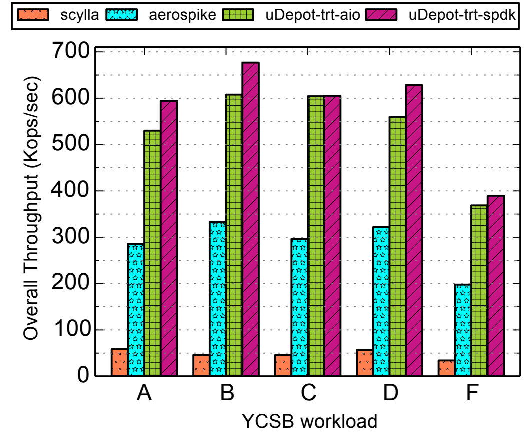


Figure 9: Overall throughput when using 256 YCSB client threads for different key-value stores.

default record size of 1 KiB. (We exclude workload E because uDepot does not support range queries.) We configure all systems to use two Optane drives and 10 cores (more than enough to drive 2 Optane drives), and generate load using a single client machine connected via 10 Gbit/sec Ethernet. For uDepot, we develop a YCSB driver using the uDepot JNI interface to act as a client. Because TRT is incompatible with the JVM, clients use the Linux uDepot backend. For Aerospike and ScyllaDB we use their available YCSB driver. We use YCSB version 0.14, Scylla version 2.0.2, and Aerospike version 3.15.1.4. For Scylla, we set the cassandra-cql driver's `core` and `maxconnections` parameters at least equal to the YCSB client threads, and capped its memory use to 64GiB to mitigate failing YCSB runs on high client thread counts due to memory allocation.

Fig. 9 presents the achieved throughput for 256 client threads for all workloads. uDepot using the `trt-spdk` backend improves YCSB throughput from $\times 1.95$ (workload D) up to $\times 2.1$ (workload A) against Aerospike, and from $\times 10.2$ (workload A) up to $\times 14.7$ (workload B) against ScyllaDB. Fig. 10 focuses on the update-heavy workload A (50/50), depicting the reported aggregate throughput, update and read latency for different number of client threads (up to 256) for all the examined stores. For 256 clients, uDepot using SPDK achieves a read/write latency of 345 µs/467 µs, Aerospike 882 µs/855 µs, and ScyllaDB 4940 µs (3777 µs).

We profile execution under workload A, to understand the causes of the performance differences between Aerospike, ScyllaDB, and uDepot. Aerospike is limited by its use of multiple IO threads and synchronous IO. Indeed, synchronization functions occupied a significant amount of its execution time due to contention created by the multiple threads. ScyllaDB uses asynchronous IO (and in general has an efficient IO subsystem), but it exhibits significant IO amplification. We measured the read IO amplification of the user data (YCSB key and value) versus what was read from the FNDs

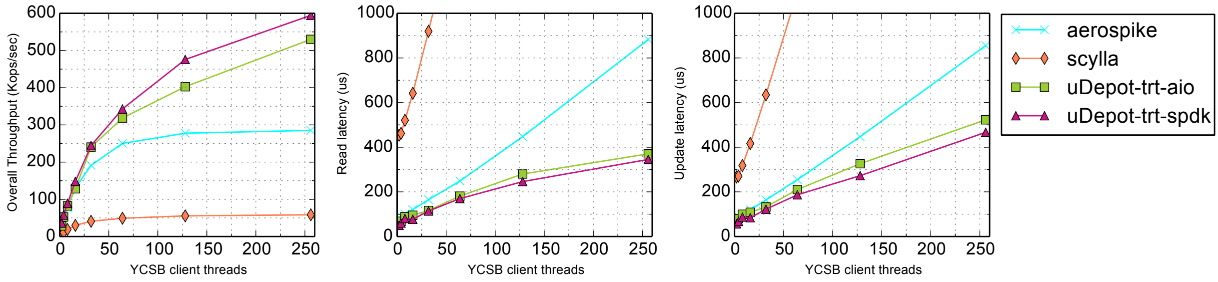


Figure 10: Overall throughput, update and read latency, as reported by the YCSB benchmark for different number of client threads applying workload A (50/50 reads/writes) to different key-value stores.

and the results were as follows: ScyllaDB: $\times 8.5$, Aerospike: $\times 2.4$, and uDepot (TRT-aio): $\times 1.5$.

Overall, uDepot exposes the performance of FNDs significantly better than Aerospike and ScyllaDB. We note that YCSB is inefficient since it uses synchronizing Java threads with synchronous IO, and under-represents uDepot's performance. In the next section, we use a more performant benchmark that better illustrates uDepot's efficiency.

5.4 uDepot Memcache

Lastly, we evaluate the performance of our uDepot Memcache implementation, and investigate if it can provide comparable performance to DRAM-based services.

We use memcached [65] (version: 1.5.4), the standard implementation of Memcache that uses DRAM, as the standard on what applications using Memcache expect, MemC3 [25] (commit: `84475d1`), a state-of-the-art Memcache implementation, and Fatcache [27] (commit: `512caf3`), a Memcache implementation on SSDs.

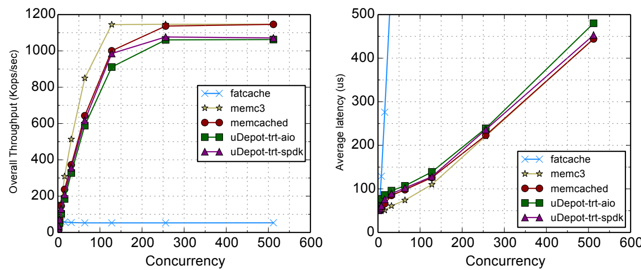


Figure 11: Memcache performance as reported by memaslap using the default 10%-`PUT`, 90%-`GET` workload of 1 KiB objects for different number of clients (concurrency).

We use memaslap[4] [62], a standard Memcache benchmark, and generate the default workload: 10%-`PUT`, 90%-`GET` with 1 KiB objects. We execute memaslap on a different machine, connected over 10 Gbit/s Ethernet to the server. The Memcache servers are configured to use all 20 cores of our machine. DRAM-based memcached, and MemC3 are configured to use enough memory to fit all the working set, while Fatcache and uDepot are configured to use the two Optane drives in a RAID-0 configuration, using the Linux md driver when required. We use the default options for Fatcache.

The reported latency and throughput is summarized in Fig. 11. For a single client, the reported latency is 49 µs for MemC3, 51 µs for both `memcached` and uDepot using `trt-spdk`, 52 µs for Fatcache, and 67 µs for uDepot using `trt-aio`. Contrarily to uDepot, Fatcache caches data in DRAM which leads to the low latency at low queue depths. As the number of clients increase, however, the performance of Fatcache significantly diverges, while uDepot's performance remains close. Case in point, for 128 clients, MemC3's latency is 110 µs, memcached's 126 µs, uDepot with `trt-spdk` achieves 128 µs, uDepot with `trt-aio` 139 µs, and Fatcache 2418 µs; The achieved throughputs are: MemC3:1145 kops/s, memcached:1001 kops/s, uDepot `trt-spdk`: 985 kops/s uDepot `trt-aio`: 911 kops/s, and Fatcache: 53 kops/s.

Hence, our results show that `memcached` on DRAM can be replaced with uDepot on NVM with a negligible performance hit, since the bottleneck is the network. Moreover, Fatcache cannot exploit the performance benefits of FNDs.

6 Related work

Flash KV stores Two early KV stores that specifically targeted Flash are FAWN [3], a distributed KV store, built with low-power CPUs and small amounts of Flash storage, and FlashStore [19], a multi-tiered KV store using both DRAM, Flash, and Disks. These systems are similar to uDepot in that they keep an index in the form of a hash-table in DRAM, and they use a log-structured approach. They both use 6-byte entries: 4 bytes to address Flash, and 2 bytes for they key fingerprint, while subsequent evolutions of these works [20,56] further reduce the entry size. uDepot increases the entry to 8 bytes, enabling features not supported by the above systems: i) uDepot stores the size of the KV entry, allowing it to fetch both key and value with a single read request. That

[4] We applied a number of scalability patches [63] to improve performance.

is, a GET operation requires a single access. ii) uDepot supports online resizing that does not require accessing NVM storage. iii) uDepot uses 40 instead of 32 bits for addressing storage, supporting up to 1 PB of grains. Moreover, uDepot efficiently accesses FNDs (via asynchronous IO backends) and scales over many devices and cores which these systems, built for slower devices, do not support. A number of works [60,91] built Flash KV stores or caches [81,84] that rely on non-standard storage devices, such as open-channel SSDs. uDepot does not depend on special devices, and using richer storage interfaces to improve uDepot is future work.

High-performance DRAM KV stores A large number of works targets to maximize the performance of DRAM-based KV stores using RDMA [21,44,68,73], direct access to network hardware [57], or, FPGAs [10, 52]. uDepot, on the other hand, operates over TCP/IP and places data in storage devices. Nevertheless, many of these systems use a hash-table to maintain their mapping, and access it with one-sided RDMA operations from the client when possible. FaRM [21], for example, identifies the problems of cuckoo hashing, and, similarly to uDepot, uses a variant of hopscotch hashing. A fundamental difference of FaRM and uDepot is that the former is concerned with minimizing RDMA operations to access the hash table, which is not a concern for uDepot. Moreover, uDepot's index structure supports online resizing, while FaRM uses an overflow chain per bucket that can cause a performance hit for checking the chain.

NVM KV stores A number of recent works [4,71,92,95] propose NVM KV stores. These systems are fundamentally different in that they operate on byte-addressable NVM placed on the memory bus. uDepot, instead, uses NVM on storage devices because the technology is widely available and more cost effective. MyNVM [22] also uses NVM storage as a way to reduce the memory footprint of RocksDB, where NVM storage is introduced as a second level block cache. uDepot takes a different approach by building a KV store that places data exclusively on NVM. Aerospike [87], that targets NVMe SSDs, follows a similar design to uDepot by keeping its index in DRAM and the data in a log that resides in storage. Nevertheless, because it is designed with SSDs in mind, it cannot fully exploit the performance of FNDs (e.g., it uses synchronous IO). Faster [11] is a recent KV store that, similarly to uDepot, maintains an resizable in-memory hash index and stores its data into a log. In contrast to uDepot, Faster uses a hybrid log that resides both in DRAM and in storage.

Memcache Memcache is an extensively used service [5, 6, 32, 65, 70]. MemC3 [25] redesigns memcached using a concurrent cuckoo hashing table. Similarly to the original memcached, the hash table cannot be dynamically resized and the amount of used memory must be defined when the service starts. uDepot supports online resizing of the hash table, while also allowing for faster warm-up times if the service restarts since the data are stored in persistent storage. Recently, usage of FNDs in memcached was explored as means to reduce costs and expand the cache [66].

Task-based asynchronous IO A long-standing debate exists on programming asynchronous IO using threads versus events [1, 18, 49, 51, 90]. uDepot is built on TRT that uses a task-based approach, where each task has its own stack. A useful extension to TRT would be to provide a composable interface for asynchronous IO [36]. Flashgraph [97] uses an asynchronous task-based IO system to process graphs stored on Flash. Seastar [83], the run-time used by ScyllaDB, follows the same design principles as TRT, but does not (currently) support SPDK.

7 Conclusion and Future Work

We presented uDepot, a KV store that fully utilizes the performance of fast NVM storage devices like Intel Optane. We showed that uDepot reaches the performance available from the underlying IO facility it uses, and can better utilize these new devices compared to existing systems. Moreover, we showed that uDepot can use these devices to implement a cache service that achieves a similar performance to DRAM implementations, at a much lower cost. Indeed, we use our uDepot Memcache implementation as the basis of an experimental public cloud service [39].

uDepot has two main limitations that we plan to address in future work. First, uDepot does not (efficiently) support a number operations that have been proven useful for applications such as range queries, transactions, checkpoints, data structure abstractions [78], etc. Second, there are many opportunities to improve efficiency by supporting multiple tenants [13], that uDepot does not currently exploit.

8 Acknowledgements

We would like to thank the anonymous reviewers and especially our shepherd, Peter Macko, for their valuable feedback and suggestions, as well as Radu Stoica for providing feedback on early drafts of our paper. Finally, we would like to thank Intel for providing early access to an Optane testbed.

References

[1] Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. Cooperative task management without manual stack management. USENIX ATC '02.

[2] Aerospike — high performance NoSQL database. https://www.aerospike.com/.

[3] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. FAWN: A fast array of wimpy nodes. SOSP '09.

[4] Arulraj, J., Levandoski, J., Minhas, U. F., and Larson, P.-A. Bztree: a high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow. 11*, 5 (2018).

[5] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. SIGMETRICS '12.

[6] Amazon ElastiCache. https://aws.amazon.com/elasticache/.

[7] Barroso, L., Marty, M., Patterson, D., and Ranganathan, P. Attack of the killer microseconds. *Commun. ACM 60*, 4 (2017).

[8] Oracle Berkeley DB. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.

[9] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A protected dataplane operating system for high throughput and low latency. OSDI '14.

[10] Chalamalasetti, S. R., Lim, K., Wright, M., AuYoung, A., Ranganathan, P., and Margala, M. An FPGA memcached appliance. FPGA '13.

[11] Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., and Barnett, M. FASTER: an embedded concurrent key-value store for state management. *Proceedings of the VLDB Endowment* (2018).

[12] Chang, L.-P., Kuo, T.-W., and Lo, S.-W. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst. 3*, 4 (2004).

[13] Cidon, A., Rushton, D., Rumble, S. M., and Stutsman, R. Memshare: a dynamic multi-tenant key-value cache. USENIX ATC '17.

[14] CityHash, a family of hash functions for strings. https://github.com/google/cityhash.

[15] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. SoCC '10.

[16] Corbet, J. The current state of kernel page-table isolation. https://lwn.net/Articles/741878/, Dec. 2017.

[17] Corbet, J. A new kernel polling interface. https://lwn.net/Articles/743714/, 2018.

[18] 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* (2002).

[19] Debnath, B., Sengupta, S., and Li, J. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow. 3*, 1-2 (2010).

[20] Debnath, B., Sengupta, S., and Li, J. Skimpystash: RAM space skimpy key-value store on flash-based storage. SIGMOD '11.

[21] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. FaRM: fast remote memory. NSDI '14.

[22] Eisenman, A., Gardner, D., AbdelRahman, I., Axboe, J., Dong, S., Hazelwood, K., Petersen, C., Cidon, A., and Katti, S. Reducing DRAM footprint with NVM in Facebook. EuroSys '18.

[23] Flexible I/O tester, https://linux.die.net/man/1/fio.

[24] Facebook. RocksDB — a persistent key-value store. http://rocksdb.org.

[25] Fan, B., Andersen, D. G., and Kaminsky, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. NSDI '13.

[26] Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W. SAP HANA database: Data management for modern business applications. *SIGMOD Rec. 40*, 4 (2012).

[27] fatcache. https://github.com/twitter/fatcache.

[28] fio `user_io_getevents()` implementation. https://github.com/axboe/fio/blob/fio-3.3/engines/libaio.c#L120.

[29] C++ documentation: std::future. http://en.cppreference.com/w/cpp/thread/future.

[30] Java documentation: java.util.concurrent: Future. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html.

[31] Google. LevelDB. https://github.com/google/leveldb.

[32] App engine memcache service. https://cloud.google.com/appengine/docs/standard/python/memcache/.

[33] Graefe, G., Volos, H., Kimura, H., Kuno, H., Tucek, J., Lillibridge, M., and Veitch, A. In-memory performance for big data. *Proc. VLDB Endow. 8*, 1 (2014).

[34] Grupp, L. M., Davis, J. D., and Swanson, S. The bleak future of nand flash memory. FAST '12.

[35] Harizopoulos, S., Abadi, D. J., Madden, S., and Stonebraker, M. OLTP through the looking glass, and what we found there. SIGMOD '08.

[36] Harris, T., Abadi, M., Isaacs, R., and McIlroy, R. AC: Composable asynchronous io for native languages. OOPSLA '11.

[37] Herlihy, M., Shavit, N., and Tzafrir, M. Hopscotch hashing. DISC '08.

[38] Hill, M. D., and Smith, A. J. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38*, 12 (1989).

[39] "IBM". Data store for memcache. https://cloud.ibm.com/catalog/services/data-store-for-memcache.

[40] Are the major dram suppliers stunting dram demand? http://www.icinsights.com/news/bulletins/Are-The-Major-DRAM-Suppliers-Stunting-DRAM-Demand/. Accessed: 2018-09-10.

[41] Ioannou, N., Kourtis, K., and Koltsidas, I. Elevating commodity storage with the SALSA host translation layer. MASCOTS '18.

[42] io_getevents(2) - read asynchronous i/o events from the completion queue. http://man7.org/linux/man-pages/man2/io_getevents.2.html.

[43] io_submit(2) - submit asynchronous I/O blocks for processing. http://man7.org/linux/man-pages/man2/io_submit.2.html.

[44] Kalia, A., Kaminsky, M., and Andersen, D. G. Design guidelines for high performance RDMA systems. USENIX ATC '16.

[45] Kegel, D. The c10k problem. http://www.kegel.com/c10k.html, 2014.

[46] Kerrisk, M. *The Linux Programming interface.* 2010.

[47] Kim, H.-J., Lee, Y.-S., and Kim, J.-S. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. HotStorage '16.

[48] Koh, S., Lee, C., Kwon, M., and Jung, M. Exploring system challenges of ultra-low latency solid state drives. HotStorage '18.

[49] Krohn, M., Kohler, E., and Kaashoek, M. F. Events can make sense. USENIX ATC '07.

[50] Kyoto cabinet: a straightforward implementation of dbm. http://fallabs.com/kyotocabinet/, 2011.

[51] Lauer, H. C., and Needham, R. M. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev. 13*, 2 (1979).

[52] Li, B., Ruan, Z., Xiao, W., Lu, Y., Xiong, Y., Putnam, A., Chen, E., and Zhang, L. KV-Direct: high-performance in-memory key-value store with programmable NIC. SOSP '17.

[53] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. EuroSys '14.

[54] libcuckoo. https://github.com/efficient/libcuckoo. Accessed: 2018-09-20.

[55] libevent – an event notification library. http://libevent.org/. Accessed: 2017-02-27.

[56] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A memory-efficient, high-performance key-value store. SOSP '11.

[57] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A holistic approach to fast in-memory key-value storage. NSDI '14.

[58] LU, L., PILLAI, T. S., GOPALAKRISHNAN, H., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: separating keys from values in SSD-conscious storage. *Trans. Storage 13*, 1 (2017).

[59] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. EuroSys '12.

[60] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. NVMKV: A scalable and lightweight flash aware key-value store. HotStorage '14.

[61] Mcdipper: A key-value cache for flash storage. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/, 2013.

[62] memaslap - Load testing and benchmarking a server. http://docs.libmemcached.org/bin/memaslap.html.

[63] Scalability issues with memaslap client. https://bugs.launchpad.net/libmemcached/+bug/1721048.

[64] Memcache protocol. https://github.com/memcached/memcached/wiki/Protocols. Retrieved Oct 2017.

[65] memcached – a distributed memory object caching system. http://www.memcached.org/.

[66] Caching beyond RAM: the case for NVMe. https://memcached.org/blog/nvm-caching/. Accessed: 2019-12-15.

[67] MENON, J. A performance comparison of RAID-5 and log-structured arrays. In *High Performance Distributed Computing* (1995).

[68] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. USENIX ATC '13.

[69] MUTLU, O., AND SUBRAMANIAN, L. Research problems and opportunities in memory systems. *Supercomput. Front. Innov.: Int. J. 1*, 3 (2014).

[70] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. NSDI '13.

[71] OUKID, I., LASPERAS, J., NICA, A., WILLHALM, T., AND LEHNER, W. FPTree: a hybrid SCM-DRAM persistent and concurrent B-Tree for storage class memory. SIGMOD '16.

[72] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for RAMClouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev. 43*, 4 (2010).

[73] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The RAMCloud storage system. *ACM Trans. Comput. Syst. 33*, 3 (2015).

[74] PAIK, Y. Developing extremely low-latency NVMe SSDs. Flash Memory Summit, 2017. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170809_FA21_Paik.pdf.

[75] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. OSDI '14.

[76] PLETKA, R., KOLTSIDAS, I., IOANNOU, N., TOMIĆ, S., PAPANDREOU, N., PARNELL, T., POZIDIS, H., FRY, A., AND FISHER, T. Management of next-generation nand flash to achieve enterprise-level endurance and latency targets. *ACM Trans. Storage 14*, 4 (2018).

[77] qemu io_getevents_peek() and io_getevents_commit() implementation. https://git.qemu.org/?p=qemu.git;a=blob;f=block/linux-aio.c;h=88b8d55ec71076e24436ba4a80ec6de4d711e896;hb=HEAD#l131.

[78] Redis. http://redis.io/.

[79] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992).

[80] RUMBLE, S. M., KEJRIWAL, A., AND OUSTERHOUT, J. Log-structured memory for dram-based storage. FAST '14.

[81] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. Flashtier: A lightweight, consistent and durable storage cache. EuroSys '12.

[82] ScyllaDB. http://www.scylladb.com/.

[83] Seastar: High performance server-side application framework. http://www.seastar-project.org/. Accessed: 2017-03-01.

[84] SHEN, Z., CHEN, F., JIA, Y., AND SHAO, Z. DIDACache: A deep integration of device and application for flash based key-value caching. FAST '17.

[85] Storage performance development kit. http://www.spdk.io/.

[86] Spdk perf. https://github.com/spdk/spdk/blob/master/examples/nvme/perf/perf.c.

[87] SRINIVASAN, V., BULKOWSKI, B., CHU, W.-L., SAYYAPARAJU, S., GOODING, A., IYER, R., SHINDE, A., AND LOPATIC, T. Aerospike: Architecture of a real-time operational dbms. *Proc. VLDB Endow.* (2016).

[88] TALLIS, B. The intel Optane SSD DC P4800X (375GB) review: Testing 3D XPoint performance. http://www.anandtech.com/show/11209/intel-optane-ssd-dc-p4800x-review-a-deep-dive-into-3d-xpoint-enterprise-performance, 2017.

[89] TALLIS, B. Samsung launches Z-SSD SZ985: Up to 800gb of Z-NAND. https://www.anandtech.com/show/12376/samsung-launches-zssd-sz985-up-to-800gb-of-znand, 2018.

[90] VON BEHREN, R., CONDIT, J., AND BREWER, E. Why events are a bad idea (for high-concurrency servers). HOTOS '03.

[91] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. EuroSys '14.

[92] XIA, F., JIANG, D., XIONG, J., AND SUN, N. HiKV: a hybrid index Key-Value store for DRAM-NVM memory systems. USENIX ATC '17.

[93] XIA, Q., AND XIAO, W. High-performance and endurable cache management for flash-based read caching. *IEEE Transactions on Parallel and Distributed Systems 27*, 12 (2016).

[94] YANG, J., MINTURN, D. B., AND HADY, F. When poll is better than interrupt. FAST'12.

[95] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. NV-Tree: reducing consistency cost for NVM-based single level systems. FAST '15.

[96] ZHENG, D., BURNS, R., AND SZALAY, A. S. A parallel page cache: Iops and caching for multicore systems. HotStorage '12.

[97] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. Flashgraph: Processing billion-node graphs on an array of commodity SSDs. FAST '15.

Notes: IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Intel, Intel Xeon, and Intel Optane are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Other products and service names might be trademarks of IBM or other companies.

Optimizing Systems for Byte-Addressable NVM by Reducing Bit Flipping

Daniel Bittman
UC Santa Cruz

Peter Alvaro
UC Santa Cruz

Darrell D. E. Long
UC Santa Cruz

Ethan L. Miller
UC Santa Cruz
Pure Storage

Abstract

New byte-addressable non-volatile memory (BNVM) technologies such as phase change memory (PCM) enable the construction of systems with large persistent memories, improving reliability and potentially reducing power consumption. However, BNVM technologies only support a limited number of lifetime writes per cell and consume most of their power when flipping a bit's state during a write; thus, PCM controllers only rewrite a cell's contents when the cell's value has changed. Prior research has assumed that reducing the number of *words* written is a good proxy for reducing the number of *bits* modified, but a recent study has suggested that this assumption may not be valid. Our research confirms that approaches with the fewest writes often have *more* bit flips than those optimized to reduce bit flipping.

To test the effectiveness of bit flip reduction, we built a framework that uses the number of bits flipped over time as the measure of "goodness" and modified a cycle-accurate simulator to count bits flipped during program execution. We implemented several modifications to common data structures designed to reduce power consumption and increase memory lifetime by reducing the number of bits modified by operations on several data structures: linked lists, hash tables, and red-black trees. We were able to reduce the number of bits flipped by up to $3.56\times$ over standard implementations of the same data structures with negligible overhead. We measured the number of bits flipped by memory allocation and stack frame saves and found that careful data placement in the stack can reduce bit flips significantly. These changes require no hardware modifications and neither significantly reduce performance nor increase code complexity, making them attractive for designing systems optimized for BNVM.

1 Introduction

As byte-addressable non-volatile memories (BNVMs) become common [15, 18, 24], it is increasingly important that systems are optimized to leverage their strengths and avoid stressing their weaknesses. Historically, such optimizations have included reducing the number of writes performed, either by designing data structures that require fewer writes or by using hardware techniques such as caching to reduce writes. However, it is the number of *bits flipped* that matter most for BNVMs such as phase-change memory (PCM), *not* the number of words written.

BNVMs such as PCM suffer from two problems caused by flipping bits: energy usage and cell wear-out. As these memory technologies are adopted into longer-term storage solutions and battery powered mobile and IoT devices, their costs become dominated by physical replacement from wear-out and energy use respectively, so increasing lifetime and dropping power consumption are vital optimizations for BNVM. Flipping a bit in a PCM consumes $15.7-22.5\times$ more power than reading a bit or "writing" a bit that does not actually change [13, 14, 24, 29]. Thus, many controllers optimize by only flipping bits when the value being written to a cell differs from the old value [39]. While this approach saves some energy, it cannot eliminate flips required by software to update modified data structures. An equally important concern is that PCM has limited endurance: cells can only be written a limited number of times before they "wear out". Unlike flash, however, PCM cells are written individually, so it is possible (and even likely) that some cells will be written more than others during a given period because of imbalances in values written by software. Reducing bit flips, an optimization goal that has yet to be sufficiently explored, can thus both save energy and extend the life of BNVM.

Previously, we showed that small changes in data structures can have large impacts in the bit flips required to complete a given set of data structure modifications [4]. While it is possible to reduce bits flipped with changes to hardware, we can gain more by optimizing compiler constructs and choosing data structures to take advantage of semantic information that is not available at other layers of the stack; it is critical we design our data structures with this in mind. Successful BNVM-optimized systems will need to target new optimizations for BNVM, including bit flip reduction.

We implemented three such data structures and evaluated the impact on the number of writes and bit flips, demonstrating the effectiveness of designing data structures to minimize bit flips. These simple changes reduce bit flips by as much as 3.56×, and therefore will reduce power consumption and extend lifetime by a proportional amount, with no need to modify the hardware in any way. Our contributions are:

- Implementation of bit flip counting in a full cycle-accurate simulation environment to study bit flip behavior.
- Empirical evidence that reducing memory writes may not reduce bit flips proportionally.
- Measurements of the number of bit flips required by operations such as memory allocation and stack frame use, and suggestions for reducing the bit flips they require.
- Modification of three data structures (linked lists, hash tables, red-black trees) to reduce bit flips and evaluation of the effectiveness of the techniques.

The paper is organized as follows. Section 2 gives background demonstrating how bit flips impact power consumption and BNVM lifetime. Section 3 discusses some techniques for reducing bit flips in software, which are evaluated for bit flips (Section 4) and performance (Section 5). Section 6 discusses the results, followed by comments on future work (Section 7) and a conclusion (Section 8).

2 BNVM and Bit Flips

Non-volatile memory technologies [6] such as phase-change memory (PCM) [24], resistive RAM (RRAM, or memristors) [33, 35], Ferroelectric RAM (FeRAM) [15], and spin-torque transfer RAM (STT-RAM) [22], among others, have the potential to fundamentally change the design of devices, operating systems, and applications. Although these technologies are starting to make their way into consumer devices [18] and embedded systems [33], their full potential will be seen when they replace or coexist with DRAM as byte-addressable non-volatile memory (BNVM). Such a memory hierarchy will allow the processor, and thus applications, to use load and store instructions to update persistent state, bypassing the high-latency I/O operations of the OS. However, power consumption, especially for write operations, and device lifetime are more serious concerns for these technologies than for existing memory technologies.

2.1 Optimizing for Memory Technologies

Data structures should be designed to exploit the advantages and mitigate the disadvantages of the technologies on which they are deployed. For example, data structures for disks are block-oriented and favor sequential access, while those designed for flash reduce writes, especially random writes, often by trading them for an increase in random reads [10]. Prior data structures and programming models for NVM [9, 11, 16, 25, 36, 38] have typically exploited its byte-addressability while mitigating the relatively slow access times of most BNVM technologies. However, in the case of technologies such as PCM or RRAM, existing research ignores two critical characteristics: asymmetric read/write power usage and the ability to avoid rewriting individual bits that are unchanged by a write [6, 39].

For example, writes to PCM are done by melting a cell's worth of material with a relatively high current and cooling it at two different rates, leaving the material in either an amorphous or crystalline phase [30]. These two phases have different electrical resistance, each corresponding to a bit value of zero or one. The writing process takes much more energy than reading the phase of the cell, which is done by sensing the cell's resistance with a relatively low current. To save energy, the PCM controller can avoid writing to a cell during a write if it already contains the desired value [39], meaning that the major component of the power required by a write is proportional not to the number of bits (or words) *written*, but rather to the number of bits *actually flipped* by the write. Based on this observation, we should design data structures for BNVM to minimize the number of bits flipped as the structures are modified and accessed rather than simply reducing the number of writes, as is more commonly done.

2.2 Power Consumption of PCM and DRAM

While our research applies to any BNVM technology in which writes are expensive, we focus on PCM because its power consumption figures are more readily available. Figure 1 shows the estimated power consumption of 1 GB of DRAM and PCM as a function of bits flipped per second, using power measurements from prior studies of memory systems [4, 7, 13, 14, 24, 29]. The number of writes to DRAM has little effect on overall power consumption since the *entire* DRAM must be periodically refreshed (read and rewritten); refresh dominates, resulting in a high power requirement regardless of the number of writes. In contrast, PCM requires no "maintenance" power, but needs a great deal more energy to write an individual bit (~50 pJ/b [2]) compared to the low overhead for writing a DRAM page (~1 pJ/b [24]). The result is that power use for DRAM is largely proportional to memory *size*, while power consumption for PCM is largely proportional to *cell change rate*. The exact position of the cross-over point in Figure 1 will be narrowed down as these devices become more common; many features of these devices, including asymmetric write-zero and write-one costs, increased density of PCM over DRAM, and decreasing feature sizes, will affect the trade-off point over time.

Figure 1 demonstrates the need for data structures for PCM to minimize cell writes. Because the memory controller can minimize the cost of "writing" a memory cell with the same value it already contains, the primary concern for

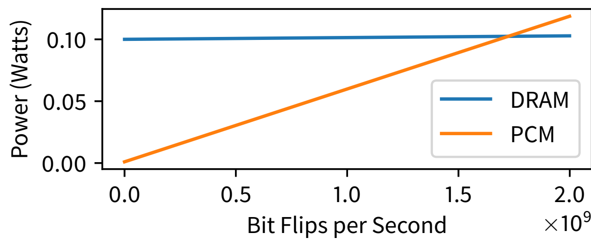


Figure 1: Power use as a function of flips per second [4].

data structures in PCM is reducing the number of bit flips, which the memory controller cannot easily eliminate.

Power consumption is particularly concerning for battery-operated Internet of Things (IoT) devices, which may become a significant consumer of BNVM technologies to facilitate fast power-up and reduce idle power consumption [20, 21]. Devices that collect large amounts of data and write frequently to BNVM may find power usage increasing depending on access patterns. Thus, IoT devices may benefit significantly from bit-flip-aware systems and data structures.

2.3 Wear-out

Another significant advantage to avoiding bit flips is reducing memory cell wear-out. BNVM technologies typically have a maximum number of lifetime writes, and fewer writes means a longer lifetime. However, by avoiding unnecessary overwrites, the controller would introduce uneven wear *within* BNVM words where some of the bits flip more frequently than others due to biases of certain writes. For example, pointer overwrites may only alter the low-order bits, except for the few that are zero because of structure alignment in memory, if the pointers are to nearby regions. Thus, the middle bits in a 64-bit word may wear out faster than the lowest and highest bits. While reducing bit-flips increases the average lifetime of the cells in a word, it has the potential to exacerbate the uneven wear problem since such techniques might increase the biases of certain writes.

Fortunately, we can take advantage of existing research in wear-leveling for BNVM that allows the controller to spread out the cell updates within a given word. While a full remapping layer similar to a flash translation layer is infeasible for BNVM—the overhead would be too high—hardware techniques such as row shifting [40], content-aware bit shuffling [17], and start-gap wear leveling [28] may be able to mitigate biased write patterns with low overhead. This would allow BNVM to leverage bit flip reduction to reduce wear even if the result is that some bits are flipped more frequently than others. These techniques, implemented at the memory controller level, can work in tandem with the techniques described in this paper since they benefit bit flip reduction and can distribute "hot" bits across a word, mitigating the biased write patterns bit flip reduction techniques may introduce.

2.4 Reducing Impact of Bit Flips in BNVM

Although bit flips in BNVM have been studied previously, much of that work has focused on hardware encoding, which re-encodes cache lines to reduce bit flips, but re-encoding has limited efficacy [8, 19, 32] because it must also store information on *which* encoding was used. While hardware techniques are worth exploring, software techniques to reduce bit flips can be more effective because they can leverage semantic knowledge available in the software but not visible in the memory controller's limited view of single cache lines.

Chen *et al.* [7] evaluate data structures on BNVM and argue that reducing bit flips is workload dependent and difficult to reason about, so we should strive to reduce writes because writes are approximately proportional to bit flips. We found that this is often *not* the case—our prior experiments revealed that bit flips were often *not* proportional to writes, and we were able to examine bit flips and optimize for them in an example data structure [4]. These findings are further corroborated by our experiments in Section 4.

Since bit flips directly affect power consumption and wear, we can study three separate aspects for bit flip reduction:

- **Data structure design**: Since data organization plays a large role in the writes that make it to memory, we designed new data structures built around the idea of *pointer distance* [34] instead of storing pointers directly. While *data* writes themselves significantly affect bit flips, these writes are often unavoidable (since the data must be written), while data *structure* writes are more easily optimized (as we see in existing BNVM data structure research). Furthermore, data structures often require a significant number of updates over time, while data is often written once (since we can reduce writes by updating pointers instead of moving data). Thus the overall proportion of bit flips caused by data writes may drop over time as data structures are updated.
- **Effects of program operation**: A common source of writes is the stack, where return addresses, saved registers, and register spills are written. Understanding how these writes affect bit flips plays a critical role in recommendations for bit flip reduction for system designers.
- **Effects of caching layers**: Since writes must first go through the cache, it is vital to understand how different caching layers and cache sizes affect bit flips in memory. Complicating matters is the unique consistency challenges of BNVM [9, 11, 36], wherein programs often flush cache-lines to main memory more frequently than they otherwise would, use write-through caching, or more complex, hardware-supported cache flushing protocols. These questions are evaluated in Section 4.6.

3 Reducing Bit Flips in Software

By reducing bit flips in software, we can effect improvements in BNVM lifetime and power use without the need for hardware changes. To build data structures to reduce bit flips (Sections 3.1–3.3), we propose several optimizations to pointer storage along with additional optimizations for indicating occupancy. For stack writes, we propose changes to compilers to spill registers such that they avoid writing different registers to the same place in the stack (Section 3.4).

3.1 XOR Linked Lists

XOR linked lists [34] are a memory-efficient doubly-linked list design where, instead of storing a previous and next node pointer, each node stores only a `siblings` value that is the XOR between the previous and next node. If the previous node is at address p and the next node is at address n, the node stores $siblings = p \oplus n$. This scheme cuts the number of stored pointers per node in half while still allowing bi-directional traversal of the list—having pointers to two adjacent nodes is sufficient to traverse both directions. However, an XOR linked list has disadvantages; it does not allow $O(1)$ removal of a node with just a single pointer to that node, as a node's siblings cannot be determined from the node alone, and it increases code complexity by requiring XOR operations before pointers are dereferenced.

When they were proposed, XOR linked lists had little advantage over doubly linked lists beyond a modest memory saving. However, with the need for fewer bit flips on BNVM, they gain a critical advantage: they cut the number of stored pointers in half, reducing writes, but they also store the XOR of two pointers, which are likely to contain similar higher-order bits, making the `siblings` pointer mostly zeros.

One problem with the original design for XOR linked lists is that each node stores $siblings = p \oplus n$, but for the first and last node, p or n are `NULL`, so the full pointer value for its adjacent node is stored in the head and tail. To further cut down on bit flips, we changed this design so that the head and tail XOR their adjacent nodes with themselves (if the node at address h is the head, then it stores $siblings = h \oplus n$ instead of $siblings = 0 \oplus n$). The optimization here is *not* a performance optimization—in fact, it's likely to reduce performance—and only makes sense in the context of bit flips, an optimization goal that would not be targeted before the introduction of BNVM. However, with bit flips in-mind, it becomes critical. Other data structures may have similar optimizations that we can easily make to reduce bit flips [1].

[1] Circular linked lists solve the head and tail siblings pointer problem automatically, since no pointers are stored as `NULL`; however, in XOR linked lists this increases the number of pointer updates during an insert operation and requires storing two adjacent head nodes to traverse.

3.2 XOR Hash Tables

A direct application of XOR linked lists is chained hashing, a common technique for dealing with hash table collisions [12]. An array of linked list heads is maintained as the hash table, and when an item is inserted, it is appended to the list at the bucket that the item hashes to. To optimize for bit flips, we can store an XOR list instead of a normal linked list, but since bidirectional traversal is not needed in a hash table bucket, we need not complicate the implementation with a full XOR linked list. Instead, we apply the property of XOR linked lists that we find useful—XORing pointers.

Each pointer in each list node is XORed with the address of the node that contains that pointer. For example, a list node n whose next node is p will store $n \oplus p$ instead of p. In effect, this stores the distance between the nodes rather than the absolute address of the next node and exploits locality in memory allocators. The end of the list is marked with a `NULL` pointer. In addition to a distance pointer, each node contains a key and a pointer to a value. The list head stored in the hash table is a full node, allowing access to the first entry in the list without needing to follow a pointer.

A second optimization we make is that an empty list can be marked in one of two ways: the least-significant bit (LSB) of the `next` pointer set to one, or the data pointer set to `NULL`. When we initialize the table, it is set to zero everywhere, so the data pointers are `NULL`. During delete, if the list becomes empty, the LSB of the next pointer in the list head is set to 1, a value it would never have when part of a list. This allows the data pointer to remain set to a value such that when it is later overwritten, fewer bits need to change. This is an example of an optimization that only makes sense in the context of bit flips, as it increases code complexity for no other gain.

3.3 XOR Red-Black Trees

Binary search trees are commonly used for data indexing, support range queries, and allow efficient lookup and modification, as long as they are balanced. Red-black trees [12, 31] are a common balanced binary tree data structure with strictly-bounded rebalancing operations during modification. A typical red-black tree (RBT) node contains pointers to its left child and right child, along with meta-data. They often also contain a pointer to the parent node, since this enables easier balancing implementation and more efficient range-query support without significantly affecting performance due to the increased memory usage [23].

We can generalize XOR linked lists to *XOR trees*. Instead of storing `left`, `right`, and `parent` pointers, each node stores `xleft` and `xright`, which are the XOR between each child and the parent addresses. This reduces the memory usage to the two-pointer case while maintaining the benefits of having a parent pointer, since given a node and one of its children (or its parent), we can traverse the entire tree.

Like XOR linked lists, the root node stores `xleft = root ⊕ left`, where `root` is the address of the root node and `left` is the address of its left child, saving bit flips. To indicate that a node has no left or right child, it stores `NULL`.

Determining the child of a node requires both the node and its parent:

```
get_left_child(Node *node, Node *parent) {
  return (parent ⊕ node->xleft);
}
```

Getting a node's parent, however, requires additional work. Given a child c and a node n, getting n's parent requires we know *which* child (left or right) c is. Fortunately, in a binary search tree we store the key k of a node in each node, and the nodes are well-ordered by their k. Thus, getting the parent works as follows:

```
get_parent(Node *n, Node *c) {
  if(c->k < n->k) return (n->xleft ⊕ c);
  else return (n->xright ⊕ c);
}
```

Note that this is not the only way to disambiguate between pointers. In fact, it's not strictly necessary to do so because the algorithms can be implemented recursively without ever needing to traverse up the tree explicitly. However, providing upwards traversal can reduce the complexity of implementation and improve the performance of iteration over ranges. Another solution to getting the parent node would be to record whether a node is a left or right child by storing an extra bit along with the color. We did not evaluate this method, as it would increase both writes and bit flips over our method.

With these helper functions, we implemented both an XOR red black tree (`xrbt`) and a normal red-black tree (`rbt`) using similar algorithms. The code for `xrbt` was just 20 lines longer, with only a minor increase in code complexity. Node size was smaller in `xrbt`, with a node being 40 bytes instead of 48 bytes as in `rbt`. To control for the effects of node size on performance and bit flips, we built a variant of `xrbt` with the same code but with a node size of 48 bytes (`xrbt-big`).

Generalization These techniques can generalize beyond a red-black tree. Any ordered k-ary tree can use XOR pointers in the same way. As discussed above, disambiguating between pointers during traversal depends on either additional bits being stored or using an ordering property. Either technique can work with arbitrary graph nodes.

3.4 Stack Frames

Data structure layout and data writes are only some of the writes made by a program. Register spills, callee-saved register saving, and return addresses pushed during function calls are all writes to memory, and if these writes make it to BNVM, they will cause bit flips as well. These writes may make it to main memory if the cache is saturated or if the program is designed to keep program state in BNVM to enable instantaneous restart after power cycles [26]. Additionally, systems designed for BNVM may run with write-through caches to reduce consistency complexity, resulting in execution state reaching BNVM.

The exact pattern of stack writes depends on the ABI and the calling convention of a system and processor, though we focus on x86-64 Linux systems. When a program calls a function, it (potentially) pushes a number of arguments to the stack, followed by a return address. In the called function, callee-saved registers are pushed to the stack, but *only* if they are modified during that function's execution. When finished, the callee pops all the saved registers and returns.

Our observation is that the order that callee-saved registers are pushed to the stack is *not specified*, meaning that two different functions could push the same registers in a different order. Secondly, the same callee-saved register is less-likely to change drastically in a small amount of code in a tight loop, since these registers are typically used for loop counters or bases for addressing. Thus, a loop that calls two functions alternately will likely have similar or the same values in the callee-saved registers during the invocation of both functions. If these two functions push the (often unchanged) callee-saved registers to the same place both times, fewer bit flips will occur than if the functions pushed them in different orders. While this is just a simple example, such loops that call out to alternating functions with different characteristics can occur, for example, when rehashing a table, rebuilding a tree, or reading task items from a linked list.

We propose specifying a callee-saved register frame layout that functions adhere to, so that the registers are always pushed in the same order. To handle variable numbers of arguments, we make use of passing arguments in registers, common in many modern ABIs. If a function need not push any callee-saved registers, it can still reserve the stack space for that frame and then not push anything to save writes. Functions which only save a small number of registers can still push them to the correct locations within the frame. Finally, if this is standardized, programs need not worry about library calls increasing bit flips.

For example, if we have two functions `A` and `B` in an ABI where registers e, f, g, h are callee-saved, and `A` uses e while `B` uses g, then traditionally each function would simply push the frame pointer followed by the register they wish to save:

```
A:                      B:
push fp                 push fp
mov fp ← sp             mov fp ← sp
push e                  push g
...                     ...
pop e                   pop g
pop fp; ret             pop fp; ret
```

If e and g are significantly different, then a significant amount of needless bit flips could occur if these functions

are called often. Instead, if we define a layout that functions adhere to for register saving, the code would look like:

```
A:                      B:
push fp                 push fp
mov fp ← sp             mov fp ← sp
push e                  sub sp, 16
sub sp, 24              push g
...                     sub sp, 8
add sp, 24              ...
pop e                   add sp, 8
pop fp; ret             pop g
                        add sp, 16
                        pop fp; ret
```

Here the code always pushes the same register to the same place, regardless of the registers it needs to save, thereby allowing overwrites by likely similar values. While it does add some additional instructions, code could instead write registers directly to the stack locations using offset style addressing, reducing code size.

4 Memory Characteristics Results

We evaluated XOR linked lists, XOR hash tables, and XOR red-black trees, tracking bits flipped in memory, bytes written to memory, and bytes read from memory during program execution. Our goal was not only to demonstrate that our bit flip optimizations were effective, but to also understand how different system and program components affected bit flips. In addition to tracking bit flips caused by our data structures, we also studied bit flips caused by varying levels and sizes of caching, calls to `malloc`, and writes to the stack. Finally, we evaluated the accuracy of in-code instrumentation for bit flips, which would allow programmers to more easily optimize for bit flips at lower cost than full-system simulation. All of these experiments were designed to demonstrate how effective certain bit flipping reduction techniques are. Existing systems are poorly equipped to handle evaluation of these techniques, since existing systems are poorly optimized for BNVM. The techniques we present here are designed to be used by system designers when building *new*, BNVM-optimized systems.

4.1 Experimental Methods

Evaluating bit flips during data structure operations requires more than simply counting the bits flipped in each write in the code. Compiler optimizations, store-ordering, and the cache hierarchy can all conspire to change the order and frequency of writes to main memory, potentially causing a manual count of bit flips in the code to deviate from the bits flipped by writes that *actually* make it to memory. To record better metrics than in-code instrumentation, we ran

Table 1: Cache parameters used in Gem5.

Cache	Count	Size	Associativity
L1d	1	64KB	2-way
L1i	1	32KB	2-way
L2	1	2MB	8-way

our test programs on a modified version of Gem5 [3], a full-system simulator that accurately tracks writes through the cache hierarchy and memory. We modified the simulator's memory system so that, for each cache-line written, it could compute the Hamming distance between the existing data and the incoming write, thereby counting the bit flips caused by each write to memory. The bit flips for each write were added to a global count, which was reported after the program terminated, along with the number of bytes written to and read from memory. This gave us a more accurate picture of the bit flips caused by our programs, since writes that stay within the simulated cache hierarchy do not contribute to the global count. We ran the simulator in *system-call emulation* mode, which runs a cycle-accurate simulation, emulating system calls to provide a Linux-like environment, while tracking statistics about the program, including the memory events we recorded.

We used the default cache hierarchy (shown in Table 1) provided by Gem5, using the command-line options "`--caches --l2cache`". For the XOR linked list and stack writes experiments, we used `clwb` instructions to simulate consistency points (in the linked list, `clwb` was issued to persist the contents of a node before persisting the pointers to the node, and for stack writes, `clwb` was issued after each write). This was not done for the `malloc` experiment (we used an unmodified system `malloc` for testing), the XOR hash table (the randomness of access to the table quickly saturated the caches anyway), or manual instrumentation (caches were irrelevant). For the XOR red-black tree, in addition to the bit flip characteristics, we focused on observing how cache behavior affected more complex data structures; these results, along with the results of varying L2 size, are discussed in Section 4.6.

Most of the programs we ran accept as their first argument an `iteration_count`, which specifies how many iterations the program should run. For example, the red-black tree would do `iteration_count` number of insertions. We ran the simulator on a range of `iteration_counts`, recording the bits flipped, bytes written, and bytes read (collectively referred to as *memory events*) for each value of `iteration_count`. An example of a typical result is shown in Figure 2. The result was often linear, allowing us to calculate a linear regression using gnuplot, giving us both a slope and confidence intervals. The slope of the line is "bit flips per operation"—for example, a slope of 10 for linked list insert means that it flipped 10 bits on average during insert operations. Throughout our results, only the slope is presented unless the raw data is non-linear. Since the slope encodes

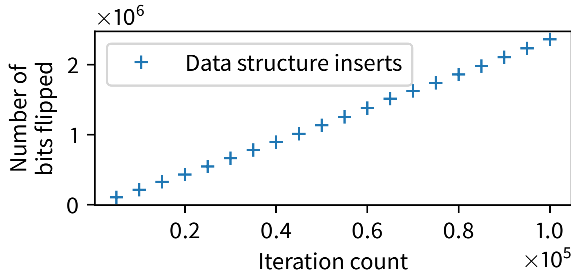


Figure 2: A typical result of running a test program with increasing values of `iteration_count`.

the bit flips per operation, we can directly compare variants of a data structure by comparing their slopes. Error bars are 95% confidence intervals.

4.2 Calls to `malloc`

Many data structures allocate data during their operation. For example, a binary tree may allocate space for a node during insert or a hash table might decide to resize its table. An allocator allocating data from BNVM must store the allocation metadata within BNVM as well, so the internal allocator structures affect bit flips for data structures which allocate memory. Additionally, the pointers returned *themselves* contribute to the bits flipped as they are written.

We called `malloc` 100,000 times with allocation sizes of 16, 24, 40, and 48 bytes. We chose these sizes because our data structure nodes were all one of these sizes. The number of bits flipped per `malloc` call is shown in Figure 3. As expected, larger allocation sizes flip more bits, since the allocator meta-data and the allocated regions span additional cache lines. Interestingly, 40 byte allocations and 48 byte allocations switch places partway through, with 40 byte allocations initially causing fewer bit flips and later causing more after a cross-over point. We believe this is due to 40 byte allocations using fewer cache lines, but 48 byte allocations having better alignment.

After a warm-up period where the cache hierarchy has a greater effect, the trends become linear, allowing us to calculate the bit flips per `malloc` call. Allocating 40 bytes costs 1.5× more bit flips on average than allocating 48 bytes. Allocating 24 or 16 bytes has the same flips per `malloc` as 48 bytes but has a longer warm-up period, such that programs would need to call `malloc` (24) 1.56× as often to flip the same number of bits as `malloc` (48).

While the relative savings for bit flips between `malloc` sizes are significant, their absolute values must be taken into consideration. Calls to `malloc` for 16 and 48 bytes cost 2 ± 0.1 flips per `malloc` (after the warm-up period) while calls to `malloc` for 40 bytes cost 3 ± 0.1 flips per `malloc`. As we will see shortly, the data structures we are evaluating flip tens of bits per operation, indicating that savings from `malloc` sizes are less significant than the specific optimizations they employ.

4.3 XOR Linked Lists

We evaluated the bit flip characteristics of an XOR linked list compared to a doubly-linked list, where we randomly inserted (at the head) and popped nodes from the tail at a ratio of 5:1 inserts to pops. The results are shown in Figure 4. As expected, bit flips are significantly reduced when using XOR linked lists, by a factor of 3.56×. However, both the number of bytes written to and read from memory were the same between both lists. The reason is that, although an XOR list node is smaller, `malloc` actually allocates the same amount of memory for both.

We counted the number of pointer read and write operations in the code, and discovered that, although the XOR linked list performs fewer write operations during updates, it performs *more* read operations than the doubly-linked list. This is because updating the data structure requires more information than in a doubly-linked list. However, Figure 4 shows that the number of reads *from memory* are the same, indicating that the additional reads are always in-cache.

4.4 XOR Hash Tables

We implemented two variants of our hash table: "single-linked", which implemented chaining using a standard linked list, and "XOR Node", which XORs each pointer in the chain with the address of the node containing the pointer. We ran a Zipfian workload on them [5], where 80% of updates happen to 20% of keys[2], where keys and values were themselves Zipfian. During each iteration, if a key was present, it was deleted, while if it was not present, it was

[2]Skew of 1, with a population of 100,000.

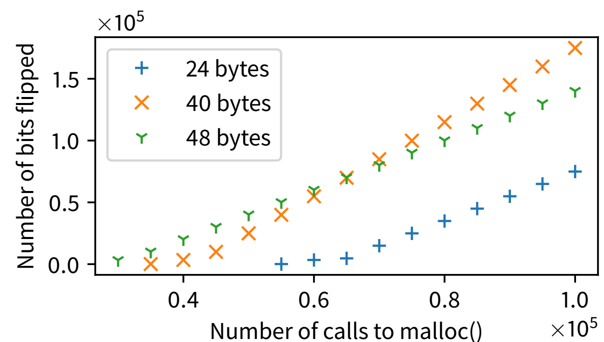


Figure 3: Bit flips due to calls to `malloc`. Allocation size of 16 bytes is not shown because it matches with 24 bytes.

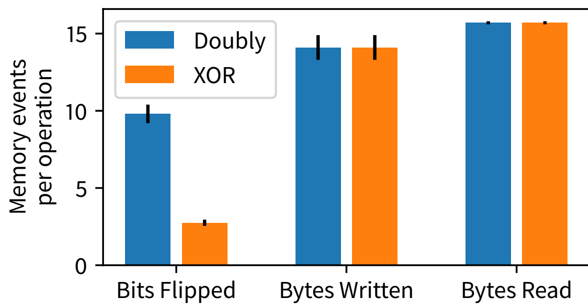


Figure 4: Memory characteristics of of XOR linked lists compared to Doubly-Linked Lists.

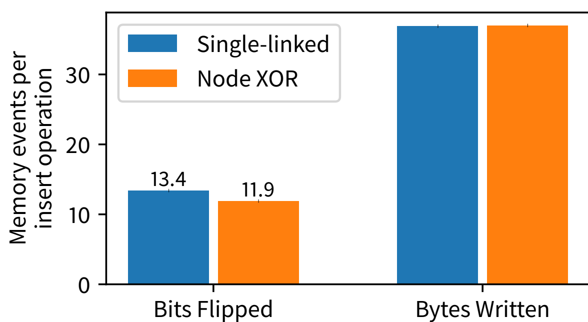


Figure 5: Memory characteristics of XOR hash table variants under Zipfian workload.

inserted. This resulted in a workload where a large number of keys were rarely modified, but a smaller percentage were repeatedly inserted or removed from the hash table.

Figure 5 shows the bits flipped and bytes written by the hash table after 100,000 updates. As expected, the XOR lists saw a reduction in bit flips over the standard, singly-linked list implementation while the number of bytes written were unchanged. We were initially surprised by the relatively low reduction in bit flips ($1.13\times$) considering the relative success of XOR linked lists; however, the common case for hash tables is short chains. We observed that longer chains improve the bit flips savings, but forcing long hash chains is an unrealistic evaluation. Since buckets typically have one element in them, and that element is stored in the table itself, there are few pointers to XOR, meaning the reduction is primarily from indicating a list is valid via the least-significant bit of the next pointer. The bit flips in all variants come primarily from writing the key and value, which comprise 9.3 bit flips per iteration on average. Thus, this data structure had little room for optimization, and the improvements we made were relatively minor—although they still translate directly to power saving and less wear, and are easy to achieve while not affecting code complexity significantly.

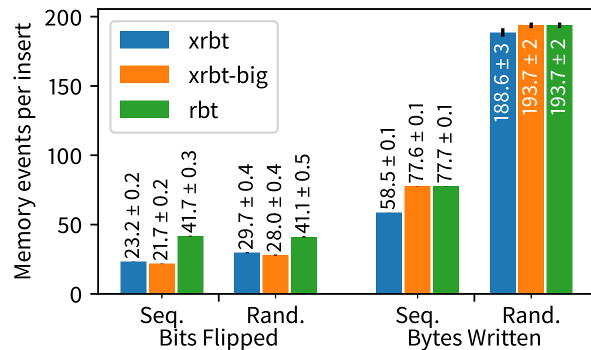


Figure 6: Memory characteristics of XOR red-black trees compared to normal red-black trees.

4.5 XOR Red-Black Trees

Figure 6 shows the memory event characteristics of `xrbt` (our XOR RBT with two pointers, `xleft` and `xright`), `xrbt-big` (our XOR RBT with each node inflated to the size of our normal RBT nodes), and `rbt` (our standard RBT) under sequential and random inserts of one million unique items. Each item comprises an integer key from 0 to one million and a random value. Both `xrbt` and `xrbt-big` cut bit flips by $1.92\times$ (nearly in half) in the case of sequential inserts and by $1.47\times$ in the case of random inserts, a dramatic improvement for a simple implementation change. The small saving in bit flips in `xrbt-big` over `xrbt` is likely due to the allocation size difference as discussed in Section 4.2.

The number of bytes written is also shown in Figure 6. Due to the cache absorbing writes, `xrbt-big` and `rbt` write the same number of bytes to memory in all cases, even though `rbt` writes more pointers during its operation. We can also see a case where the number of writes was not correlated with the number of bits flipped, since `xrbt` writes fewer bytes but flips more bits than `xrbt-big`.

We did not implement and test delete operation in our red-black trees because the algorithm is similar to insert in that its balancing algorithm is tail-recursive and merely recolors or rotates the tree a bounded number of times. Since the necessary functions to implement this algorithm are present in all variations, it is certainly possible to implement, and we expect the results to be similar between them.

4.6 Cache Effects

Although it is easy to exceed the size of the L1 cache during normal operation of large data structures at scale, larger caches may have more of an effect on the frequency of writes to memory. Of course, a persistent data structure which issues cache-line writebacks or uses write-through caching by-

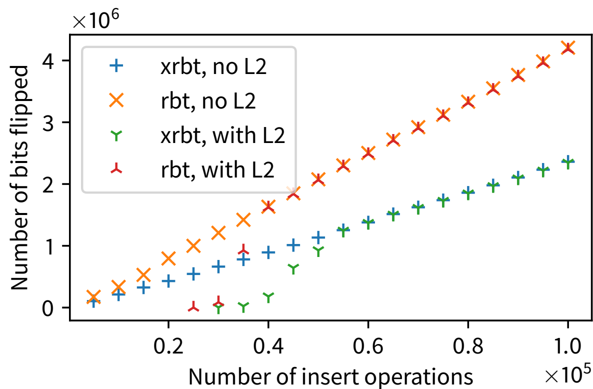


Figure 7: Bits flipped by xrbt and rbt over a varying number of sequential inserts, with and without the L2 cache present.

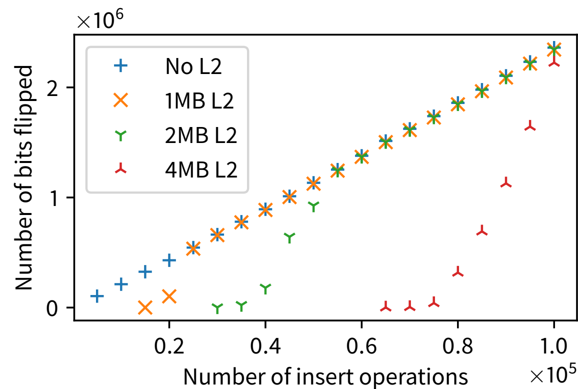


Figure 8: Bits flipped by xrbt over a varying number of sequential inserts, with different sizes for L2.

passes this by causing *all* writes to go to memory[3], but it is still worth studying the effects of larger write-back caches on bit flips. They may absorb specific writes that have higher than average flips, or they may cause coalescing even for persistent data structures worrying about consistency.

We studied cache effects in two ways—how the mere presence of a layer-2 cache affects the data structures we studied and how varying the size of that cache affects them. Figure 7 shows `xrbt` compared with `rbt`, with and without L2. The effect of L2 is limited as the operations scale, with the bit flips for both data structures reaching a steady, linear increase once L2 is saturated. The bit flips per operation for both data structures with L2 is the same as without L2 once the saturation point is reached, indicating that while the presence of the cache *delays* bit flips from reaching memory, it does little to reduce them in the long term. Finally, since `xrbt` has fewer bit flips overall and fewer memory writes, it took longer to saturate L2, delaying the effect.

Next, we looked at different L2 sizes, running `xrbt` with no L2, 1MB L2, 2MB L2 (the default), and 4MB L2, as shown in Figure 8. The exact same pattern emerges for each size, delayed by an amount proportional to the cache size. This is to be expected, and it further corroborates our claim that cache size has only short-term effects.

4.7 Manual Instrumentation

While testing data structures on Gem5 was straightforward, if time consuming, more complex structures and programs may be difficult to evaluate, either due to Gem5's relatively limited system call support or due to the extreme slowdown caused by the simulation. Since real hardware does not provide bit flip counting methods, we are left with using in-

[3]Even if this is the case, a full system simulator will give a more accurate picture than manually counting writes, since store ordering and compiler optimizations still affect memory behavior.

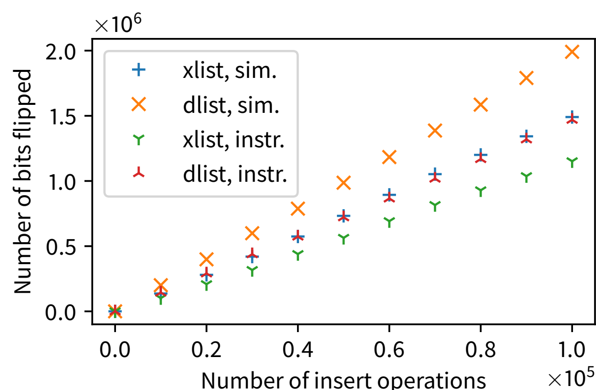


Figure 9: Manual instrumentation for counting bit flips (instr) compared to full-system simulation (sim).

program instrumentation if we want to avoid the Gem5 overhead. However, these results may be less accurate.

To study the accuracy of in-code instrumentation, we manually counted bit flips in the XOR and doubly-linked lists. We did this by replacing all direct data structure writes (*e.g.*, `node->prev = pnode`) with a macro that both did that write and also counted the number of bytes (by looking at the types), and computing the Hamming distance between the original and new values. Totals of each were kept track of and reported at the end of program execution. While not difficult to implement, manual instrumentation adds the possibility of error and increases implementation complexity.

Figure 9 shows the results of manual instrumentation compared to results from Gem5. While accuracy suffered, manual counting was not off by orders of magnitude. It properly represented the relationship between XOR linked lists and doubly-linked lists in terms of bit flips, and it was off by a constant factor across the test. We hypothesize that the dis-

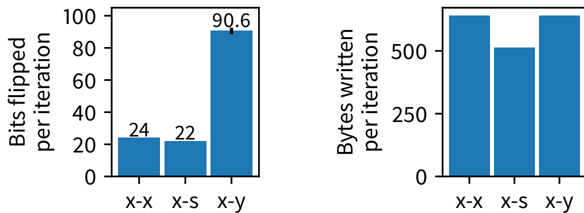


Figure 10: Evaluating memory events for different stack frame layouts.

crepancy arose from the fact that our additional flip counting code affected the write combining and (possibly) the cache utilization. We expect that future system designs could "calibrate" manual instrumentation by running a smaller version of their system on Gem5 to calculate the discrepancy between its counts and theirs, allowing them to more accurately extrapolate the bits flipped in their system using instrumentation. Additionally, one could modify toolchains and debugging tools to automatically emit such instrumentation code during code generation. Manual instrumentation may find its use here for large systems that are too complex or unwieldy to run on Gem5, or as a way to quickly prototype bit flipping optimizations.

4.8 Stack Frames

To study bit flips caused by stack writes, we wrote an assembly program that alternates between two function calls in a tight loop while incrementing several callee-saved registers on x86-64. The loop could call two of three functions—function *x*, which pushed six registers (the callee-save registers on x86_64, including the base pointer) in a given order, *y*, which pushed the registers in a different, given order, and *s*, which pushed only two of the registers, but pushed them to the same locations as function *x*. Our program had three variations: **x-x**, which called function *x* twice, **x-s**, which alternated between functions *x* and *s*, and **x-y**, which alternated between functions *x* and *y*. The x-y variant represents the worst-case scenario of today's methods for register spilling, while x-s demonstrates our suggestion for reducing bit flips. To force the writes to memory, we used `clwb` after the writes to simulate write-through caching or resumable programs.

Figure 10 shows both bit flips (left) and bytes written (right) by all three variants. The x-s and x-x variants have similar behavior in terms of bit flips, which is understandable because they are pushing registers to the same locations within the frame. The x-y variant, however, had 3.8× the number of bit flips compared to x-x and 4.1× the number of bit flips compared to x-s, showing that consistency of frame layout has dramatic impact. Unsurprisingly, x-x and x-y had the same number of bytes written, since they write the same number of registers, while x-s wrote fewer registers. By keeping frame layout consistent, we can reduce bit flips, and the optimization of only pushing the registers needed but to the same locations can further reduce writes as well.

Table 2: Performance of XOR linked lists compared with doubly-linked lists.

Operation	XOR Linked	Doubly-Linked
Insert (ns)	45 ± 1	45 ± 1
Pop (ns)	27 ± 1	28 ± 1
Traverse (ns/node)	2.6 ± 0.1	2.2 ± 0.1

5 Performance Analysis

While bit flip optimization is important, it is less attractive if it produces a large performance cost. We compared our data structures' performance to equivalent "normal" versions not designed to reduce bit flips. Benchmarks were run on an i7-6700K Intel processor at 4GHz, running Linux 4.18, `glibc` 2.28. They were compiled using `gcc` 8.2.1 and linked with GNU `ld` 2.31.1. Unless otherwise stated, programs were linked dynamically and compiled with `O3` optimizations.

XOR Linked Lists The original publication of XOR linked lists found little performance difference between them and normal linked lists [34]; we see the same relationship in our implementation (see Table 2). The only statistically significant difference was seen in traversal, where XOR linked lists have a 1.18× increase in latency; however, both lists average less than three nanosecond-per-node during traversal.

XOR Hash Tables Figure 11 shows the performance of the two hash table variants we developed. We inserted 100,000 keys, followed by lookup and delete. As expected, both variants have nearly identical latencies, with a slowdown of only 1.06× for using XOR lists during lookup.

XOR Red-Black Trees We measured `xrbt`, `xrbt-big`, and `rbt` during 100,000 inserts and lookups, the results of

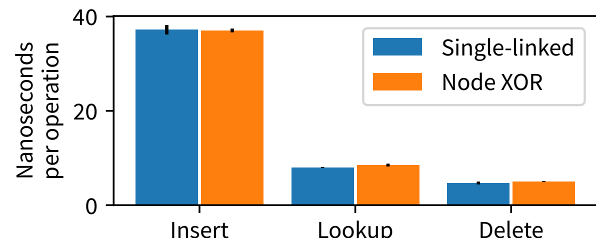


Figure 11: Performance of XOR hash table variants.

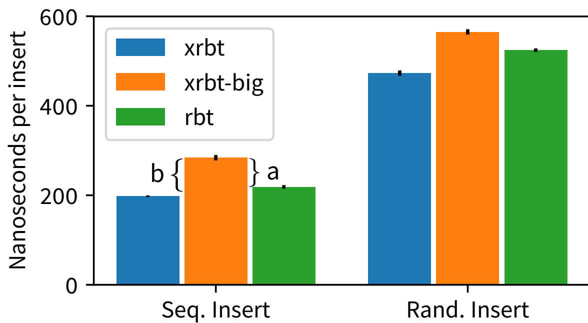


Figure 12: Insert latency for XOR red-black trees compared to normal red-black trees. The label "a" shows the cost of the XORs, while "b" shows the cost of the larger node.

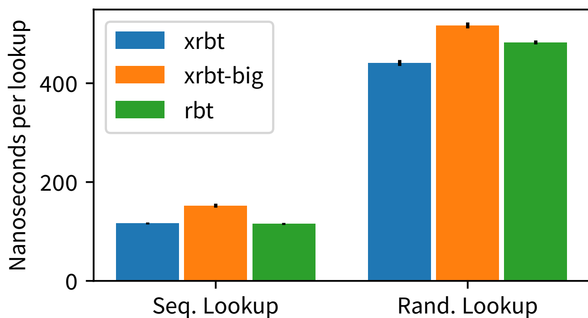


Figure 13: Lookup latency for XOR red-black trees compared to normal red-black trees.

which are shown in Figure 12 and Figure 13. During insert, `xrbt` is actually slightly faster than `rbt`, with `xrbt-big` being slower, indicating that although there is a non-zero cost for the additional XOR operations, it is outweighed by the performance improvement from smaller node size and better cache utilization. The lookup performance shown in Figure 13 demonstrates a similar pattern, although for sequential lookup the overheads are similar enough that there is no significant performance difference between `xrbt` and `rbt`.

6 Discussion

Software Bit Flip Reduction The data structures presented here are both old and new ideas. While not algorithmically different from existing implementations (both `xrbt` and `rbt` use the same, standard red-black tree algorithms), they present a new approach to implementation with optimizations for bit flipping. This has not been sufficiently studied before in the context of software optimization; after all, there is no theoretical advance nor is there an overwhelming practical advantage to these data structures outside of the bit flip reduction, an optimization goal that is new with BNVM. However, keeping this in mind has huge ramifications for data structures in persistent memory and applications for new storage technologies, as it presents a whole new field of study in optimization and practical data structure design. The goal is not performance improvements; instead we strive to prolong the lifetime of expensive memory devices while reducing power use, with at most a minor performance cost. These improvements can be achieved without hardware changes, meaning even savings of 10% ($1.1\times$) or less are worthwhile to implement because savings are cumulative.

These optimizations are not specific to PCM; any memory with a significant read/write disparity and bit-level updates could benefit from this. The energy savings from bit flip optimization will, of course, be technology-dependent, numbers for which will solidify as the technologies are adopted. Our estimates of the linear relationship between flips and power use (Figure 1) indicate that, on PCM, the energy savings will be roughly proportional to the bit flip savings since the difference between read and write energy is so high.

Bit flips can and should be reasoned about directly. Not only is it possible to do so, but the methods presented here are straightforward once this goal is in mind. Furthermore, while reducing writes *can* reduce bit flips, we have confirmed that this is not *always* true. XOR linked lists reduced bit flips without affecting writes, while `xrbt` reduced writes over `xrbt-big` at the cost of increasing bit flips. With stack frames, the biggest reduction in bit flips corresponded with no change in writes, while the reduction of writes was correlated with only a modest bit flip reduction.

The implications are far-reaching, especially when considering novel computation models that include storing program state in BNVM. Writes to the stack also affect bit flips, but these can be dramatically optimized. Compilers can implement standardized stack frame layouts for register spills that save many bit flips while remaining backwards compatible since nothing in these optimizations breaks existing ABIs. Further research is required to better study the effects of stack frame layout in larger programs, and engineering work is needed to build these features into existing toolchains.

Of course, we must be cautious to optimize where it matters. While different allocation sizes reduced bit flips relative to each other, the overall effect was minor compared to the savings gained in other data structures. In fact, the reduction in allocation size from 48 to 40 bytes in `xrbt` actually *increased* bit flips in calls to `malloc`, but this increase is dwarfed by the savings from the XOR pointers. Additionally, the hash table saw a relatively small saving compared to other data structures since it already flipped a minimal number of bits in the average case; red-black trees often do more work during *each* update operation, resulting in a number of pointer updates. Hash tables often do their "rebalancing" during a single rehash operation; perhaps bit flip optimization for hash tables should focus on these operations, something we plan to investigate in the future.

Cache Effects The data structures we tested all had the same behavior—a warm-up period where the cache system absorbed many of the writes followed by a period of proportional increasing of bit flips as the number of update operations increased. We must keep this in-mind when evaluating data structures for bit flips, since we must ensure that the ranges of inputs we test reach the expected scale for our data structures, or we may be blind to its true behavior. The cache size affects this, of course, but it does so in a predictable way in the case of `xrbt`, with only the warm-up period being extended by an amount proportional to cache size. Of course, the behaviour may be heavily dependent on write patterns. Thus, we recommend further experiments and that system designers take caches into account when evaluating bit flip behaviour of their systems.

The cache additionally affects the read amplification seen in XOR linked lists, wherein the XOR linked list implementation issues more reads than a doubly-linked list implementation. However, the reads that make it to memory are the same between the two, indicating that those extra reads are always in-cache. The resultant write reduction and bit flip reduction is well worth the cost since a read from cache is significantly cheaper than a write to memory.

7 Future Work

Although we covered a range of different data structures, there are many more used in storage systems that should be examined, such as B-trees [1] and LSM-trees [27], both to understand their bit flipping behavior as compared to other data structures and to examine for potential optimizations. In addition to data structures, different algorithms such as sorting can be evaluated for bit flips. Though this may come down to data movement minimization, there may be optimizations in locality that could affect bit flips.

While data structure and algorithm evaluation can provide system designers with insights for how to reduce bit flips, examining bit flips in a large system, including one that properly implements consistency and our suggested stack frame modifications (perhaps through compiler modification), would be worthwhile. There are a number of BNVM-based key-value stores [37]; comparing them for bit flips could demonstrate the benefits of some designs over others.

Studying bit flips directly is a good metric for understanding power consumption and wear, but a better understanding through the evaluation of real BNVM would be illuminating. The power study discussed earlier was derived from a number of research papers that give approximate numbers or estimates. On a real system, we could *measure* power consumption, and cooperation with vendors may enable accurate studies of wear caused by bit flips. Additionally, some technologies (*e.g.*, PCM) have a disparity between writing a 1 or a 0, something that could be leveraged by software (in cooperation with hardware) to further optimize power use.

8 Conclusion

The pressures from new storage hardware trends compel us to explore new optimization goals as BNVM becomes more common as a persistent store; the read/write asymmetry of BNVM must be addressed by reducing bit flips. As we showed, the number of raw writes is not always a faithful proxy for the number of bit flips, so simple techniques that minimize writes overall may be ineffective. At the OS level, we can reconsider memory allocator design to minimize bit flips as pointers are written. Different data structures use and write pointers in different ways, leading to different tradeoffs for data structures when considering BNVM applications. At the compiler level, we show that careful layout of stack frames can have a significant impact on bit flips during program operation. Since it can be challenging to reason directly about how application-level writes translate to raw writes due to the compiler and caches, more sophisticated profiling tools are needed to help navigate the tradeoffs between performance, consistency, power use, and wear-out.

Most importantly, we demonstrated the value of reasoning at the *application level* about bit flips, reducing bit flips by $1.13-3.56\times$ with minor code changes, no significant increase in complexity, and little performance loss. The data structures we studied had novel implementations, but were *algorithmically* the same as their standard implementations; yet we still saw dramatic improvements with little effort. This indicates that reasoning about bit flips in software can yield significant improvements over in-hardware solutions and opens the door for additional research at a variety of levels of the stack for bit flip reduction. These techniques translate directly to power saving and lifetime improvements, both important optimizations for early adoption of new storage trends that will have lasting impact on systems, applications, and hardware.

Availability

Source code, scripts, Gem5 bit flip patch, and raw results are available at `https://gitlab.soe.ucsc.edu/gitlab/crss/opensource-bitflipping-fast19`.

Acknowledgments

This research was supported in part by the National Science Foundation grant number IIP-1266400 and by the industrial partners of the Center for Research in Storage Systems. The authors additionally thank the members of the Storage Systems Research Center for their support and feedback. We would like to extend our gratitude to our paper shepherd, Sam H. Noh, and the anonymous reviewers for their feedback and assistance.

References

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[2] F. Bedeschi, C. Resta, O. Khouri, E. Buda, L. Costa, M. Ferraro, F. Pellizzer, F. Ottogalli, A. Pirovano, and M. Tosi. An 8MB demonstrator for high-density 1.8 V phase-change memories. In *Symposium on VLSI Circuits 2004 Digest of Technical Papers*, pages 442–445. IEEE, 2004.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, Aug. 2011.

[4] D. Bittman, M. Gray, J. Raizes, S. Mukhopadhyay, M. Bryson, P. Alvaro, D. D. E. Long, and E. L. Miller. Designing data structures to minimize bit flips on NVM. In *Proceedings of the 7th IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA 2018)*, Aug. 2018.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *In Proceedings of Conference on Computer Communications, IEEE INFOCOM '99*, volume 1, pages 126–134, March 1999.

[6] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4/5):449–464, July 2008.

[7] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, pages 21–31, January 2011.

[8] S. Cho and H. Lee. Flip-N-Write: a simple deterministic technique to improve PRAM write performance, energy and endurance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 347–357. ACM, 2009.

[9] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 105–118, Mar. 2011.

[10] J. Colgrove, J. D. Davis, J. Hayes, E. L. Miller, C. Sandvig, R. Sears, A. Tamches, N. Vachharajani, and F. Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of SIGMOD 2015*, June 2015.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 133–146, Oct. 2009.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[13] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A hybrid PRAM and DRAM main memory system. In *Proceedings of the 46th IEEE Design Automation Conference (DAC '09)*, pages 664–669. IEEE, 2009.

[14] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi. NVSim: a circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7), July 2012.

[15] G. Fox, F. Chu, and T. Davenport. Current and future ferroelectric nonvolatile memory technology. *Journal of Vacuum Science & Technology B: Microelectronics and Nanometer Structures Processing, Measurement, and Phenomena*, 19(5):1967–1971, 2001.

[16] K. M. Greenan and E. L. Miller. PRIMS: Making NVRAM suitable for extremely reliable storage. In *Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep '07)*, June 2007.

[17] M. Han, Y. Han, S. W. Kim, H. Lee, and I. Park. Content-aware bit shuffling for maximizing PCM endurance. *ACM Transactions on Design Automation of Electronic Systems*, 22(3):48:1–48:26, May 2017.

[18] Intel Newsroom. Intel and Micron produce breakthrough memory technology, 2015. `http://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/`; Accessed 2019-01-10.

[19] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin. Coset coding to extend the lifetime of memory. In *Proceedings of High Performance Computer Architecture (HPCA '13)*, pages 222–233. IEEE, 2013.

[20] H. Jayakumar, K. Lee, W. S. Lee, A. Raha, Y. Kim, and V. Raghunathan. Powering the internet of things. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '14)*, pages 375–380, New York, NY, USA, 2014. ACM.

[21] H. Jayakumar, A. Raha, and V. Raghunathan. Quickrecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *Proceedings of the 27th International Conference on VLSI Design and 13th International Conference on Embedded Systems*, pages 330–335. IEEE, 2014.

[22] T. Kawahara. Scalable spin-transfer torque RAM technology for normally-off computing. *IEEE Design and Test of Computers*, 28(1):52–63, Jan 2011.

[23] E. Kohler. Left-leaning red-black trees considered harmful. `http://read.seas.harvard.edu/~kohler/notes/llrb.html`. Accessed 2018-09-22.

[24] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 2–13. ACM, 2009.

[25] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu. A case for efficient hardware/software cooperative management of storage and memory. In *5th Workshop on Energy-Efficient Design (WEED '13)*, June 2013.

[26] D. Narayanan and O. Hodson. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, pages 401–500, Mar. 2012.

[27] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[28] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[29] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture (ICSA '09)*, pages 24–33, 2009.

[30] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4/5):465–480, July 2008.

[31] R. Sedgewick and L. J. Guibas. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS '78)*, volume 00, pages 8–21, 10 1978.

[32] S. M. Seyedzadeh, R. Maddah, D. Kline, A. K. Jones, and R. Melhem. Improving bit flip reduction for biased and random data. *IEEE Transactions on Computers*, 65(11):3345–3356, 2016.

[33] S.-S. Sheu et al. Fast-write resistive RAM (RRAM) for embedded applications. *IEEE Design & Test of Computers*, pages 64–71, Jan. 2011.

[34] P. Sinha. A memory-efficient doubly linked list. *Linux Journal*, 129, 2004. `http://www.linuxjournal.com/article/6828`.

[35] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453:80–83, May 2008.

[36] H. Volos, A. Jaan Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, Mar. 2011.

[37] F. Xia, D. Jiang, J. Xiong, and N. Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, 2017. USENIX Association.

[38] J. Xu and S. Swanson. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Feb. 2016.

[39] B. D. Yang, J. E. Lee, J. S. Kim, J. Cho, S. Y. Lee, and B. G. Yu. A low power phase-change random access memory using a data-comparison write scheme. In *Proceedings of IEEE International Symposium on Circuits and Systems*, May 2007.

[40] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 14–23, 2009.

Write-Optimized Dynamic Hashing for Persistent Memory

Moohyeon Nam[†], Hokeun Cha[‡], Young-ri Choi[†], Sam H. Noh[†], Beomseok Nam[‡]
UNIST (Ulsan National Institute of Science and Technology)[†]
Sungkyunkwan University[‡]

Abstract

Low latency storage media such as byte-addressable persistent memory (PM) requires rethinking of various data structures in terms of optimization. One of the main challenges in implementing hash-based indexing structures on PM is how to achieve efficiency by making effective use of cachelines while guaranteeing failure-atomicity for *dynamic hash expansion and shrinkage*. In this paper, we present *Cacheline-Conscious Extendible Hashing* (CCEH) that reduces the overhead of dynamic memory block management while guaranteeing constant hash table lookup time. CCEH guarantees failure-atomicity without making use of explicit logging. Our experiments show that CCEH effectively adapts its size as the demand increases under the fine-grained failure-atomicity constraint and its maximum query latency is an order of magnitude lower compared to the state-of-the-art hashing techniques.

1 Introduction

In the past few years, there have been numerous efforts to leverage the byte-addressability, durability, and high performance of persistent memory (PM) [7, 13, 18, 32, 34, 39, 40, 45, 47]. In particular, latency critical transactions on storage systems can benefit from storing a small number of bytes to persistent memory. The fine-grained unit of data I/O in persistent memory has generated interest in redesigning block-based data structures such as B+-trees [2, 11, 20, 28, 46]. Although a large number of previous studies have improved tree-based indexing structures for byte-addressable persistent memory, only a few have attempted to adapt hash-based indexing structures to persistent memory [48, 49]. One of the main challenges in hash-based indexing for PM is in achieving efficient *dynamic rehashing* under the fine-grained failure-atomicity constraint. In this paper, we present Cacheline-Conscious Extendible Hashing (CCEH), which is a variant of extendible hashing [6] optimized for PM to minimize cacheline accesses and satisfy failure-atomicity without explicit logging.

Due to the static flat structure of hash-based indexes, they can achieve constant lookup time. However, static hashing does not come without limitations. Such traditional hashing schemes must typically estimate the size of hash tables and allocate sufficient buckets in advance. For certain applications, this is a feasible task. For example, in-memory hash tables in key-value stores play a role of fixed-sized buffer cache, i.e., recent key-value records replace old records. Hence, we can set the hash table size a priori based on the available memory space.

However, not all applications can estimate the hash table size in advance, with database systems and file systems being typical examples. If data elements are dynamically inserted and deleted, static fixed-sized hashing schemes suffer from hash collisions, overflows, or under-utilization. To resolve these problems, dynamic resizing must be employed to adjust the hash table size proportional to the number of records. In a typical situation where the load factor (bucket utilization) becomes high, a larger hash table must be created, and a *rehash* that moves existing records to new bucket locations must be performed.

Unfortunately, rehashing is not desirable as it degrades system throughput as the index is prevented from being accessed during rehashing, which significantly increases the tail latency of queries. To mitigate the rehashing overhead, various optimization techniques, such as *linear probing*, *separate chaining*, and *cuckoo hashing*, have been developed to handle hash collisions [4, 14, 25, 29, 31]. However, these optimizations do not address the root cause of hash collisions but defer the rehashing problem. As such, static hashing schemes have no choice but to perform expensive full-table (or 1/3-table [49]) rehash operations later if the allocated hash table size is not sufficient.

In light of PM, rehashing requires a large number of writes to persistent memory. As writes are expected to induce higher latency and higher energy consumption in PM, this further aggravates performance. Furthermore, with lifetime of PM expected to be shorter than DRAM, such extra writes can be detrimental to systems employing PM.

Unlike these static hashing schemes, *extendible hashing* [6] dynamically allocates and deallocates memory space on demand as in tree-structured indexes. In file systems, extendible hash tables and tree-structured indexes such as B-trees are used because of their dynamic expansion and shrinkage capabilities. For example, extendible hashing is used in Oracle ZFS [26], IBM GPFS [30, 33], Redhat GFS, and GFS2 file systems [38, 44], while tree structured indexes are used for SGI XFS, ReiserFS, and Linux EXT file systems. However, it is noteworthy that static hashing schemes are not as popular as dynamic indexes because they fall short of the dynamic requirements of file systems.

In this work, we show the effectiveness of extendible hashing in the context of PM. Byte-addressable PM places new challenges on dynamic data structures because the issue of failure-atomicity and recovery must be considered with care so that when recovered from failure, the data structure returns to a consistent state. Unfortunately, extendible hashing cannot be used as-is, but requires a couple of sophisticated changes to accommodate failure-atomicity of dynamic memory allocations on PM. As in other dynamic indexes, extendible hashing manages discontiguous memory spaces for hash buckets and the addresses of buckets are stored in a separate directory structure. When a bucket overflows or is underutilized, extendible hashing performs split or merge operations as in a tree-structured index, which must be performed in a failure-atomic way to guarantee consistency.

Cacheline-Conscious Extendible Hashing (CCEH) is a variant of extendible hashing with engineering decisions for low latency byte-addressable storage such as PM. For low latency PM, making effective use of cachelines becomes very important [11, 16, 35, 43]. Therefore, CCEH sets the size of buckets to a cacheline in order to minimize the number of cacheline accesses. Although CCEH manages a fine-grained bucket size, CCEH reduces the overhead of directory management by grouping a large number of buckets into an intermediate-sized *segment*. That is, CCEH works in three levels, namely, the global directory, segments pointed by the directory, and cache-line sized buckets in the segment. We also present how CCEH guarantees the failure-atomicity and recoverability of extendible hash tables by carefully enforcing the ordering of store instructions.

The main contributions of this work are as follows:

- First, we propose to use cacheline-sized buckets but reduce the size of the directory by introducing intermediate level segments to extendible hashing. The three-level structure of our cacheline-conscious extendible hashing (CCEH) guarantees that a record can be found within two cacheline accesses.
- Second, we present a failure-atomic rehashing (split and merge) algorithm for CCEH and a recovery algorithm based on MSB (most significant bit) keys that does not use explicit logging. We also show that MSB rather than LSB (least significant bit) is a more effective key for extendible hashing on PM, which is contrary to popular belief.
- Third, our extensive performance study shows that CCEH effectively adapts its size as needed while guaranteeing failure-atomicity and that the tail latency of CCEH is up to 3.4× and 8× shorter than that of the state-of-the-art Level Hashing [49] and Path Hashing [48], respectively.

The rest of this paper is organized as follows. In Section 2, we present the background and the challenges of extendible hashing on PM. In Section 3, we present Cacheline-Conscious Extendible Hashing and show how it provides failure-atomicity while reducing the amount of writes to PM. In Section 4, we present the recovery algorithm of CCEH. In Section 5, we discuss concurrency and consistency issues of CCEH. In Section 6, we evaluate the performance of PM-based hash tables. Finally, we conclude the paper in Section 7.

2 Background and Related Work

The focus of this paper is on dynamic hashing, that is, hashing that allows the structure to grow and shrink according to need. While various methods have been proposed [17, 19, 22], our discussion concentrates on extendible hashing as this has been adopted in numerous real systems [26, 30, 33, 38, 44] and as our study extends it for PM.

Extendible Hashing: Extendible hashing was developed for time-sensitive applications that need to be less affected by full-table rehashing [6]. In extendible hashing, re-hashing is an incremental operation, i.e., rehashing takes place per bucket as hash collisions make a bucket overflow. Since extendible hashing allocates a bucket as needed, pointers to dynamically allocated buckets need to be managed in a hierarchical manner as in B-trees in such a way that the split history can be kept track of. This is necessary in order to identify the correct bucket for a given hash key.

Figure 1 shows the legacy design of extendible hashing. In extendible hashing, a hash bucket is pointed to by an entry of a *directory*. The directory, which is simply a *bucket address table*, is indexed by either the leading (most significant) or the trailing (least significant) bits of the key. In the example shown in Figure 1, we assume the trailing bits are used as in common practice and each bucket can store a maximum of five key-value records. The *global depth* G stores the number of bits used to determine a directory entry. Hence, it determines the *maximum* number of buckets, that is, there are 2^G directory entries. When more hash buckets are needed, extendible hashing doubles the size of the directory by incrementing G. From the example, G is 2, so we use the low end 2 bits of the key to designate the directory entry in the directory of size 4 (2^2). Eventually, when the buckets fill up and split, needing more directory entries, G can be incremented to 3, resulting in a directory of size 8.

While every directory entry points to a bucket, a single bucket may be pointed to by multiple directory entries. Thus,

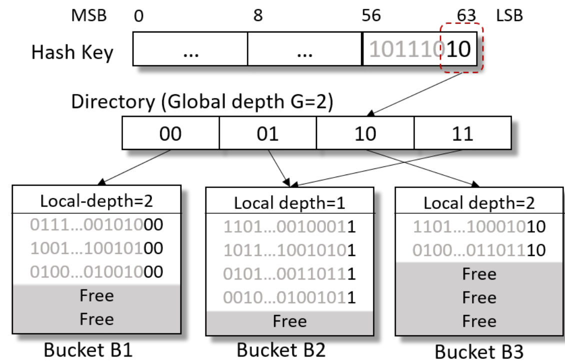


Figure 1: *Extendible Hash Table Structure*

each bucket is associated with a *local depth* (L), which indicates the length of the common hash key in the bucket. If a hash bucket is pointed by k directory entries, the local depth of the bucket is $L = G - log_2 k$. For example in Figure 1, B2 is pointed to by 2 directory entries. For this bucket, as the global depth (G) is 2 and the bucket is pointed to by two directory entries, the local depth of the bucket (L) is 1.

When a hash bucket overflows, extendible hashing compares its local depth against the global depth. If the local depth is smaller, this means that there are multiple directory entries pointing to the bucket, as for bucket B2 in Figure 1. Thus, if B2 overflows, it can be split without increasing the size of the directory by dividing the directory entries to point to two split buckets. Thus, G will remain the same, but the Ls for the two resulting buckets will both be incremented to 2. In the case where the bucket whose local depth is equal to the global depth overflows, i.e., B1 or B3 in Figure 1, the directory needs to be doubled. In so doing, both the global depth and the local depth of the two buckets that result from splitting the overflowing bucket also need to be incremented. Note, however, that in so doing, overhead is small as rehashing of the keys or moving of data only occur for keys within the bucket. With the larger global and local depths, the only change is that now, one more bit of the hash key is used to address the new buckets.

The main advantage of extendible hashing compared to other hashing schemes is that the rehashing overhead is independent of the index size. Also, unlike other static hash tables, no extra buckets need to be reserved for future growth that results in extendible hashing having higher space utilization than other hashing schemes [37]. The disadvantage of extendible hashing is that each hash table reference requires an extra access to the directory. Other static hashing schemes do not have this extra level of indirection, at the cost of full-table rehashing. However, it is known that the directory access incurs only minor performance overhead [23, 37].

PM-based Hashing: Recently a few hashing schemes, such as *Level Hashing* [49], *Path Hashing* [48], and PCM(Phase-Change Memory)-friendly hash table (PFHT) [3] have been proposed for persistent memory as the legacy in-memory hashing schemes fail to work on persistent memory due to the lack of consistency guarantees. Furthermore, persistent memory is expected to have limited endurance and asymmetric read-write latencies. We now review these previous studies.

PFHT is a variant of bucketized cuckoo hashing designed to reduce write accesses to PCM as it allows only one cuckoo displacement to avoid cascading writes. The insertion performance of cuckoo hashing is known to be about $20 \sim 30\%$ slower than the simplest linear probing [29]. Furthermore, in cuckoo hashing, if the load factor is above 50%, the expected insertion time is no longer constant. To improve the insertion performance of cuckoo hashing, PFHT uses a stash to defer full-table rehashing and improve the load factor. However, the stash is not a cache friendly structure as it linearly searches a long overflow chain when failing to find a key in a bucket. As a result, PFHT fails to guarantee the constant lookup cost, i.e., its lookup cost is not $O(1)$ but $O(\text{S})$ where S is the stash size.

Path hashing is similar to PFHT in that it uses a stash although the stash is organized as an inverted binary tree structure. With the binary tree structure, path hashing reduces the lookup cost. However, its lookup time is still not constant but in log scale, i.e., $O(log\text{B})$, where B is the number of buckets.

Level hashing consists of two hash tables organized in two levels. The top level and bottom level hash tables take turns playing the role of the stash. When the bottom level overflows, the records stored in the bottom level are rehashed to a $4\times$ larger hash table and the new hash table becomes the new top level, while the previous top level hash table becomes the new bottom level stash. Unlike path hashing and PFHT, level hashing guarantees constant lookup time.

While level hashing is an improvement over previous work, our analysis shows that the rehashing overhead is no smaller than legacy static hashing schemes. As the bottom level hash table is always almost full in level hashing, it fails to accommodate a collided record resulting in another rehash. The end result is that level hashing is simply performing a full-table rehash in two separate steps. Consider the following scenario. Say, we have a top level hash table that holds 100 records and the bottom level stash holds 50 records. Hence, we can insert 150 records without rehashing if a hash collision does not occur. When the next 151st insertion incurs a hash collision in the bottom level, the 50 records in the bottom level stash will be rehashed to a new top level hash table of size 200 such that we have 150 free slots. After the rehash, subsequent 150 insertions will make the top level hash table overflow. However, since the bottom level hash table does not have free space either, the 100 records in the bottom level hash table have to be rehashed. To expand a hash table size to hold 600 records, level hashing rehashes a total of 150 records, that is, 50 records for the first rehashing

and another 100 records for the second rehashing.

On the other hand, suppose the same workload is processed by a legacy hash table that can store 150 records as the initial level hash table does. Since the 151st insertion requires more space in the hash table, we increase the hash table size by four times instead of two as the level hashing does for the bottom level stash. Since the table now has 600 free spaces, we do not need to perform rehashing until the 601th insertion. Up to this point, we performed rehashing only once and only 150 records have been rehashed. Interestingly, the number of rehashed records are no different. We note that the rehashing overhead is determined by the hash table size, not by the number of levels. As we will show in Section 6, the overhead of rehashing in level hashing is no smaller than other legacy static hashing schemes.

To mitigate the shortage of space in bottom-level stash, level hashing proposes to use the bottom-to-top cuckoo displacement that evicts records from the bottom level stash to the top level hash table. However, in our experiments, when we insert 160 million records into a level hash table we observe the bottom-to-top cuckoo displacement occurs with a probability of 0.001% (only 1882 times) while rehashing occurs 14 times. As such, we find that in our experiments, bottom-to-top eviction rarely helps in improving the load factor or postponing rehashing.

One of the challenges in cuckoo displacement is that two cachelines need to be updated in a failure-atomic manner as we move a record into another bucket. If a system crashes during migration, there can be duplicate records after the system recovers. Suppose one of the duplicate records exists in the top level and the other record is in the bottom level. When a subsequent transaction updates the record, the one in the top level will be updated. Later, the top level hash table becomes the bottom level stash and another transaction will access the new top level hash table and find the stale record, which is not acceptable. Level hashing proposes to delete one of the two items when a subsequent transaction updates the item. Since every update transaction has to detect if there is a duplicate record, update transactions in level hashing needs to access other cachelines that have the possibility of having a duplicate record. In the worst case, each update transaction has to access every cacheline in each bucket referenced by two cuckoo hash functions in both levels. We note that such a worst case happens when there are no duplicate records, which would be the most common case in practice. To fix the problem in a more efficient way, we need to scan the entire hash table every time the system recovers from failure.

3 Cacheline-Conscious Extendible Hashing

In this section, we present Cacheline-Conscious Extendible Hashing (CCEH), a variant of extendible hashing that overcomes the shortcomings of traditional extendible hashing by guaranteeing failure-atomicity and reducing the number of cacheline accesses for the benefit of byte-addressable PM.

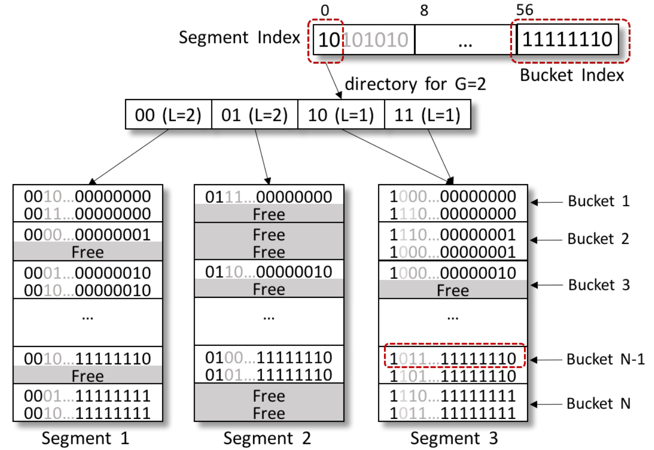


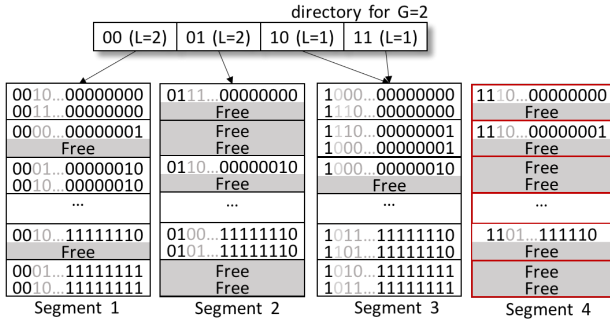
Figure 2: *Cacheline-Conscious Extendible Hashing*

3.1 Three Level Structure of CCEH

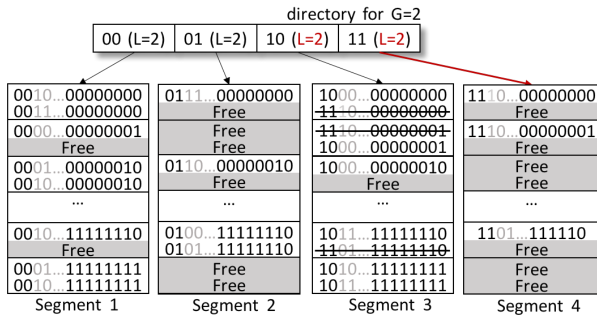
In byte-addressable PM, the unit of an atomic write is a word but the unit of data transfer between the CPU and memory corresponds to a cacheline. Therefore, the write-optimal size of a hash bucket is a cacheline. However, a cacheline, which is typically 64 bytes, can hold no more than four key-value pairs if the keys and values are word types. Considering that each cacheline-sized bucket needs an 8-byte pointer in the directory, the directory can be the tail wagging the dog, i.e., if each 64-byte bucket is pointed by a single 8-byte directory entry, the directory can be as large as 1/8 of the total bucket size. If multiple directory entries point to the same bucket, the directory size can be even larger. To keep the directory size under control, we can increase the bucket size. However, there is a trade-off between bucket size and lookup performance as increasing the bucket size will make lookup performance suffer from the large number of cacheline accesses and failure to exploit cache locality.

In order to strike a balance between the directory size and lookup performance, we propose to use an intermediate layer between the directory and buckets, which we refer to as a *segment*. That is, a segment in CCEH is simply a group of buckets pointed to by the directory. The structure of CCEH is illustrated in Figure 2. To address a bucket in the three level structure, we use the G bits (which represents the global depth) as a segment index and an additional B bits (which determines the number of cachelines in a segment) as a bucket index to locate a bucket in a segment.

In the example shown in Figure 2, we assume each bucket can store two records (delimited by the solid lines within the segments in the figure). If we use B bits as the bucket index, we can decrease the directory size by a factor of $1/2^B$ (1/256 in the example) compared to when the directory addresses each bucket directly. Note that although the three level structure decreases the directory size, it allows access to a specific



(a) Step 1: Create Sibling



(b) Step 2: Split and Lazy Deletion

Figure 3: *Failure-Atomic Segment Split Example*

bucket (cacheline) without accessing the irrelevant cachelines in the segment.

Continuing the example in Figure 2, suppose the given hash key is $10101010...11111110_{(2)}$ and we use the least significant byte as the bucket index and the first two leading bits as the segment index since the global depth is 2. We will discuss why we use the leading bits instead of trailing bits as the segment index later in Section 3.4. Using the segment index, we can lookup the address of the corresponding segment (Segment 3). With the address of Segment 3 and the bucket index ($11111110_{(2)}$), we can directly locate the address of the bucket containing the search key, i.e., ($\&\text{Segment3}+64 \times 11111110_{(2)}$). Even with large segments, the requested record can be found by accessing only two cachelines — one for the directory entry and the other for the corresponding bucket (cacheline) in the segment.

3.2 Failure-Atomic Segment Split

A split performs a large number of memory operations. As such, a segment split in CCEH cannot be performed by a single atomic instruction. Unlike full-table rehashing that requires a single failure-atomic update of the hash table pointer, extendible hashing is designed to reuse most of the segments and directory entries. Therefore, the segment split algorithm of extendible hashing performs several in-place updates in the directory and copy-on-writes.

In the following, we use the example depicted in Figure 3 to walk through the detailed workings of our proposed failure-atomic segment split algorithm. Suppose we are to insert key $1010...11111110_{(2)}$. Segment 3 is chosen as the leftmost bit is 1, but the 255th ($11111111_{(2)}$th) bucket in the segment has no free space, i.e., a hash collision occurs. To resolve the hash collision, CCEH allocates a new Segment and copies key-value records not only in the collided bucket of the segment but also in the other buckets of the same segment according to their hash keys. In the example, we allocate a new Segment 4 and copy the records, whose key prefix starts with 11, from Segment 3 to Segment 4. We use the two leading bits because the local depth of Segment 3 will be increased to 2. If the prefix is 10, the record remains in Segment 3, as illustrated in Figure 3(a).

In the next step, we update the directory entry for the new Segment 4 as shown in Figure 3(b). First, (1) the pointer and the local depth for the new bucket are updated. Then, (2) we update the local depth of the segment that we split, Segment 3. I.e., we update the directory entries from right to left. The ordering of these updates must be enforced by inserting an `mfence` instruction in between each instruction. Also, we must call `clflush` when it crosses the boundary of cachelines, as was done in FAST and FAIR B-tree [11]. Enforcing the order of these updates is particularly important to guarantee recovery. Note that these three operations cannot be done in an atomic manner. That is, if a system crashes during the segment split, the directory can find itself in a partially updated inconsistent state. For example, the updated pointer to a new segment is flushed to PM but two local depths are not updated in PM. However, we note that this inconsistency can be easily detected and fixed by a recovery process without explicit logging. We detail our recovery algorithm later in Section 4.

A potential drawback of our split algorithm for three level CCEH is that a hash collision may split a large segment even if other buckets in the same segment have free space. To improve space utilization and avoid frequent memory allocation, we can employ ad hoc optimizations such as *linear probing* or *cuckoo displacement*. Although these ad hoc optimizations help defer expensive split operations, they increase the number of cacheline accesses and degrade the index lookup performance. Thus, they must be used with care. In modern processors, serial memory accesses to adjacent cachelines benefit from hardware prefetching and memory level parallelism [11]. Therefore, we employ simple linear probing that bounds the number of buckets to probe to four cachelines to leverage memory level parallelism.

Similar to the segment split, a segment merge performs the same operations, but in reverse order. That is, (1) we migrate the records from the right segment to the left segment. Next, (2) we decrease the local depths and update pointers of the two segments in the directory. Note that we must update these directory entries from left to right, which is the

opposite direction to that used for segment splits. This ordering is particularly important for recovery. Details about the ordering and recovery will be discussed in Section 4.

3.3 Lazy Deletion

In legacy extendible hashing, a bucket is atomically cleaned up via a page write after a split such that the bucket does not have migrated records. For failure-atomicity, disk-based extendible hashing updates the local depth and deletes migrated records with a single page write.

Unlike legacy extendible hashing, CCEH does not delete migrated records from the split segment. As shown in Figure 3(b), even if Segments 3 and 4 have duplicate key-value records, this does no harm. Once the directory entry is updated, queries that search for migrated records will visit the new segment and queries that search for non-migrated records will visit the old segment but they always succeed in finding the search key since the split Segment 3 contains all the key-value records, with some unneeded duplicates.

Instead of deleting the migrated records immediately, we propose *lazy deletion*, which helps avoid the expensive copy-on-write and reduce the split overhead. Once we increase the local depth of the split segment in the directory entry, the migrated keys (those crossed-out keys in Figure 3(b)) will be considered invalid by subsequent transactions. Therefore, there is no need to eagerly overwrite migrated records because they will be ignored by read transactions and they can be overwritten by subsequent insert transactions in a lazy manner. For example, if we insert a record whose hash key is $1010...11111110_{(2)}$, we access the second to last bucket of Segment 3 (in Figure 3(b)) and find the first record's hash key is $1000...11111110_{(2)}$, which is valid, but the second record's hash key is $1101...11111110_{(2)}$, which is invalid. Then, the insert transaction replaces the second record with the new record. Since the validity of each record is determined by the local depth, the ordering of updating directory entries must be preserved for consistency and failure-atomicity.

3.4 Segment Split and Directory Doubling

Although storing a large number of buckets in each segment can significantly reduce the directory size, directory doubling is potentially the most expensive operation in large CCEH tables. Suppose the segment pointed to by the first directory entry splits, as shown in Figure 4(a). To accommodate the additional segment, we need to double the size of the directory and make each existing segment referenced by two entries in the new directory. Except for the two new segments, the local depths of existing segments are unmodified and they are all smaller than the new global depth.

For disk-based extendible hashing, it is well known that using the least significant bits (LSB) allows us to reuse the directory file and to reduce the I/O overhead of directory

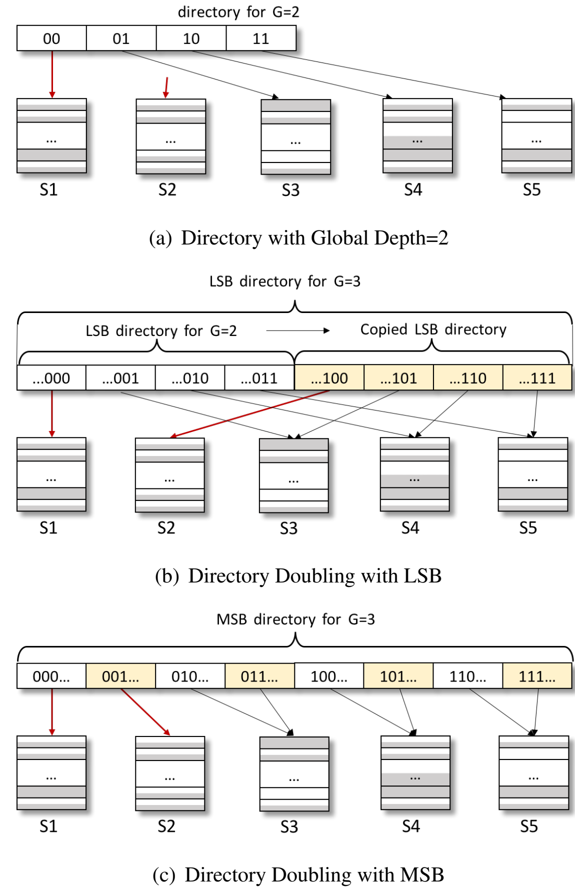


Figure 4: *MSB segment index makes adjacent directory entries be modified together when a segment splits*

doubling because we can just copy the directory entries as one contiguous block and append it to the end of the file as shown in Figure 4(b). If we use the most significant bits (MSB) for the directory, new directory entries have to be sandwiched in between existing entries, which makes all pages in the directory file dirty.

Based on this description, it would seem that making use of the LSB bits would be the natural choice for PM as well. In contrary, however, it turns out when we store the directory in PM, using the most significant bits (MSB) performs better than using the LSB bits. This is because the existing directory entries cannot be reused even if we use LSB since all the directory entries need to be stored in contiguous memory space. That is, when using LSB, we must allocate twice as much memory as the old directory uses, copy the old directory to the first half as well as to the second half.

The directory doubling is particularly expensive because of cacheline flushes that are required for failure atomicity. In fact, the overhead of doubling the directory with two `memcpy()` function calls and iterating through a loop to duplicate each directory entry is minimal compared to the

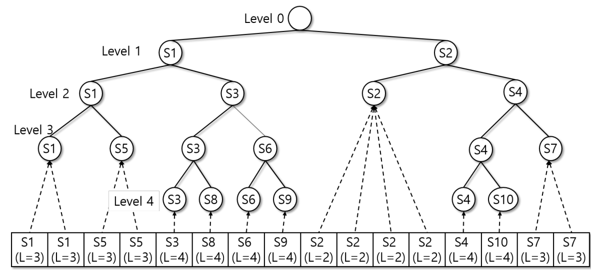
overhead of `clflush`. Note that when we index 16 million records using 16 KByte segments, it takes 555 usec and 631 usec to double the directory when we use LSB and MSB respectively. However, `clflush()` takes about 2 msec (3$\sim$4$\times$ higher). In conclusion, LSB does not help reduce the overhead of enlarging the directory size unlike the directory file on disks.

The main advantage of using MSB over LSB comes from reducing the overhead of segment splits, not from reducing the overhead of directory doubling. If we use MSB for the directory, as shown in Figure 4(c), the directory entries for the same segment will be adjacent to each other such that they benefit from spatial locality. That is, if a segment splits later, multiple directory entries that need to be updated will be adjacent. Therefore, using MSB as segment index reduces the number of cacheline flushes no matter what local depth a split segment has. We note, however, that even though this has a positive effect of reducing the overhead for directory doubling, in terms of performance, it is more important to reduce the overhead of segment splits as segment splits occur much more frequently. Even though preserving the spatial locality has little performance effect on reducing the overhead of directory doubling because both MSB and LSB segment index call the same number of `clflush` instructions in batches when doubling the directory, MSB segment index has a positive effect of reducing the overhead of segment splits, which occur much more frequently than directory doubling. As we will see next, using MSB has another benefit of allowing for easier recovery.

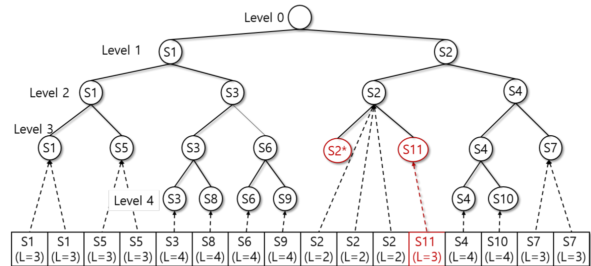
4 Recovery

Various system failures such as power loss can occur while hash tables are being modified. Here, we present how CCEH achieves failure-atomicity by discussing system failures at each step of the hash table modification process.

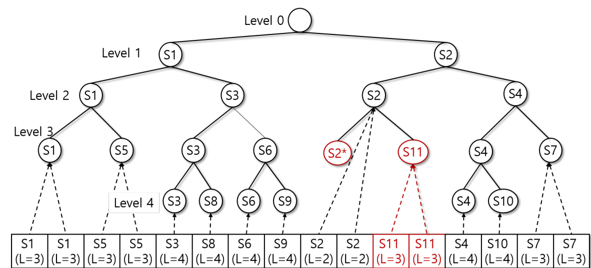
Suppose a system crashes when we store a new record into a bucket. First, we store the value and its key next. If the key is of 8 bytes, the key can be atomically stored using the key itself as a commit mark. Even if the key is larger than 8 bytes, we can make use of the leading 8 bytes of the key as a commit mark. For example, suppose the key type is a 32 byte string and we use the MSB bits as the segment index and the least significant byte as the bucket index. We can write the 24 byte suffix first, call `mfence`, store the leading 8 bytes as a commit mark, and call `clflush`. This ordering guarantees that the leading 8 bytes are written after all the other parts of the record have been written. Even if the cacheline is evicted from the CPU cache, partially written records will be ignored because the key is not valid for the segment, i.e., the MSB bits are not a valid segment index. This is the same situation as when our lazy deletion considers a slot with any invalid MSB segment index as free space. Therefore, the partially written records without the correct leading 8 bytes will



(a) Tree Representation of Segment Split History



(b) Split: Update Pointer and Level for new Segment from Right to Left



(c) Split: Increase Level of Split Segment from Right to Left

Figure 5: *Buddy Tree Traversal for Recovery*

be ignored by subsequent transactions. Since all hash tables including CCEH initialize new hash tables or segments when they are first allocated, there is no chance for an invalid key to have a valid MSB segment index by pure luck. To delete a record, we change the leading 8 bytes to make the key invalid for the segment. Therefore, the insertion and deletion operations that do not incur bucket splits are failure-atomic in CCEH.

Making use of the MSB bits as a segment index not only helps reduce the number of cacheline flushes but also makes the recovery process easy. As shown in Figure 5, with the MSB bits, the directory entries allow us to keep track of the segment split history as a binary buddy tree where each node in the tree represents a segment. When a system crashes, we visit directory entries as in binary tree traversal and check their consistency, which can be checked by making use of G and L. That is, we use the fact that, as we see in Figure 3, if G is larger than L then the directory buddies must point to the same segment, while if G and L are equal, then each must point to different segments.

Algorithm 1 Directory Recovery

```
1: while i < Directory.Capacity do
2:     Depth_Cur ← Directory[i].Depth_local
3:     Stride ← 2^(Depth_global − Depth_Cur)
4:     j ← i + Stride                                  ▷ Buddy Index
5:     Depth_Buddy ← Directory[j].Depth_local
6:     if Depth_Cur < Depth_Buddy then                 ▷ Left half
7:         for k ← j − 1; i < k; k ← k − 1 do
8:             Directory[k].Depth_local ← Depth_Cur
9:     else
10:        if Depth_Cur = Depth_Buddy then             ▷ Right half
11:            for k ← j + 1; k < j + Stride; k ← k + 1 do
12:                Directory[k] ← Directory[j]
13:        else                         ▷ Depth_Cur > Depth_Buddy; Shrink
14:            for k ← j + Stride − 1; j <= k; k ← k − 1 do
15:                Directory[k] ← Directory[j + Stride − 1]
16:    i ← i + 2^(Depth_global − (Depth_Cur − 1))
```

Let us now see how we traverse the directories. Note that the local depth of each segment and the global depth determine the segment's stride in the directory, i.e., how many times the segment appears contiguously in the directory. Since the leftmost directory entry is always mapped to the root node of the buddy tree because of the in-place split algorithm, we first visit the leftmost directory entry and check its buddy entry. In the walking example, the buddy of S1 (directory[0]) is S5 (directory[2]) since its stride is $2^{G-L} = 2$. After checking the local depth and pointer of its right buddy, we visit the parent node by decreasing the local depth by one. I.e., S1 in level 2. Now, the stride of S1 in level 2 is $2^{G-L} = 4$. Hence, we visit S3 (directory[4]) and check its local depth. Since the local depth S3 is higher (4 in the example), we can figure out that S3 has split twice and its stride is 1. Hence, we visit directory[5] and check its consistency, continuing this check until we find any inconsistency. The pseudo code of this algorithm is shown in Algorithm 1.

Suppose a system crashes while splitting segment S2 in the example. According to the split algorithm we described in Section 3.2, we update the directory entries for the split segment from right to left. Say, a system crashes after making directory[11], colored red in the Figure 5(b), point to a new segment S11. The recovery process will traverse the buddy tree and visit directory[8]. Since the stride of S2 is 4, the recovery process will make sure directory[9], directory[10], and directory[11] have the same local depth and point to the same segment. Since directory[11] points to a different segment, we can detect the inconsistency and fix it by restoring its pointer. If a system crashes after we update directory[10] and directory[11] as shown in Figure 5(c), we can either restore the two buddies or increase the local depth of directory[8] and directory[9].

5 Concurrency and Consistency Model

Rehashing is particularly challenging when a large number of transactions are concurrently running because rehashing requires all concurrent write threads to wait until rehashing is complete. To manage concurrent accesses in a thread-safe way in CCEH, we adapt and make minor modifications to the two level locking scheme proposed by Ellis [5], which is known to show reasonable performance for extendible hashing [24]. For buckets, we protect them using a reader/writer lock. For segments, we have two options. One option is that we protect each segment using a reader/writer lock as with buckets. The other option is the lock-free access to segments.

Let us first describe the default reader/writer lock option. Although making use of a reader/writer lock for each segment access is expensive, this is necessary because of the in-place lazy deletion algorithm that we described in Section 3.2. Suppose a read transaction T1 visits a segment but goes to sleep before reading a record in the segment. If we do not protect the segment using a reader/writer lock, another write transaction T2 can split the segment and migrate the record to a new segment. Then, another transaction accesses the split segment and overwrites the record that the sleeping transaction is to read. Later, transaction T1 will not find the record although the record exists in the new buddy segment.

The other option is lock-free access. Although lock-free search cannot enforce the ordering of transactions, which makes queries vulnerable to *phantom and dirty reads* problems [37], it is useful for certain types of queries, such as OLAP queries, that do not require a strong consistency model because lock-free search helps reduce query latency.

To enable lock-free search in CCEH, we cannot use the lazy deletion and in-place updates. Instead, we can copy-on-write (CoW) split segments. With CoW split, we do not overwrite any existing record in the split segment. Therefore, a lock-free query accesses the old split segment until we replace the pointer in the directory with a new segment. Unless we immediately deallocate the split segment, the read query can find the correct key-value records even after the split segment is replaced by two new segments. To deallocate the split segment in a thread-safe way, we keep count of how many read transactions are referencing the split segment. If the reference count becomes zero, we ask the persistent heap memory manager to deallocate the segment. As such, a write transaction can split a segment even while it is being accessed by read transactions.

We note that the default CCEH with lazy deletion has a much smaller overhead for segment split than the CCEH with CoW split, which we denote as `CCEH(C)`, because it reuses the original segment so that it can allocate and copy only half the amount required for `CCEH(C)`. If a system failure occurs during a segment split, the recovery cost for lazy deletion is also only half of that of `CCEH(C)`. On the other hand, `CCEH(C)` that enables lock-free search at the cost of

weak consistency guarantee and higher split overhead shows faster and more scalable search performance, as we will show in Section 6. Another benefit of CCEH(C) is that its probing cost for search operations is smaller than that of CCEH with lazy deletion because all the invalid keys are overwritten as NULL.

For more scalable systems, lock-free extendible hashing has been studied by Shalev et al. [36]. However, such lock-free extendible hashing manages each key-value record as a *split-ordered* list, which fails to leverage memory level parallelism and suffers from a large number of cacheline accesses.

To minimize the impact of rehashing and reduce the tail latency, numerous hash table implementations including Java Concurrent Package and Intel Thread Building Block partition the hash table into small regions and use an exclusive lock for each region [8, 12, 21, 27], hence avoiding full-table rehashing. Such region-based rehashing is similar to our CCEH in the sense that CCEH rehashes only one segment at a time. However, we note that the existing region-based concurrent hash table implementations are not designed to guarantee failure-atomicity for PM. Furthermore, their concurrent hash tables use separate chaining hash tables, not dynamic hash tables [8, 12, 21, 27].

6 Experiments

We run experiments on a workstation that has four Intel Xeon Haswell-EX E7-4809 v3 processors (8 cores, 2.0GHz, 8×32KB instruction cache, 8×32KB data cache, 8×256KB L2 cache, and 20MB L3 cache) and 64GB of DDR3 DRAM. Since byte-addressable persistent main memory is not commercially available yet, we emulate persistent memory using *Quartz*, a DRAM-based PM latency emulator [9, 41]. To emulate write latency, we inject stall cycles after each `clflush` instructions, as was done in previous studies [10, 20, 15, 35, 42].

A major reason to use dynamic hashing over static hashing is to dynamically expand or shrink hash table sizes. Therefore, we set the initial hash table sizes such that they can store only a maximum of 2048 records. For all experiments, we insert 160 million random keys, whose keys and values are of 8 bytes. Although we do not show experimental results for non-uniformly distributed keys such as skewed distributions due to the page limit, the results are similar because well designed hash functions convert a non-uniform distribution into one that is close to uniform [1].

6.1 Quantification of CCEH Design

In the first set of experiments, we quantify the performance effect of each design of CCEH. Figure 6 shows the insertion throughput and the number of cacheline flushes when we insert 160 million records into variants of the extendible hash table, while increasing the size of the memory blocks pointed by directory entries, i.e., the segment in CCEH and the hash bucket in extendible hashing. We fix the size of the bucket in CCEH to a single cacheline, but employ linear probing and bound the probing distance to four cachelines to leverage memory level parallelism.

CCEH(MSB) and CCEH(LSB) show the performance of CCEH when using MSB and LSB bits, respectively, as the segment index and LSB and MSB bits, respectively, as the bucket index. EXTH(LSB) shows the performance of legacy extendible hashing that uses LSB as the bucket index, which is the popular practice.

When the bucket size is 256 bytes, each insertion into EXTH(LSB) calls `clflush` instructions about 3.5 times on average. Considering an insertion without collision requires only a single `clflush` to store a record in a bucket, 2.5 cacheline flushes are the amortized cost of bucket splits and directory doubling. Note that CCEH(LSB) and EXTH(LSB) are the same hash tables when a segment can hold a single bucket. Therefore, their throughputs and number of cacheline accesses are similar when the segment size of CCEH(LSB) and the bucket size of EXTH(LSB) are 256 bytes.

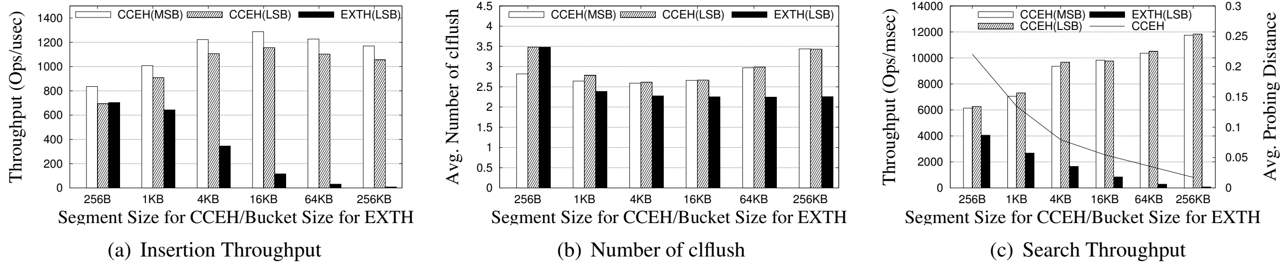
As we increase the bucket size, EXTH(LSB) splits buckets less frequently, decreasing the number of `clflush` down to 2.3. However, despite the fewer number of `clflush` calls, the insertion and search throughput of EXTH(LSB) decreases sharply as we increase the bucket size. This is because EXTH(LSB) reads a larger number of cachelines to find free space as the bucket size increases.

In contrast, as we increase the segment size up to 16KB, the insertion throughput of CCEH(MSB) and CCEH(LSB) increase because segment splits occur less frequently while the number of cachelines to read, i.e., LLC (Last Level Cache) misses, is not affected by the large segment size. However, if the segment size is larger than 16KB, the segment split results in a large number of cacheline flushes, which starts degrading the insertion throughput.

Figure 6(b) shows CCEH(MSB) and CCEH(LSB) call a larger number of `clflush` than EXTH(LSB) as the segment size grows. This is because CCEH(MSB) and CCEH(LSB) store records in a sparse manner according to the bucket index whereas EXTH(LSB) sequentially stores rehashed records without fragmented free spaces. Thus, the number of updated cachelines written by EXTH(LSB) is only about two-third of CCEH(LSB) and CCEH(MSB). From the experiments, we observe the reasonable segment size is in the range of 4KB to 16KB.

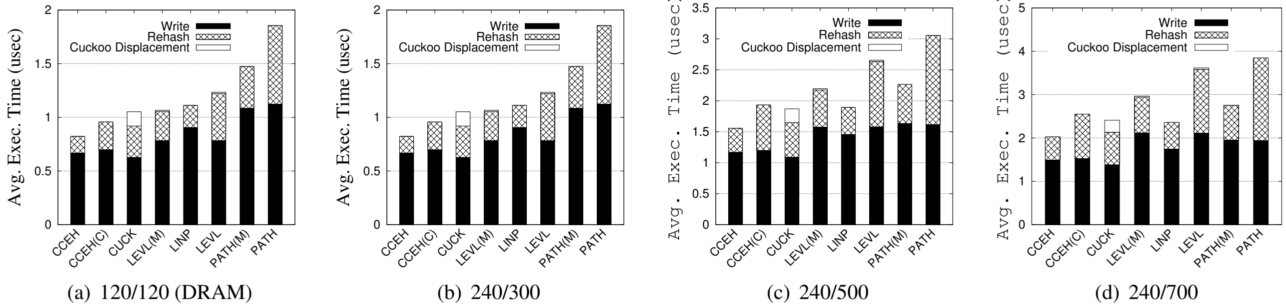
When the segment size is small, the amortized cost of segment splits in CCEH(MSB) is up to 29% smaller than that of CCEH(LSB) because CCEH(MSB) updates adjacent directory entries, minimizing the number of `clflush` instructions. However, CCEH(LSB) accesses scattered cachelines and fails to leverage memory level parallelism, which results in about 10% higher insertion time on average.

It is noteworthy that the search performance of



(a) Insertion Throughput (b) Number of clflush (c) Search Throughput

Figure 6: *Throughput with Varying Segment/Bucket Size*



(a) 120/120 (DRAM) (b) 240/300 (c) 240/500 (d) 240/700

Figure 7: *Breakdown of Time Spent for Insertion While Varying R/W latency of PM*

CCEH(MSB) and CCEH(LSB) improves as the segment size grows. This is because the larger the segment size, the more bits CCEH uses to determine which cacheline in the segment needs to be accessed, which helps CCEH perform linear probing less frequently. Figure 6(c) shows the average number of extra cache line accesses per query caused by linear probing. As we increases the segment size, the average probing distance decreases from 0.221 cacheline to 0.017 cacheline.

6.2 Comparative Performance

For the rest of the experiments, we use a single byte as the bucket index such that the segment size is 16 Kbytes, and we do not show the performance of CCEH(LSB) since CCEH(MSB) consistently outperforms CCEH(LSB). We compare the performance of CCEH against a static hash table with linear probing (LINP), cuckoo hashing [29] (CUCK), path hashing [48] (PATH), and level hashing [49] (LEVL).[1]

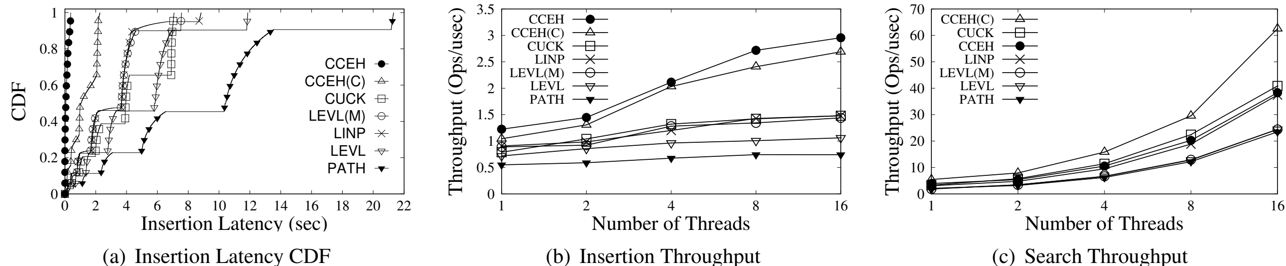
For path hashing, we set the reserved level to 8, which achieves 92% maximum load factor as suggested by the authors [48]. For cuckoo hashing, we let CUCK perform full-table rehashing when it fails to displace a collided record 16 times, which shows the fastest insertion performance on our testbed machine. Linear probing rehashes when the load factor reaches 95%.

[1]Our implementations of CCEH, linear probing (LINP), and cuckoo hashing (CUCK) are available at https://github.com/DICL/CCEH. For path hashing (PATH) and level hashing (LEVL), we downloaded the authors' implementations from https://github.com/Pfzuo/Level-Hashing.

In the experiments shown in Figure 7, as the latency for reads and writes of PM are changed, we insert 160 million records in batches and breakdown the insertion time into (1) the bucket search and write time (denoted as Write), (2) the rehashing time (denoted as Rehash), and (3) the time to displace existing records to another bucket, which is necessary for cuckoo hashing (denoted as Cuckoo Displacement).

CCEH shows the fastest average insertion time throughout all read/write latencies. Even if we disable lazy deletion but perform copy-on-write for segment splits, denoted as CCEH(C), CCEH(C) outperforms LEVL. Note that the Rehash overhead of CCEH(C) is twice higher than that of CCEH that reuses the split segment via lazy deletion. However, as the write latency of PM increases, CCEH(C) is outperformed by CUCK and LINP because of frequent memory allocations and expensive copy-on-write operations.

Interestingly, the rehashing overhead of LEVL is even higher than that of LINP, which is just a single array that employs linear probing for hash collisions. Although LINP suffers from a large number of cacheline accesses due to open addressing, its rehashing overhead is smaller than all the other hashing schemes except CCEH. We note that the rehashing overhead of LEVL and PATH is much higher than that of LINP because the rehashing implementation of LEVL calls clflush to delete each record in the bottom level stash when rehashing it to the new enlarged hash table. This extra clflush is unnecessary for LINP and CUCK, because we can simply deallocate the previous hash table when the new hash table is ready. If a system crashes before the new hash table



(a) Insertion Latency CDF (b) Insertion Throughput (c) Search Throughput

Figure 8: *Performance of concurrent execution: latency CDF and insertion/search throughput*

is ready, we discard the new hash table and perform rehashing from the beginning. As LEVL and PATH can employ the same rehashing strategy, we implement the improved rehashing code for them, denoted as LEVL(M) and PATH(M). With the modification, LEVL(M) shows similar rehashing overhead with CCEH(C). However, it is outperformed by CCEH and LINP because its two-level structure and ad hoc optimizations such as bucketization increases the number of cacheline accesses. Note that the bucket search and write time (Write) of LEVL(M) is higher than that of CCEH and even CUCK. It is noteworthy that LEVL performs the cuckoo displacement much less frequently than CUCK and its overhead is almost negligible.

PATH hashing shows the worst performance throughout all our experiments mainly because its lookup cost is not constant, but $O(log_2\text{N})$. As the write latency increases, the performance gap between LEVL and PATH narrows down because the lookup cost becomes relatively inexpensive compared to the Write time.

6.3 Concurrency and Latency

Full-table rehashing is particularly challenging when multiple queries are concurrently accessing a hash table because it requires exclusive access to the entire hash table, which blocks subsequent queries and increases the response time. Therefore, we measure the latency of concurrent insertion queries including the waiting time, whose CDF is shown in Figure 8(a). For the workload, we generated query inter-arrival patterns using Poisson distribution where the λ rate is set to the batch processing throughput of LINP.

While the average batch insertion times differ by only up to 180%, the maximum latency of PATH is up to 56× higher than that of CCEH (378 msec vs. 21.3 sec), as shown in Figure 8(a). This is because full-table rehashing blocks a large number of concurrent queries and significantly increases their waiting time. The length of each flat region in the CDF graph represents how long each full-table rehashing takes. PATH takes the longest time for rehashing whereas LEVL, LINP, and CUCK spend a similar amount of time on rehashing. In contrast, we do not find any flat region in the graph for CCEH. Compared to LEVL, the maximum latency of CCEH is reduced by over 90%.

For the experimental results shown in Figures 8(b) and (c), we evaluate the performance of the multi-threaded versions of the hashing schemes. Each thread inserts 160/*k* million records in batches where *k* is the number of threads. Overall, as we run a larger number of insertion threads, the insertion throughputs of all hashing schemes improve slightly but not linearly due to lock contention.

Individually, CCEH shows slightly higher insertion throughput than CCEH(C) because of smaller split overhead. LEVL, LINP, CUCK, and PATH use a fine-grained reader/writer lock for each sub-array that contains 256 records (4 KBytes), which is even smaller than the segment size of CCEH (16 KBytes), but they fail to scale because of the rehashing overhead. We note that these static hash tables must obtain exclusive locks for all the fine-grained sub-arrays to perform rehashing. Otherwise, queries will access a stale hash table and return inconsistent records.

In terms of search throughput, CCEH(C) outperforms CCEH as CCEH(C) enables lock-free search by disabling lazy deletion and in-place updates as we described in Section 5. Since the read transactions of CCEH(C) are non-blocking, search throughput of CCEH(C) is 1.63×, 1.53×, and 2.74× higher than that of CCEH, CUCK, and LEVL, respectively. Interestingly, LEVL shows worse search performance than LINP. Since level hashing uses cuckoo displacement and two-level tables, which accesses noncontiguous cachelines multiple times, it fails to leverage memory level parallelism and increases the LLC misses. In addition, level hashing uses small-sized buckets as in bucketized hashing and performs linear probing for at most four buckets, which further increases the number of cacheline accesses, hurting search performance even more. As a result, LEVL shows poor search throughput.

While the results in Figure 8(c) were for queries where the lookup keys all existed in the hash table, Figure 9 shows search performance for non-existent keys. Since CUCK accesses no more than two cachelines, it shows even higher search performance than CCEH, which accesses up to four cachelines due to linear probing. Although LINP shows similar search performance with CCEH for positive queries, it suffers from long probing distance for negative queries and

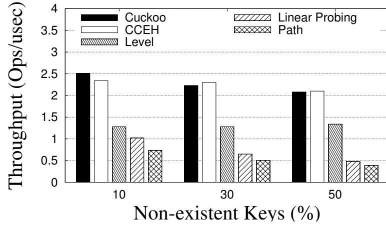


Figure 9: *Negative search*

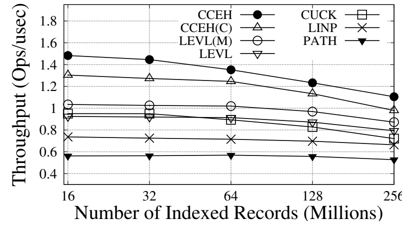


Figure 10: *YCSB throughput (Workload D: Read Latest)*

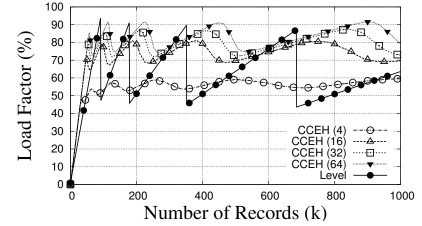


Figure 11: *Load factor per thousand insertions*

shows very poor search performance. We see that LEVL also suffers as making use of long probing, cuckoo displacement, and stash hurts search performance even more. Interestingly, PATH shows even worse search performance than LINP because of its non-constant lookup time.

We now consider the YCSB benchmarks representing realistic workloads. Figure 10 shows the throughput results of YCSB workload D as we vary the number of indexed records. In the workload, 50% of the queries insert records of size 32 bytes, while the other 50% read recently inserted records. As we increase the number of indexed records, the size of CCEH grows from 593 MBytes to 8.65 GBytes. Since hashing allows for constant time lookups, insertion and search throughput of most of the hashing schemes are insensitive to the size of hash tables. However, we observe that throughput of CCEH decreases linearly because of the hierarchical structure. When CCEH indexes 16 million records, the directory size is only 1 MBytes. Since the directory is more frequently accessed than segments, it has a higher probability of being in the CPU cache. However, when CCEH indexes 256 million records, the directory size becomes 16 MBytes while the total size of all segments is 8 GBytes. Considering that the LLC size of our testbed machine is 20 MBytes, the LLC miss ratio for the directory increases as the directory size grows. As a result, search performance of CCEH becomes similar to that of LEVL and CUCK when we index more than 64 million records and the throughput gap between CCEH and LEVL(M) narrows down.

6.4 Load Factor and Recovery Overhead

Figure 11 shows the memory utilization of CCEH and LEVL. The load factor of LEVL fluctuates between 50% and 90% because of the full-table rehashing. On each rehash, the bottom level hash table is quadrupled and the load factor drops down to 50%, which is no different from other static hash tables as we discussed in Section 2. In contrast, CCEH shows more smooth curves as it dynamically allocates small segments. Note that we can improve the load factor by increasing the linear probing distance as CCEH allocates a new segment when linear probing fails to insert a record into adjacent buckets. When we set the linear probing distance to 4 and 16, the load factor of CCEH, denoted as CCEH(4) and CCEH(16), range from 50% to 60% and from 70% to 80%, respectively. As we increase the distance up to 64, the load factor of CCEH increases up to 92%. However, as we increase the linear probing distance, the overall insertion and search performance suffers from the larger number of cacheline accesses.

While recovery is trivial in other static hash tables, CCEH requires a recovery process. To measure the recovery latency of CCEH, we varied the number of indexed records and deliberately injected faults. When we insert 32 million and 128 million records, the directory size is only 2 MBytes and 8 MBytes, respectively, and our experiments show that recovery takes 13.7 msec and 59.5 msec, respectively.

7 Conclusion

In this work, we presented the design and implementation of the cacheline-conscious extendible hash (CCEH) scheme, a failure-atomic variant of extendible hashing [6], that makes effective use of cachelines to get the most benefit out of byte-addressable persistent memory. By introducing an intermediate layer between the directory and cacheline-sized buckets, CCEH effectively reduces the directory management overhead and finds a record with at most two cacheline accesses. Our experiments show that CCEH eliminates the full-table rehashing overhead and outperforms other hash table schemes by a large margin on PM as well as DRAM.

Acknowledgments

We would like to give our special thanks to our shepherd Dr. Vasily Tarasov and the anonymous reviewers for their valuable comments and suggestions. This work was supported by the R&D program of NST (grant B551179-12-04-00) and ETRI R&D program (grant 18ZS1220), National Research Foundation of Korea (NRF) (grant No. NRF-2018R1A2B3006681 and NRF-2016M3C4A7952587), and Institute for Information & Communications Technology Promotion(IITP) (grant No. 2018-0-00549) funded by Ministry of Science and ICT, Korea. The corresponding author is Beomseok Nam.

References

[1] CARTER, J. L., AND WEGMAN, M. N. Universal classes of hash functions (extended abstract). In *Proceedings of the ACM 9th Symposium on Theory of Computing (STOC)* (1977), pp. 106–112.

[2] CHEN, S., AND JIN, Q. Persistent B+-Trees in non-volatile main memory. *Proceedings of the VLDB Endowment (PVLDB) 8*, 7 (2015), 786–797.

[3] DEBNATH, B., HAGHDOOST, A., KADAV, A., KHATIB, M. G., AND UNGUREANU, C. Revisiting hash table design for phase change memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads* (2015), INFLOW '15, pp. 1:1–1:9.

[4] DIETZFELBINGER, M., AND WEIDLING, C. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science 380*, 1-2 (2007), 47–68.

[5] ELLIS, C. S. Extendible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (New York, NY, USA, 1983), PODS '83, ACM, pp. 106–116.

[6] FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, H. R. Extendible hashing - a fast access method for dynamic files. *ACM Trans. Database Syst. 4*, 3 (Sept. 1979).

[7] FANG, R., HSIAO, H.-I., HE, B., MOHAN, C., AND WANG, Y. High performance database logging using storage class memory. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)* (2011), pp. 1221–1231.

[8] GOETZ, B. Building a better HashMap: How ConcurrentHashMap offers higher concurrency without compromising thread safety, 2003. *https://www.ibm.com/developerworks/java/library/j-jtp08223/*.

[9] HPE. Quartz, 2018. *https://github.com/HewlettPackard/quartz*.

[10] HUANG, J., SCHWAN, K., AND QURESHI, M. K. Nvram-aware logging in transaction systems. *Proceedings of the VLDB Endowment 8*, 4 (2014).

[11] HWANG, D., KIM, W.-H., WON, Y., AND NAM, B. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Trees. In *Proceedings of the 11th USENIX Conference on File and Storage (FAST)* (2018).

[12] INTEL. Intel Threading Building Blocks Developer Reference , 2018. *https://software.intel.com/en-us/tbb-reference-manual*.

[13] IZRAELEVITZ, J., KELLY, T., AND KOLLI, A. Failure-atomic persistent memory updates via JUSTDO logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages (ASPLOS))* (2016).

[14] JOHNSON, L. An indirect chaining method for addressing on secondary keys. *Communications of the ACM 4*, 5 (1961), 218–222.

[15] KIM, W.-H., KIM, J., BAEK, W., NAM, B., AND WON, Y. NVWAL: Exploiting NVRAM in write-ahead logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[16] KIM, W.-H., SEO, J., KIM, J., AND NAM, B. clfB-tree: Cacheline friendly persistent B-tree for NVRAM. *ACM Transactions on Storage (TOS), Special Issue on NVM and Storage* (2018).

[17] KNOTT, G. D. Expandable open addressing hash table storage and retrieval. In *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control* (1971), ACM, pp. 187–206.

[18] KOLLI, A., PELLEY, S., SAIDI, A., CHEN, P. M., AND WENISCH, T. F. High-performance transactions for persistent memories. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016), pp. 399–411.

[19] LARSON, P.-Å. Dynamic hashing. *BIT Numerical Mathematics 18*, 2 (1978), 184–201.

[20] LEE, S. K., LIM, K. H., SONG, H., NAM, B., AND NOH, S. H. WORT: Write optimal radix tree for persistent memory storage systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)* (2017).

[21] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 27.

[22] LITWIN, W. Virtual hashing: A dynamically changing hashing. In *Proceedings of the 4th International Conference on Very Large Data Bases-Volume 4* (1978), VLDB Endowment, pp. 517–523.

[23] MENDELSON, H. Analysis of extendible hashing. *IEEE Transactions on Software Engineering*, 6 (1982), 611–619.

[24] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2002).

[25] MORRIS, R. Scatter storage techniques. *Communications of the ACM 11*, 1 (1968), 38–44.

[26] ORACLE. Architectural Overview of the Oracle ZFS Storage Appliance, 2018. *https://www.oracle.com/technetwork/server-storage/sun-unified-storage/documentation/o14-001-architecture-overview-zfsa-2099942.pdf*.

[27] ORACLE. Java Platform, Standard Edition 7 API Specification, 2018. *https://docs.oracle.com/javase/7/docs/api/java/util/conc urrent/ConcurrentHashMap.html*.

[28] OUKID, I., LASPERAS, J., NICA, A., WILLHALM, T., AND LEHNER, W. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2016).

[29] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[30] PATIL, S., AND GIBSON, G. A. Scale and concurrency of giga+: File system directories with millions of files. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2011), vol. 11, pp. 13–13.

[31] PETERSON, W. W. Addressing for random-access storage. *IBM journal of Research and Development 1*, 2 (1957), 130–146.

[32] RUDOFF, A. Programming models for emerging non-volatile memory technologies. *;login 38*, 3 (June 2013), 40–45.

[33] SCHMUCK, F. B., AND HASKIN, R. L. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2002), vol. 2.

[34] SEHGAL, P., BASU, S., SRINIVASAN, K., AND VORUGANTI, K. An empirical study of file systems on nvm. In *Proceedings of the 31st International Conference on Massive Stroage Systems (MSST)* (2015).

[35] SEO, J., KIM, W.-H., BAEK, W., NAM, B., AND NOH, S. H. Failure-atomic slotted paging for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[36] SHALEV, O., AND SHAVIT, N. Split-ordered lists: Lock-free extensible hash tables. *J. ACM 53*, 3 (May 2006), 379–405.

[37] SILBERSCHATZ, A., KORTH, H., AND SUDARSHAN, S. *Database Systems Concepts*. McGraw-Hill, 2005.

[38] SOLTIS, S. R., RUWART, T. M., AND OKEEFE, M. T. The global file system. In *Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies* (1996), vol. 2, pp. 319—-342.

[39] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)* (2003), pp. 43–58.

[40] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)* (2011).

[41] VOLOS, H., MAGALHAES, G., CHERKASOVA, L., AND LI, J. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 15th Annual Middleware Conference (Middleware '15)* (2015).

[42] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[43] WEISS, Z., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Densefs: a cache-compact filesystem. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (2018).

[44] WHITEHOUSE, S. The gfs2 filesystem. In *Proceedings of the Linux Symposium* (2007), Citeseer, pp. 253–259.

[45] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)* (2016).

[46] YANG, J., WEI, Q., CHEN, C., WANG, C., AND YONG, K. L. NV-Tree: reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)* (2015).

[47] ZHAO, J., LI, S., YOON, D. H., XIE, Y., AND JOUPPI, N. P. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2013), pp. 421–432.

[48] ZUO, P., AND HUA, Y. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of the 33st International Conference on Massive Storage Systems and Technology (MSST)* (2017).

[49] ZUO, P., HUA, Y., AND WU, J. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, 2018).

Software Wear Management for Persistent Memories

Vaibhav Gogte[1], William Wang[2], Stephan Diestelhorst[2], Aasheesh Kolli[3,4],
Peter M. Chen[1], Satish Narayanasamy[1], and Thomas F. Wenisch[1]

[1]University of Michigan
[2]ARM
[3]Pennsylvania State University
[4]VMware Research

Abstract

The commercial release of byte-addressable persistent memories (PMs) is imminent. Unfortunately, these devices suffer from limited write endurance—without any wear management, PM lifetime might be as low as 1.1 months. Existing wear-management techniques introduce an additional indirection layer to remap memory across physical frames and require hardware support to track fine-grain wear. These mechanisms incur storage overhead and increase access latency and energy consumption.

We present *Kevlar*, an OS-based wear-management technique for PM that requires no new hardware. Kevlar uses existing virtual memory mechanisms to remap pages, enabling it to perform both *wear leveling*—shuffling pages in PM to even wear; and *wear reduction*—transparently migrating heavily written pages to DRAM. Crucially, Kevlar avoids the need for hardware support to track wear at fine grain. Instead, it relies on a novel *wear-estimation* technique that builds upon Intel's Precise Event Based Sampling to approximately track processor cache contents via a software-maintained Bloom filter and estimate write-back rates at fine grain. We implement Kevlar in Linux and demonstrate that it achieves lifetime improvement of 18.4× (avg.) over no wear management while incurring 1.2% performance overhead.

1 Introduction

Forthcoming Persistent Memory (PM) technologies, such as 3D XPoint [3, 46], promise to revolutionize storage hierarchies. These technologies are appealing in many ways. For example, they are being considered as cheaper, higher capacity and/or energy-efficient replacements for DRAM [5, 64, 87, 119], low-latency and byte-addressable persistent storage [22, 23, 83, 101], and even as hardware accelerators for neural networks [89, 94]. We focus on systems with heterogeneous memory—with both DRAM and PM connected to the memory bus. Such systems may use PM for persistent data storage or to replace some or all of DRAM with a cheaper/higher-capacity technology.

Nevertheless, PM's limited write endurance [21, 64, 87, 114, 119] may hinder adoption. Just like erase operations wear out Flash cells, PM devices may also wear out after a certain number of writes. The expected PM cell write endurance varies significantly across technologies. For example, a phase-change memory is expected to endure $10^7 - 10^9$ writes [64, 85, 87] while resistive RAM may sustain over 10^{10} writes [106]. So, system developers must consider PM cell write frequency and manage wear to ensure memory endures for the expected system lifetime.

PM wear-management techniques employ *wear leveling*, spreading writes uniformly over all memory locations, and/or *wear reduction*, reducing the number of writes with additional caching layers [26, 64, 85, 88, 92, 119]. Unfortunately, prior techniques rely on various kinds of hardware support. Some proposals [85, 119] add an additional programmer-transparent address translation mechanism in the PM memory controller. These mechanisms periodically remap memory locations to uniformly distribute writes across the PM. Other techniques [26, 88, 114] perform wear reduction by remapping contents of frequently-written PM page frames to higher-endurance DRAM. Such techniques depend on hardware support to estimate wear, for example, via per-page counters or specialized priority queues/monitoring in the memory controller. Unfortunately, PM-based mechanisms [26, 88, 114] that rely on higher-endurance but volatile DRAM to reduce wear do not support applications [77] that require crash consistency when using PM as storage.

The indirection mechanisms proposed for PMs are analogous to the translation layer [33, 58, 65] in Flash firmware, which perform functionalities such as garbage collection [33, 58, 109] and out-of-place updates [33, 58, 65, 67] in addition to wear leveling, and incur high erasure latency [33, 53, 67]. Additional translation layers increase design complexity and incur higher access latency and power/energy consumption. Indeed, recent work [12, 15, 40, 41, 50, 66, 82, 115] aims to eliminate complexity and overhead associated with a Flash translation layer by combining its features in either the virtual

memory system in the OS [12, 15, 40, 41, 115], or in file-system applications [15, 50, 66, 82]. We would prefer to avoid additional indirection mechanisms for byte-addressable PMs, which have lower access latency and offer a direct load/store interface.

We note that the OS already maintains a mapping of virtual to physical memory locations and that these mappings can be periodically updated to implement wear management without an additional translation layer. We build upon virtual memory to implement *Kevlar*, a software wear-management system for fast, byte-addressable persistent memories. Kevlar performs both wear leveling, by reshuffling pages among physical PM frames, and wear reduction, by judicious migration of wear-heavy pages to DRAM, to achieve a configurable lifetime target.

A critical aspect of wear management is to estimate the wear to each memory location. Existing hardware tracks PM writes only at the granularity of memory channels—too coarse to be useful for wear management. Tracking PM writes at finer granularity is complicated by write-back hardware caches; an update to a memory location leads to a PM write only when a dirty cache block is evicted from the processor's caches.

Kevlar relies upon a novel, low-overhead *wear-estimation* mechanism by using Intel's Precise Events Based Sampling (PEBS) [44], which allows us to intercept a sample of store operations. Kevlar maintains an approximate representation of hardware cache contents using Bloom filters [16], and uses it to estimate relative fine-grain writeback rates. We demonstrate that our estimation strategy incurs less than 1% performance overhead.

Kevlar enables wear management for applications that employ PMs for capacity expansion [5, 55, 88] and/or durability [77]. When a PM device is used for capacity expansion, Kevlar exploits memory device heterogeneity and migrates frequently updated PM pages to the neighboring DRAM—a system-level option that cannot be exploited by device-level wear-management schemes [85, 92, 119]. We show that migrating as few as 1% of pages from PM to DRAM is sufficient to achieve our target PM lifetime. For pages that require durability, Kevlar relies on reserve PM capacity and performs directed migrations of frequently written pages across the nominal and reserve capacity.

We implement Kevlar in Linux version 4.5.0 and evaluate its impact on performance and PM lifetime. To summarize, the contributions of Kevlar are:

- *Wear leveling:* We first develop an analytical framework to show that even a simple, wear-oblivious random page shuffling is sufficient to achieve near-ideal (uniform) wear over the memory device lifetime at negligible ($< 0.1\%$) performance overhead. Unfortunately, even ideal wear leveling provides insufficient lifetime for lower-endurance PMs.

- *Wear estimation:* We demonstrate how to estimate wear at fine grain by using Intel's PEBS to approximate cache contents via a Bloom filter, thereby estimating the cache write-backs to each page. We show that this mechanism is 21.7× more accurate than naive write sampling.

- *Wear reduction:* We demonstrate Kevlar, which uses our wear-estimation technique to apply both wear leveling and wear reduction, reducing wear by migrating less than 1% of the application working set to neighboring DRAM (when durability is not needed) incurring 1.2% (avg.) performance overhead.

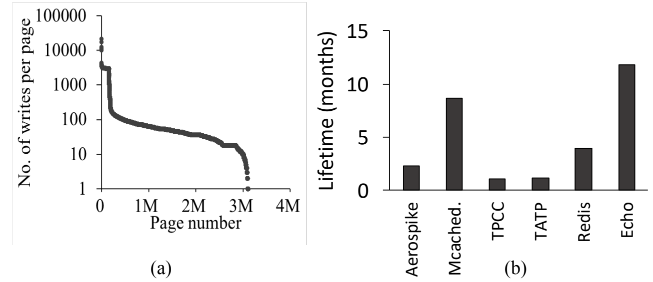


Figure 1: (a) **Pages sorted by number of writes (program entirety) in Aerospike**: There is a large disparity between most and least written pages. (b) **PM lifetime with no wear leveling**: The lifetime until 1% of pages sustain 10^7 writes can be as short as 1.1 months.

2 Background and Motivation

We briefly describe PM use cases and their drawbacks.

2.1 Persistent Memories (PMs)

Persistent memory technologies, such as Phase Change Memory [64, 87], Memristor [106], and Spin Torque Transfer RAM [111] are byte-addressable, achieve near-DRAM performance, and are denser and cheaper than DRAM. These characteristics allow systems to leverage PMs in exciting new ways. We focus on two well-studied use cases: (1) capacity expansion and (2) memory persistency.

Capacity expansion: Owing to their higher density and lower power consumption, PMs are projected to be cheaper than DRAM [5, 31, 55, 64, 87, 119] on a dollar per GB basis. Higher density enables greater peak capacity: Intel expects to soon offer servers with up to *6TB* of PM [3, 38]. System designers can use this capacity to manage larger in-memory data-structures [9, 42, 72].

Memory persistency: Since PMs are non-volatile, they blur the traditional distinctions between memory and storage. Recent research leverages PM non-volatility by accessing persistent data directly in memory via loads and stores [22, 23, 28, 36, 48, 52, 60, 61, 63, 77, 83, 101]. The byte-addressable load-store PM interface enables fined-grained accesses to persistent data and avoids the expensive serialization and de-serialization layer of conventional storage [54].

PM drawbacks: Whereas PMs exhibit many useful properties, they also have two key drawbacks. First, PM cells have limited write endurance. For example, PCM endures only 10^7 - 10^9 writes [85]. In contrast, DRAM endurance is essentially unbounded (> 10^{15} writes) [87]. Limited PM endurance may lead to rapid capacity loss for write-intensive applications. Figure 1(a) shows the disparity between writes seen by the hottest and coldest pages for Aerospike (see Section 5 for our methodology). Absent wear management, frequently written-back addresses wear out sooner, compromising lifetime. Figure 1(b) shows the lifetime until 1% of memory locations wear out in a device with a write endurance of 10^7 writes (such as PCM) under the write patterns of various applications assuming no efforts to manage wear. For example, we observe that TPCC can wear out a PCM memory device within 1.1 months.

Second, PM access latency and bandwidth, while close to DRAM, fall short [64, 87, 106]. So, applications sensitive to memory performance might still prefer DRAM. Prior works [5, 55, 84] mitigate this challenge by identifying hot/cold regions of applications' footprints and placing hot regions in DRAM and cold regions in PM. Unlike these works [5, 55, 84], we exploit memory device heterogeneity to improve device lifetime when PMs are employed for capacity expansion and/or memory persistency. To this end, we propose Kevlar, a wear-management mechanism to improve low-endurance PM device lifetime.

2.2 Wear-aware virtual memory system

Prior PM wear-management mechanisms [85–87, 92, 119] require an additional indirection layer in hardware to uniformly wear PM cells. However, these mechanisms suffer from several drawbacks. First, these mechanisms [85–87, 92] use volatile DRAM caches to reduce wear to PM. These mechanisms do not readily support applications [77] that rely on PM durability, since the volatile DRAM caches lose data upon power failure. Second, these mechanisms perform additional DRAM cache lookups and address translation for each memory access, delaying PM loads/stores. Third, wear leveling alone sometimes achieves PM lifetime of only 2.3 years (as shown later in Section 6.2)—lower than the desired system lifetimes. These device-level mechanisms are unable to exploit memory system heterogeneity for applications that employ PMs for capacity expansion.

We explore low-overhead OS wear-management mechanisms that can extend PM device lifetime to a desired target without any additional indirection layers. Indeed, our approach is analogous to similar ongoing efforts [12, 15, 40, 41, 50, 66, 82, 115] in Flash-based systems to identify and eliminate performance bottlenecks in the Flash translation layer (FTL). These works avoid FTL complexities and overheads by folding its features either into the virtual memory system [12, 15, 40, 41, 115], or into file system applications [15, 50, 66, 82]. Like these works, we aim to build PM wear-management into the virtual memory system. Note that, contrary to block-based access to Flash, PM updates arise from LLC write-backs. Unfortunately, there are no straightforward mechanisms to measure LLC write-backs directly at fine grain—a critical challenge that we solve in Kevlar.

3 Kevlar

We detail wear-management approaches in Kevlar.

3.1 Wear leveling

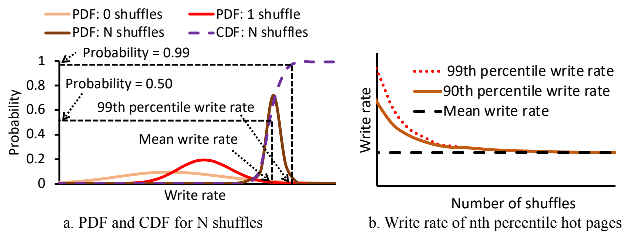
Modern OSes, such as Linux, manage memory via a paging mechanism to translate virtual to physical memory addresses. Linux manages the page tables used by the hardware translation mechanism, and already reassigns virtual-to-physical mappings for a variety of reasons (e.g., to improve NUMA locality).

Kevlar's *Wear-Leveling* (WL) mechanism uses existing OS support to periodically remap virtual pages to spread writes uniformly. Kevlar makes a conservative assumption that a write to a physical PM page modifies all locations within that page. Thus, Kevlar does not need an additional intra-page wear-leveling mechanism. We observe that periodic random shuffling of virtual-to-physical mappings—migrating each virtual page to a randomly selected physical page frame—is sufficient to uniformly distribute writes to PM provided shuffles are frequent enough. A key advantage of this approach is that it is wear oblivious—it requires no information about the wear to each location; it only requires the aggregate write-back rate to memory, which is easily measurable on modern hardware. Surprisingly, we find that this simple approach may be acceptable for PM devices with a sufficiently high endurance (e.g., 10^9 writes).

We consider a scheme that periodically performs a random *shuffle* of all virtual pages, reassigning each virtual page to a randomly selected physical page. Whereas our analysis assumes all pages are shuffled at once for simplicity, in practice, pages are shuffled continuously and incrementally over the course of the shuffle period. Our analysis poses the question: How many times must the address space be shuffled for the expected number of writes to each page to approach uniformity? Furthermore, at what point does the wear incurred by shuffling exceed the wear from the application? To simplify discussion, we use "write" to mean write-back from the last-level cache to the PM throughout this section.

Analysis. Let W represent the write distribution to physical pages and W_i be the write rate to i^{th} physical page in the memory. We define an equality function E as:

$$E(x,y) = \begin{cases} 1 & x == y \\ 0 & x! = y \end{cases} \quad (1)$$



a. PDF and CDF for N shuffles b. Write rate of nth percentile hot pages

Figure 2: **(a) Write-back rate distribution**: We use an application's write distribution to derive 99th percentile write rate after N shuffles. **(b) Write-back rate vs. shuffles**: The disparity in page write rates shrinks with the increase in shuffles.

Given a write distribution W over n physical pages, P_n^k represents the probability density function (PDF) for W after k shuffles. Using the distribution W, we can compute the probability $P_n^0(x)$ of physical page with the write rate x with 0 shuffles (initial state) as:

$$P_n^0(x) = \frac{1}{n} \times \sum_{i=1}^{n} E(W_i, x) \tag{2}$$

With no shuffles, one can easily compute the expected life of each physical page by dividing the expected endurance (in number of writes) by the write rate x, yielding an expected lifetime distribution over pages. When we consider a shuffle's effect, each page will experience an average write rate x' of two write rates x_1 and x_2 chosen uniformly at random from W. Since the PDF of the sum of two random variables is the convolution of their respective PDFs, we can calculate the expected distribution of write rates after S shuffles, P_n^S, as:

$$P_n^S(X = x'/2) = \sum_{k=-\infty}^{\infty} P_n^{S-1}(X = k) P_n^{S-1}(X = x' - k) \tag{3}$$

Note the normalization by one half, since we want the average (rather than the sum) of the random variables.

We illustrate the PDF P_n^0 (expected write rate without shuffles) of the page write distribution as expressed by Eq. 2 in Fig. 2 (a). The PDF P_n^0 has a heavy right-tailed distribution with high variance (*i.e.* the write-rate of few pages is high as compared to the mean write rate), a characteristic typical of the applications we have studied. Moreover, due to high variance, there is a wide write-rate range that might occur for any given page. Next, we compute the PDF P_n^S using Eq. 3 for shuffles ranging from one to N. With each shuffle, the PDF variance shrinks, while the probability of a near-mean write rate increases. Note that the PDF mean P_n^1 *appears* to be higher than the PDF P_n^0 due to the heavy right-tail of P_n^0. The mean in fact stays constant after each shuffle.

Fig. 2 (a) illustrates how the PDF after N shuffles converges to the mean write rate (equivalently, writes become uniformly distributed over the physical pages). In Figure 2 (a), we also show the cumulative distribution function (CDF) for N shuffles where the CDF C_n^N is used to compute the top n^{th} percentile of pages with the highest write rate after N shuffles (i.e., the "hottest" pages). $C_n^N(p)$ provides the minimum expected write rate of the most heavily written $(1-p)*100\%$ of the pages. For example, in Fig. 2 (a), we mark with a dotted line the 99th percentile. The $C_n^N(p = 0.99)$ gives the minimum expected write rate of the most heavily written 1% of pages after N shuffles. From this rate, we can estimate when we expect this 1% of pages to have worn out. As the number of shuffles grows, the variance shrinks and $C_n^N(p = 0.99)$ approaches the mean write rate.

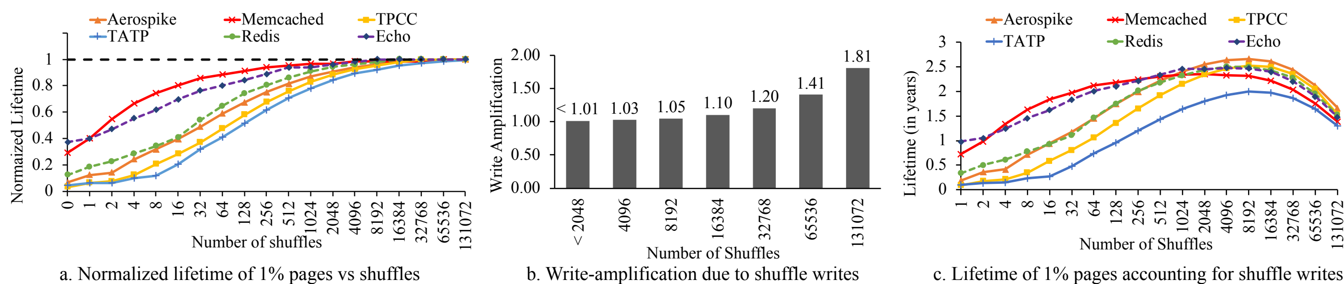
We illustrate how the write rate of the hottest pages compares to the mean as a function of the number of shuffles in Fig. 2 (b). Note that our approach can estimate the wear rate at any percentile, but we present results primarily for the 99th percentile. Without shuffles, there is a large disparity between the most-written 1% of pages and the mean. The gap rapidly shrinks with additional shuffles. Given the hottest pages' write rates in Fig. 2(b), we compute lifetime of a device with a 10^7 write endurance.

Tracing Methodology. We collect write-back traces for a set of applications (detailed in Section 5) using the DynamoRio [17] instrumentation tool and its online cache simulation client `drcachesim`. Since `drcachesim` can simulate only a two-level cache hierarchy with power-of-two cache sizes, we model an 8-way 256KB L2 cache and 32MB 16-way associative L3 cache, which is close to the configuration of the physical system on which we evaluate our Kevlar prototype (described in Table 1). We instrument loads and stores to trace all memory references and run `drcachesim` online to simulate the system's cache hierarchy. We record writebacks from the simulated LLC to PM. We then extract write rate distributions to analyze expected PM lifetime under shuffling.

Determining optimal shuffles. In Fig. 3(a), we show the lifetime, normalized to what is possible under ideal wear leveling, as a function of the number of shuffles. We assume some redundancy in the PM device similar to prior works [85, 86] and define its lifetime as the time when 1% of pages are expected to fail. Note that the lifetime under ideal wear leveling is the device endurance divided by the application's average write-back rate. As shown in Figure 3(a), frequently written virtual pages are mapped to a different set of physical pages after every shuffle, leading to improved device lifetime with more shuffles. Interestingly, for all applications, after about 8192 shuffles, the expected lifetime converges to that of ideal wear leveling (i.e., the write distribution is uniform). Note that we do not consider the additional writes incurred due to remapping virtual-to-physical page mappings after each shuffle in Figure 3(a).

Figure 3(b) shows the write amplification caused due to the shuffle operations. The write amplification shows the ratio of the total writes incurred after shuffling as compared to the application's PM writes. The write amplification can be higher than 1.4x (40% additional writes) for greater than 2^{16} shuffles as shown in Figure 3(b).

Peak lifetimes occur when memory is shuffled 8192 times over the device lifetime. With 8192 shuffles, we perform 5% additional writes for wear leveling. Fig. 3(c) shows the writes



a. Normalized lifetime of 1% pages vs shuffles

b. Write-amplification due to shuffle writes

c. Lifetime of 1% pages accounting for shuffle writes

Figure 3: **(a) Lifetime of 1% of pages vs. shuffles**: The expected lifetime converges to the ideal lifetime for shuffles > 8192, **(b) Write-amplification due to shuffle writes**: Kevlar performs 5% additional writes with 8192 shuffles, **(c) Lifetime of 1% of pages, accounting for shuffle writes**: The lifetime of PM peaks at 8192 shuffles, following which shuffle writes become significant.

due to shuffle operations, which may grow to dwarf the application's writes if shuffles are too frequent (*i.e.* >16384).

Discussion. Shuffling memory 8192 times over the PM device lifetime uniformly distributes PM writes. However, the lifetime achievable via even ideal wear leveling is limited by an application's average write rate. For our applications, this lifetime is only 2.3 to 2.8 years for a device that wears out after 10^7 writes (see Fig. 3(c)). Wear leveling alone may be insufficient to meet lifetime targets.

To achieve desired lifetimes, we must augment Kevlar's wear-leveling mechanism with a wear-reducing mechanism. The key challenge for wear reduction is to monitor the wear to each virtual page at low overhead. There is no straightforward mechanism for the OS to directly monitor device wear at fine granularity. PM devices incur wear only when writes reach the device. Write-back caches absorb much of the processor write traffic, so the number of stores to a location can be a poor indicator of actual device wear. Current x86 hardware can count writebacks per memory channel, but provides no support for finer-grain (e.g., page or cache line) monitoring. Mechanisms that monitor writes via protection faults (e.g., [5, 34]) incur high performance overhead and fail to account for wear reduction by writeback caches, grossly overestimating wear for well-cached locations. Instead, Kevlar builds a software mechanism to estimate per-page wear intensity.

3.2 Wear Estimation

We design a wear-estimation mechanism that approximately tracks hardware cache contents to estimate per-page PM write-back rates. Our mechanism builds upon Intel's PEBS performance counters [45] to sample store operations executed by the processor. Note that, although we focus on Intel platforms, other platforms—AMD Instruction Based Sampling [29] and ARM Coresight Trace Buffers [7]—provide analogous monitoring mechanisms. Kevlar's write estimation mechanism monitors the retiring stores to maintain an estimate of hardware cache contents.

Monitoring stores. PEBS captures a snapshot of processor state upon certain configurable events. We configure PEBS to monitor `MEM_UOPS_RETIRED.ALL_STORES` events. As stores retire, PEBS can trigger an interrupt to record state into a software-accessible buffer; we record the virtual address accessed by the retiring store.

Although accurate, sampling every store with PEBS is prohibitive. Instead, we rely on systematic sampling to reduce performance overhead: we configure PEBS with a *Sample After Value* (SAV). For a SAV of n, PEBS captures only every n^{th} event. Like prior work [71], we choose prime SAVs to avoid bias from periodicities in the systematic sampling. We explore the accuracy and overhead of SAV alternatives in Section 6.1.

We obtain the virtual addresses of sampled stores to estimate per-page write-back rates. A naive strategy to compute write-back rates is to assume that each sampled store results in a write-back. However, with write-back hardware caches, a PM write occurs only when a dirty block is evicted from the cache hierarchy; many stores coalesce in the caches. Indeed, in our applications, the naive strategy drastically overestimates writebacks (see Section 6.1). Consequently, we design an efficient software mechanism that estimates temporal locality due to hardware caches to predict which stores incur write-backs.

Estimating temporal locality. Prior mechanisms have been proposed to estimate temporal locality in storage [102, 103] or multicore [13, 90, 91] caches. These mechanisms maintain stacks or hashmaps to compute reuse distances for accesses to sampled locations. Instead, we focus on modeling temporal locality in hardware caches to estimate LLC write-backs using sampled stores. We estimate temporal locality by using a Bloom filter [16] to approximately track dirty memory locations stored in the caches. For each store sampled by PEBS, we insert its cache block address into the Bloom filter. (Algorithm 1: Line 12-14). Whenever a new address is added to the filter, we assume it is the store that dirties the cache block, and hence will eventually result in a writeback. Further stores to the same cache block will find their address already present in the Bloom filter; we assume these hit in the cache and hence do not produce additional write-backs. Thus, the Bloom filter maintains a compact representation of likely dirty blocks present in the cache.

Bloom filters have a limited capacity; after a certain num-

ber of insertions into the set, their false positive rate increases rapidly. We size the Bloom filter such that it can accurately (less than 1% false positives) track a set as large as the capacity of the processor's last-level cache (LLC), which is roughly 700K cache blocks on our evaluation platform. We clear the Bloom filter when the number of insertions reaches this size (Algorithm 1: Line 19-29).

Of course, after clearing the filter, Kevlar would predict a sudden false spike in writeback rates. We address this by using two Bloom filters; Kevlar probes both filters but inserts into only one "active" filter at a time (Algorithm 1: Line 3, 12-17). When the active filter becomes full, we clear the inactive filter and then make it active. As such, at steady state, one filter contains 700K cache block addresses, while the other is active and being populated (Algorithm 1: Line 12-17). We assume a cache block will result in a store hit (no additional writeback) if it is present in either filter (Algorithm 1: Line 6-10).

In essence, our tracking strategy filters out cache blocks that have write reuse distances [56] of about 700K or less, as such writes are likely to be cache hits. Effectively, we assume that dirty blocks are flushed from the cache primarily due to capacity misses, which is typically the case for large associative LLCs [39, 113]. Note that our estimate of the cache contents is approximate. For example, the Bloom filters do not track read-only cache blocks. Moreover, due to SAV, only a sample of writes are inserted. The mechanism works despite these approximations because: (1) frequently written addresses are likely to be sampled and inserted into the filters—it is these addresses that are most critical to track; and (2) few addresses have reuse distances near 700K—reuse distances are typically much shorter or longer, so the filters are effective in estimating whether or not a store is likely to hit. Although Kevlar approximates writebacks by sampling retiring stores, our goal in Kevlar is to measure relative hotness of the pages as opposed to absolute writebacks per page. We show the accuracy of our estimation mechanism to identify writeback intensive pages later in Section 6.1.

Estimating write-backs. PEBS provides the virtual address of sampled stores. Our handler then walks the software page table to obtain the corresponding physical frame (Alg. 1: Line 7). In our Linux prototype, we maintain a writeback count in `struct page`, a data-structure associated with each page frame. When we sample a store, we update the counter for the corresponding physical page as shown in Alg. 1: Line 8. Kevlar uses the estimated writebacks to identify writeback-intensive pages.

3.3 Wear Reduction

As shown in Sec. 3.1, Kevlar's wear-leveling mechanism can achieve only 2.3- to 2.8-year lifetime for a PM device that wears out after 10^7 writes. Our goal is to achieve a lifetime target for a low-endurance PM device by migrating heavily written pages to DRAM. We assume a nominal lifetime goal of four years. This target is software-configurable; we discuss longer targets in Section 6.2.

Consider an application with a memory footprint of N physical PM pages and a given lifetime target, the write rate to the PM B writes/sec to achieve the lifetime target can be computed as:

$$B = \frac{Endurance \times N}{Lifetime} \tag{4}$$

We use Eq. 4 to compute the number of writes the application may make per 1GB (*i.e.* N = 256K small pages) of PM footprint. For a given lower-bound endurance of 10^7 writes and a 4-year lifetime, writebacks must be limited to 20K writes/sec/GB. Configuring a different target lifetime or device endurance changes the allowable threshold.

One approach is to use wear leveling (as described in Sec. 3.1) by provisioning additional reserve capacity such that the target lifetime is met. This strategy is applicable both when PM is used for persistent storage or capacity expansion. For instance, with N pages in an application, and average write rate of B' writes/sec/GB, the reserve capacity R to achieve a 4-year lifetime is given by:

$$R = \frac{N \times B'}{2 \times 10^4} \tag{5}$$

When the application write rate is high relative to the device endurance, the required reserve can undermine any cost advantages, as we show later in Section 6.3. Instead, for capacity expansion, we propose wear reduction by migrating the hottest pages to high-endurance memory (DRAM). Kevlar regulates the average write rate to the pages that remain in PM to 20K writes/GB/sec such that we achieve the desired lifetime of four years.

3.3.1 Page migration

Kevlar uses its write-back estimation mechanism to measure per-page PM writeback rates and migrate the most write-intensive pages to DRAM. Kevlar must regulate average PM writeback rate to 20K writes/GB/sec to achieve a 4-year lifetime. Kevlar uses `IMC.MC_CHy_PCI_PMON_CTR` counters in the Intel memory controller to count `CAS_COUNT.WR` events, which measure write commands issued on the memory channels. Such counters already exist in DRAM controllers, and analogous counters exist on other hardware platforms (*e.g.* ARM's L3D_CACHE_WB performance monitoring unit counter [8]). This aggregate measure allows us to determine whether pages must be migrated from PM to DRAM (or can be migrated back) to maintain the target average rate of 20K writes/GB/sec.

Migrating hot-pages to DRAM. Kevlar computes the PM writeback rate at a fixed 10-second interval. If the average writeback rate exceeds 20K writes/GB/sec during an interval, Kevlar enables PEBS and samples the retiring stores as explained in Section 3.2. Kevlar estimates the PM writeback rate

Algorithm 1 Write-back estimation mechanism

```
1: Inputs:
       PEBS record rec, Bloom Filter filterA, Bloom Filter filterB
2:
3: Initialize:
       filterA.isActive = True
       filterB.isActive = False
       activate = LLC_CACHE_BLOCKS
4:
5: blockAddr = rec.strAddr » log2(LLC_BLOCK_SIZE)
6: if !filterA.isPresent(blockAddr) and !filterB.isPresent(blockAddr) then
7:     pageStruct = doPageWalk(blockAddr)
8:     pageStruct.WBCount+=1
9:     memRef+=1
10: end if
11:
12: if filterA.isActive and !filterA.isPresent(blockAddr) then
13:     filterA.add(blockAddr)
14: end if
15: if filterB.isActive and !filterB.isPresent(blockAddr) then
16:     filterB.add(blockAddr)
17: end if
18:
19: if activate == memRef then
20:     filterA.isActive = !filterA.isActive
21:     filterB.isActive = !filterB.isActive
22:     if filterA.isActive then
23:         filterA.clear()
24:     end if
25:     if filterB.isActive then
26:         filterB.clear()
27:     end if
28:     activate+=LLC_CACHE_BLOCKS
29: end if
```

at 4KB-page granularity. When migration is needed, Kevlar scans writeback counters for all page frames and sorts them by their estimated write-back counts. Kevlar then migrates the hottest 10% of pages to DRAM. It continues monitoring for an additional interval. Kevlar ceases migration, disables PEBS monitoring, and clears write-back counters when the write-back rate falls below 20K writes/GB/sec. With this monitoring and migration control loop, Kevlar achieves our lifetime target with 1.2% performance impact.

Migrating cold pages to PM. An application's access pattern might change over its execution, so pages migrated to DRAM may become cold. To minimize the application footprint in DRAM, it is desirable to migrate cold pages back to PM. If Kevlar observes five consecutive intervals with a PM writeback rate below 20K writes/GB/sec, it re-enables PEBS for a 10-second interval, estimates the write-back rate of pages in DRAM, and migrates 10% of cold pages from DRAM back to PM.

4 Implementation

We implement Kevlar in Linux kernel version 4.5.0. We use the Linux control group mechanism [74] to manage Kevlar specific configuration parameters.

Wear leveling. Kevlar should shuffle the entire application footprint once every 4.2 hours to achieve uniform wear leveling over a lifetime of 4 years. Instead of gang-scheduling the shuffle operations together every 4.2 hours, Kevlar periodically shuffles a fraction of application footprint. Kevlar maintains a shuffle bit in the `struct page` associated with each page frame to indicate whether the page was shuffled within the current shuffle interval. Kevlar scans the application pages every 300-sec *shuffle interval* to identify the pages that are yet to be shuffled. It randomly chooses a fraction of pages to be shuffled in this shuffle interval by equally apportioning the total number of pages yet to be shuffled to the time remaining in a 4.2 hour shuffle operation.

The fraction of pages are then shuffled following these steps: (1) Kevlar selects a pair of application pages in PM to be swapped. (2) It locks the page table entries for both pages so that any intermediate application accesses stall on page locks. (3) It allocates a temporary page in DRAM (for capacity workloads) to aid in swapping the contents of the two pages in PM. (4) Once the pages are swapped, Kevlar restores the page table entries so that the virtual addresses now map to the swapped pages, unlocks the pages, and deallocates the temporary DRAM page. (5) Once shuffled, Kevlar records this event in the shuffle bit in page frame's `struct page` of the two pages.

Note that, we use a temporary page mapped in DRAM to limit wear in PM due to shuffle. For persistent applications, we map the temporary page in PM to ensure that the page contents are persistent in case of intermediate failure. Once all the pages are shuffled, Kevlar clears the shuffle bit in `struct page` and initiates the next shuffle.

Wear estimation. Kevlar initializes PEBS to monitor the `MEM_UOPS_RETIRED.ALL_STORES` event and a SAV to sample the retiring stores for wear estimation. We determine SAV empirically to ensure that the monitoring has negligible performance overhead. Kevlar implements two Bloom Filters, each of size 840KB and a capacity of 700K cache blocks, corresponding to the 45MB LLC of our system. We size the Bloom filter to achieve less than 1% false positives. As explained in Section 3.3.1, Kevlar performs a software page table walk to identify the page frames being accessed by the sampled store, and records writeback counts in `struct page`.

Wear reduction. Kevlar monitors PM writeback rate at a 10-second *migration interval* to determine if it needs to initiate hot/cold page migration between DRAM and PM. If the PM writeback rate triggers a migration, Kevlar scans the application pages and identifies the top 10% hot (or cold) pages to be migrated to DRAM (or PM). It performs migration using a mechanism similar to the page shuffles in wear leveling: it locks the page to be migrated, copies its contents to a newly allocated page in DRAM (or PM), updates page table entries, and unlocks the page. If no migration is triggered, Kevlar disables PEBS sampling counters to minimize performance monitoring overhead.

5 Methodology

We next discuss details of our prototype and evaluation.

Core	Intel Xeon E5-2699 v3, 2.30GHz 36-core (72 hardware threads) Dual-socket x86 server
L1 D&I Cache	32KB, 8-way associative
L2 Cache	256KB, 8-way associative
Shared LLC	45MB, 20-way associative
DRAM	256GB per socket
Operating System	Linux Kernel 4.5.0

Table 1: **System Configuration.**

5.1 Emulating persistent memory

A system with byte-addressable persistent memory is not yet commercially available. Hence, we emulate a hybrid PM-DRAM memory system using a dual-socket server. We run the application under test on a single socket and treat memory local to that socket as DRAM. Conversely, we treat memory of the remote socket as PM. Note that the local and remote nodes are cache coherent across the sockets. Since each chip has its own memory controllers, we use the performance counters in each memory controller to monitor the total accesses to each device and distinguish "PM" and "DRAM" accesses.

Using this emulation, our Kevlar prototype incurs the actual performance overheads of monitoring and migration that would occur in a real hybrid-memory system. However, the latency and bandwidth differential between our emulated "PM" and "DRAM" is only the gap between local and remote socket accesses. The performance differential between DRAM and actual PM devices is technology dependent and remains unclear, but is likely higher than in our prototype. We expect relative performance overhead of our mechanism (as detailed later in Section 6.4) to be lower on a system with a high differential between DRAM and PM devices. Our results represent a high estimate of the Kevlar's performance overhead.

Nevertheless, our contributions with respect to wear management are orthogonal to the performance aspects of replacing DRAM with PM, which have been studied in prior work [5, 55, 84]. We focus our evaluation on quantifying the effectiveness and overheads of Kevlar's mechanisms.

5.2 System configuration

We run our experiments on a dual-socket server with the configuration listed in Table 1. We use the Linux control group mechanism [74] to isolate the application to a particular socket. We pin application threads to execute only on CPUs on the local node, but map all memory to initially allocate in the remote node using Linux's `memory` and `cpuset cgroups`, modeling a system where DRAM has been replaced by PM. Kevlar expects a lifetime goal for the PM device as an input, and performs wear leveling, estimation, and reduction for all the processes in the `cgroups`. The test applications use all 18 CPU cores of the local node with hyper-threading enabled. For client-server benchmarks, we run clients on another system to avoid performance interference.

As explained in Section 3.2, we use Intel's PEBS counters to estimate PM page writeback frequency. We isolate these counters to monitor only accesses from the application under test using Linux's `perf_event` cgroup mechanism. Thus, spurious store operations from background processes or the kernel do not perturb our measurements.

We measure the write rate to the PM (*i.e.* remote DRAM) using the performance counters in the memory controller. Unlike PEBS counters, these counters lie in a shared domain and cannot be isolated to count only events for a particular process. However, we have measured the write rate of the background processes in an idle system and find that they constitute less than 1% of the total writeback rate observed during our experiments.

5.3 Benchmarks

We study two categories of applications. We report memory footprints of the benchmarks under study in Figure 9.

5.3.1 Capacity Expansion Workloads

We evaluate both the wear-leveling and wear-reduction mechanisms of Kevlar for the following benchmarks in a "capacity expansion" PM use case.

NoSQL applications. Aerospike [1, 97], and Memcached [4] are popular in-memory NoSQL databases. We use YCSB clients [24] to generate the workload to Aerospike and Memcached. We evaluate 400M operations on 4M keys for Aerospike and 100M operations on 1M keys for Memcached. We configure each record to have 20 fields resulting in a data size of 2KB per record. As we are interested in managing wear in write-intensive scenarios, we configure YCSB for update-heavy workload with a 50:50 read-write ratio and Zipfian key distribution.

MySQL. MySQL is a SQL database management system. We drive MySQL using the open-source TPCC [98] and TATP [79] workloads from oltpbench [27]. TPCC models an order fulfillment business and TATP models a mobile carrier database. In each, we run default transactions with a scale-factor of 320 for 1800 secs.

5.3.2 Persistent Workloads

We evaluate persistent applications from the WHISPER benchmark suite [77], which use the Intel PMDK libraries [2] for persistence. These applications divide their address space into volatile and persistent subsets. The persistent subset must always be mapped to PM to ensure recoverability in the event of power failure. As such, Kevlar may not migrate pages in the persistent subset to DRAM. We instead rely only on wear leveling to shuffle these pages in PM. However, we allow pages

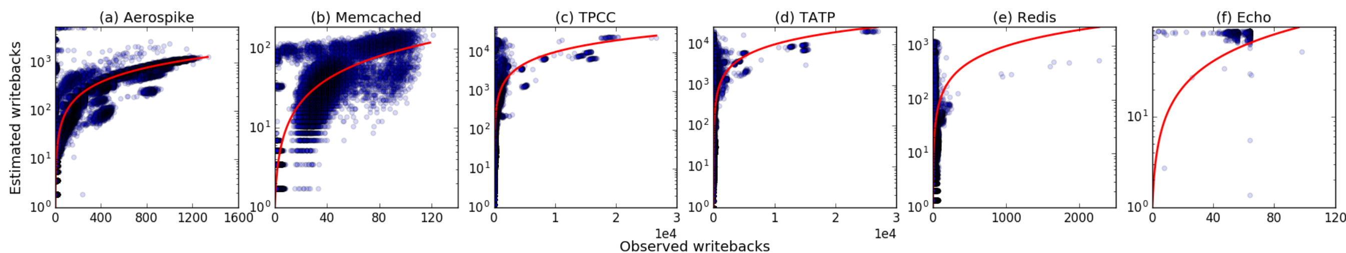


Figure 4: **Estimated writebacks vs. observed writebacks**. We compare the estimated writebacks with observed writebacks obtained from memory access tracing. Each point on the scatter plot represents the number of writebacks to a page. The red line on each plot represents the ideal prediction curve.

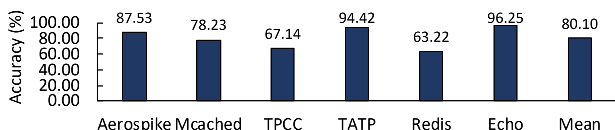


Figure 5: **Comparison of top 10% estimated hot pages to top 10% observed hot pages**. Kevlar's wear estimation identifies 80.10% (avg.) of the 10% hottest written pages correctly.

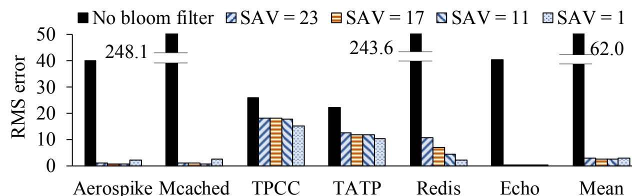


Figure 6: **RMS Error with cache modeling**. Kevlar achieves 20× lower RMS error than a mechanism without cache modeling.

in the volatile subset to migrate to DRAM if the aggregate write rate to all pages exceeds 20K writes/GB/sec.

Linux presently provides no mechanism to label pages as persistent or volatile. WHISPER benchmarks use Linux's `tmpfs` [96] memory mapped in DRAM to emulate persistency, and the persistent pages are allocated in a fixed address range. We hardcode this address range in our experiments to prevent page migrations to DRAM.

We select the two NoSQL applications, Redis and Echo, from WHISPER. Redis is a single-threaded in-memory key-value store. We configure a Redis database comprising 1M records, each with 10 fields. We use YCSB clients to perform key-value operations on the Redis server with a Zipfian distribution. For our evaluation, we run 40M operations with an update-heavy workload with a 50:50 read-to-write ratio. For echo, we use the configuration provided with the WHISPER benchmark suite and evaluate it using 2 client threads each running 40M operations.

6 Evaluation

We evaluate Kevlar's wear-management mechanisms.

6.1 Modeling Wear Estimation

We first evaluate the accuracy of Kevlar's wear-estimation mechanism as described in Section 3.2. We collect a ground-truth writeback trace for each application using the online cache simulator `drcachesim` in Dynamorio [17] with a tracing infrastructure described in Section 3.1. We model the PEBS sampling mechanism and bloom filters in `drcachesim` to record the estimated writeback rate. We compare the ground-truth writebacks against the estimates provided by the emulation of PEBS sampling and our Bloom filters.

Comparison with ideal mechanism. In Figure 4, we show estimated writebacks (vertical axis) and ground-truth observed writebacks (horizontal axis) for each application for one 10-sec sampling interval. We use log-linear scale[1] to highlight accuracy of our mechanism for higher write rate. As instrumentation results in application slowdown, we expand the 10-second sampling duration by the slowdown due to instrumentation measured for each workload. Due to the log-linear scale, we plot a red curve in the Figure to show the ideal prediction curve, where estimated and observed writebacks match. For all applications, Figure 4 (a-f) indicates that the estimated writebacks correlate closely to the ideal curve. Echo performs cache flush operation following each store to flush dirty cache blocks to PM. As a result, we observe 64 write-backs per page (owing to 64 cache blocks in a 4KB page) for nearly all pages. As shown in Figure 4(f), Kevlar is able to measure write-backs to these pages.

Prediction accuracy. Next, we compare the top 10% heavily written pages as estimated by Kevlar's wear-estimation mechanism to the top 10% hottest observed (ground-truth) pages. Figure 5 shows the percentage of heavily written pages correctly estimated by Kevlar. Kevlar correctly estimates 80.1% hottest pages on average and up to 96.3% hottest pages in Echo as compared to the ground truth.

We also demonstrate the accuracy of Kevlar's prediction mechanism by measuring root-mean-squared (RMS) error between estimated and observed writebacks. The RMS error reports the standard-deviation of the difference between estimated and observed writebacks. We study the impact of hardware cache modeling using our Bloom filter mechanism

[1] We use log-linear scale to highlight estimated and observed writebacks to hot pages that are crucial for our study. In contrast, a log-log scale discretizes lower writeback values and hides comparison between observed and estimated writebacks for hot pages.

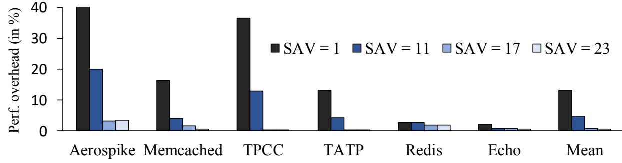


Figure 7: **PEBS sampling overhead**. Runtime overhead due to sampling every retiring store is 13.2% (avg.). We configure PEBS SAV = 17 in Kevlar with < 1% overhead.

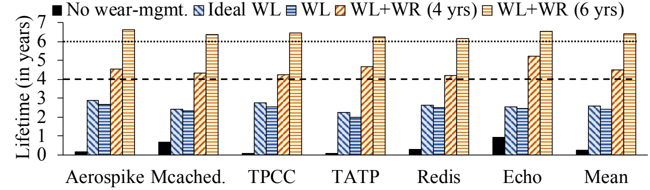


Figure 8: **PM Lifetime**: Kevlar achieves greater than 4 years of lifetime; 11.2× (avg.) higher than no wear leveling.

by comparing Kevlar's prediction mechanism with a mechanism without the Bloom filter. Figure 6 shows the RMS error of our writeback prediction mechanism normalized to the average writeback rate of the application for different PEBS SAV values. We choose prime numbers for PEBS SAV to avoid periodicities in systematic sampling.

As compared to a mechanism that does not model cache contents, we observe 100.0× and 106.8× improvement in RMS errors for Memcached and Redis, respectively, with our estimation mechanism (with SAV = 1). Overall, the Bloom filters can approximate the dirty cache contents well, allowing it to estimate writebacks with 21.6× lower RMS error on average. The Bloom filters are critical to avoiding overestimation of writebacks in Aerospike, Memcached, and Redis by estimating temporal locality of memory accesses. Note that, as shown in Figure 6, the standard deviation of the difference between absolute values of estimated and observed writebacks is 2.85× that of the mean for SAV of 1. Although the estimated writebacks are not accurate when compared to absolute values, our goal in Kevlar is to measure the relative hotness of the pages. As shown earlier in Figure 5, Kevlar identifies 80.1% of the 10% hottest pages correctly.

Configuring PEBS SAV. We study the RMS error in Figure 6 and runtime performance overhead in Figure 7 for different PEBS SAV values. Figure 7 shows the monitoring overhead for different SAVs when compared to the application runtime without PEBS monitoring. Upon sampling a store, PEBS triggers an interrupt and records architectural state in a software buffer, which can lead to a performance overhead. Taking an interrupt on every retiring store results in substantial performance overhead. Indeed, with SAV=1, the performance overhead due to PEBS sampling can be as high as 112.9% (in Aerospike), and 13.2% on an average. In contrast, the performance overhead in persistent applications, Redis and Echo, is less than 3% as we sample only stores to volatile pages, which may be migrated between PM and DRAM. Interestingly, with SAV of 17, the average performance overhead due to sampling is less than 1% (avg.) with no substantial degradation in RMS error. As we do not see any substantial performance gains for SAV > 17, we configure PEBS to sample one in every 17 stores in Kevlar.

6.2 PM Lifetime

We study Kevlar for lifetime targets of four and six years. We compare Kevlar's wear-management mechanisms to a baseline with no wear leveling. We make a conservative assumption that a write to a physical page modifies all locations within that page for Kevlar's wear-management mechanisms. In contrast, we measure lifetime for the baseline via precise monitoring at cache-line granularity.

Wear leveling alone. We first consider lifetime for the PM device achieved by Kevlar's wear-leveling mechanism alone. As discussed in Section 3.3, to achieve a four- (or six-) year lifetime until 1% of locations wear out on a PM device that can sustain only 10^7 writes, the average write rate must be below 20,000 (or 13,333) writes/GB/second. Even after wear leveling, all of the applications we study incur a higher average write rate when their entire footprints reside in PM. We also show lifetime due to ideal wear leveling in Figure 8 when writes are uniformly remapped in PM. Although wear leveling substantially improves PM lifetimes over a baseline of no wear leveling, it falls short of achieving the four-year and six-year lifetime targets for all applications. As compared to the baseline with no wear leveling, Kevlar with only wear leveling achieves an average lifetime improvement of 9.8× with 31.7× improvement in lifetime for TPCC.

Wear leveling + wear reduction. Wear reduction can improve application lifetimes to meet our target while moving only a remarkably small fraction of the application footprint to DRAM. Kevlar in wear leveling + wear reducing mode aims to limit the write-back rate to the PM at 20K (or 13.3K) write/GB/second for four (or six) year lifetime target, by identifying the "hottest pages" that are being frequently written back and migrating them to DRAM.

Owing to the writeback rate limit imposed by Kevlar's wear-reducing mechanism, as indicated in Figure 8, the lifetime with wear leveling + wear reduction exceeds the configured target of four and six years for all applications. Kevlar's wear leveling + wear reduction mode (for a 6-year lifetime configuration) achieves the highest lifetime improvement of 80.7× for TPCC, with an average improvement of 26.1× when compared to no wear leveling.

High-endurance PMs: Absent wear-management mechanisms, a PM device that can sustain 10^8 writes would wear out within 9.8 months. Moreover, for PM devices with endurance 10^8 - 10^9, wear-leveling mechanism would be sufficient to

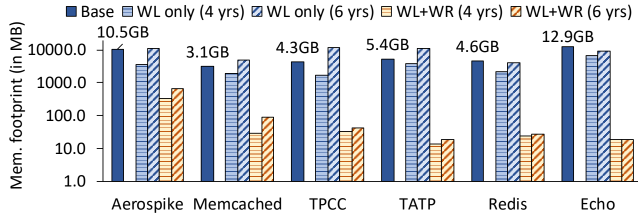


Figure 9: **Application footprint in PM and DRAM**: Kevlar migrates $< 1\%$ of application footprint to DRAM. Blue and orange bars represent application footprint in PM and DRAM respectively.

achieve the desired lifetimes of 4- and 6-years. For instance, our wear-leveling mechanism alone can achieve a lifetime of 24.0 years (average) for a PM device that can sustain 10^8 writes. Kevlar would not trigger wear-reduction mechanism for PMs with high write endurance as the application write-back rate would be lower than configured threshold. Nevertheless, the endurance numbers of commercial PM devices (i.e. Intel's 3D XPoint) are not publicly available. As such, we can configure the endurance of a PM device in Kevlar.

6.3 Memory Overhead

Figure 9 shows the baseline memory footprint of the applications, and an additional memory footprint in DRAM necessary to host the most frequently written PM pages that are migrated by Kevlar. In addition, we also show the reserve footprint that can be mapped in PM to achieve the lifetime targets using wear-leveling mechanism alone as outlined in Equation 5.

Wear reduction for persistency applications. For the WHISPER benchmarks that rely on persistency (Redis & Echo), the pages in the persistent set must always remain in PM. Nevertheless, some fraction of these applications' footprints are volatile and may reside in PM or DRAM. We initially map the entire footprint to the PM and allow only volatile pages to migrate to DRAM. As a majority of memory accesses are made to the volatile footprint in these applications [77], the wear-reducing mechanism can achieve a 4 year lifetime by migrating only 23.6MB of footprint to DRAM.

Reserve PM required can be significant. The amount of PM reserves required to ensure that the target lifetime be met are significant. It can be as high as 2.7× for TPCC and 2.0× for TATP for a six-year lifetime (1.3× average across all the benchmarks). The required reserve capacity may undermine the cost advantages of capacity expansion offered by PMs.

Reserve DRAM required is much smaller than reserve PM. As can be seen from Figure 9, the reserve DRAM required is much smaller than the reserve PM required. This difference is due to a difference in the write endurance of DRAM (practically infinite) and the cell endurance we assume for PM (10^7 writes). Note that Kevlar's goal is to limit wear while maximizing application footprint in PM (especially for the capacity expansion use-case) and achieve configured device lifetime. Thus, it migrates only the heavily written

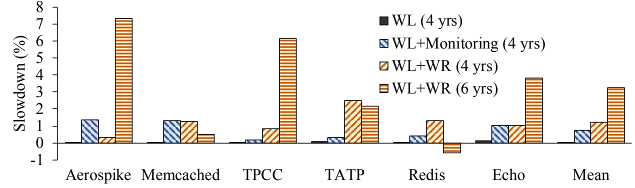


Figure 10: **Performance overhead**: Overhead of page monitoring and migration in Kevlar is 1.2% (avg.) in our applications.

application footprint from PM to DRAM. In contrast, prior mechanisms [5, 55] aggressively migrate pages to DRAM and limit application performance degradation resulting from slower PM accesses. Kevlar migrates less than 1% of the application's footprint to DRAM for four- and six-year lifetime targets, on average.

6.4 Performance Overhead

Next, we present application slowdown due to Kevlar.

Page shuffle overhead. Figure 10 illustrates the slowdown (lower is better) in applications resulting from our wear leveling, wear estimation, and page migration. The shuffle mechanism incurs a negligible average performance overhead of 0.04% (highest 0.1% in Echo) over the baseline with no wear leveling.

Overheads from Kevlar's monitoring and migration. As explained in Section 3.3, we configure PEBS with SAV of 17, and further reduce performance overhead by filtering store addresses using the Bloom filters. We observe up to 1.3% slowdown from our PEBS sampling in Aerospike, with even lower overheads in the remaining applications. Redis observes a net gain (as much as 0.9%) when we enable migration and relocate their frequently written pages to DRAM because the local NUMA node (representing DRAM) is faster than the remote node (representing PM) in our prototype. We expect the performance gains to be more pronounced with PMs that are anticipated to exhibit higher memory latency than remote DRAM in our prototype. On an average, we see 1.2% (or 3.2%) slowdown due to our wear-management mechanisms to achieve the lifetime goal of four (or six) years.

7 Related work

The adoption of PMs has been widely studied by both academia and industry in processor architectures [23, 28, 51, 52, 60, 77, 83, 95, 117], file systems [20, 23, 30, 100, 105, 107, 108], logging/databases [10, 11, 18, 19, 35–37, 59, 62, 63, 68, 73, 80, 81, 104], data structures [43, 78, 99], and distributed systems [57, 70, 116, 120]. We discuss the relevant works that address wear out problem in PMs.

7.1 Wear-reduction mechanisms

We first discuss techniques that reduce PM writes.

DRAM cache. Numerous works [32,75,87,92] advocate placing a DRAM cache in front of PM. The DRAM cache absorbs most of the writes thereby reducing wear. A DRAM cache presents three disadvantages: (1) it sacrifices capacity that could instead be used to expand memory; (2) it increases the latency of PM writes; and (3) it is inapplicable to writes that require persistency, which must write through the cache. Like many prior works [23,28,36,51,52,60,77,83,95,117], we assume that PM and DRAM are peers on the memory bus.

Page migration. Several works [6, 26, 88, 114] propose migrating pages from PM to DRAM to reduce wear. Dhiman et al. [26] use a software-hardware hybrid solution, where dedicated hardware counters (one per PM page) that track page hotness are maintained in PM and cached in the memory controller. RaPP [88] and Zhang et al. [114] use a set of queues in the memory controller to estimate write intensiveness and perform page migrations to DRAM. However, these mechanisms propose no wear-leveling solutions for the remaining pages in PM. As such, these mechanisms may still not achieve desired PM device lifetimes. For example, RaPP can achieve a device lifetimes exceeding 3 years only if the cell endurance exceeds 10^9 [88] – insufficient for PCM-based memories with endurance of only 10^7 - 10^9 writes. Moreover, these mechanisms do not support applications that require crash consistency when using PM as storage [77]. Kevlar incurs none of these hardware overheads and uses a novel sampling scheme to estimate wear completely in the OS.

Heterogeneous main memory: Several works [5,55,84] manage footprint between DRAM and PM for applications that prefer DRAMs for high performance. These works map heavily and least accessed regions of application footprint to DRAM and PM respectively. Unlike these works [5,55,84], Kevlar exploits heterogeneity to reduce PM wear.

Currently, Kevlar operates at a small (4KB) page granularity. However, huge (2MB) pages are increasingly being used to minimize performance penalties of using small pages (due to increased TLB pressure), especially in virtualized systems. Kevlar can be further extended to operate at a huge page granularity. For instance, Kevlar can be integrated with mechanisms such as Thermostat [5] to split a huge page into small pages, monitor write rate at granularity of small pages, and migrate pages between DRAM and PM. We leave evaluation of Kevlar's wear-reduction mechanism and development of shuffling strategies to operate at a huge page granularity to future work.

Other. DCW [119] and Flip-N-Write [21] perform read-compare-write operation to ensure that only the data bits that have changed are written. Bittman et al. [14] proposes data structures aimed at minimizing the number of bit-flips per PM write operation. Ferreira et al. [32] enable eviction of clean cache lines over dirty cache lines at the expense of potentially slowing down future reads to evicted cache lines. Recent works, MCT [25] and Mellow Writes [112], improve the endurance by reconfiguring memory voltage levels and slowing write accesses to the PM. These proposals can achieve high device lifetime but at a significant performance overhead, especially when write latency is critical to application performance [77]. NVM-Duet [69] employs a smart-refresh mechanism to eliminate redundant memory refresh operations thereby reducing PM wear. Others [49,118] propose solutions to manage wear when using persistent memory technologies to build caches. These techniques are orthogonal to our proposal and can be used in conjunction with Kevlar.

7.2 Wear-leveling mechanisms

Qureshi et al. [87], Zhou et al. [119], Security refresh [92], Online Attack Detection [86] and Start-Gap [85] observe that cache lines within a PM page do not wear out equally and propose mechanisms to remap cache lines for uniform intra-page wear. All of these works rely on additional address indirection mechanisms in hardware. Unlike Kevlar, these mechanisms cannot exploit the heterogeneity of memory systems as discussed earlier in Section 2.2.

Error recovery. DRM [47] and SAFER [93] gracefully degrades PM capacity as memory cells wear out by reusing and remapping failed cells to store data. FREE-p [110] and NVMAlloc [76] leverage ECC and checksum mechanisms to tolerate wear out errors.

8 Conclusion

We have presented Kevlar, a wear-management mechanism for persistent memories. Kevlar relies on a software *wear-estimation* mechanism that uses PEBS-based sampling in a novel approach to estimate dirty cache contents and predict writebacks to PM. It uses a *wear-leveling* mechanism that shuffles PM pages every ~4 hours with an overhead of less than 0.10% achieving up to 31.7× higher lifetime as compared to PM with no wear leveling. Kevlar employs *wear-reduction* mechanism to further extend PM lifetime. It migrates the hottest pages to higher durability memory. Kevlar, implemented in Linux kernel (version 4.5.0), achieves four-year target lifetime with 1.2% performance overhead.

Acknowledgements

We would like to thank our shepherd, Carl Waldspurger, and the anonymous reviewers for their valuable feedback. We are grateful to Akshitha Sriraman, Kumar Aanjaneya, Neha Agarwal, Animesh Jain, and Amirhossein Mirhosseini for their suggestions that helped us improve this work. This work was supported by ARM and the National Science Foundation under the award NSF-CCF-1525372.

References

[1] Aerospike. http://www.aerospike.com/. [Online; accessed 17-Jun-2017].

[2] pmem.io: Persistent memory programming. https://pmem.io/pmdk/.

[3] Reimagining the Data Center Memory and Storage Hierarchy. https://newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy.

[4] Memcached - a distributed memory object caching system, 2012.

[5] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[6] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 62–77, New York, NY, USA, 2018. ACM.

[7] ARM. Embedded trace macrocell, 2011. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0014q/IHI0014Q_etm_architecture_spec.pdf.

[8] ARM. Arm architecture reference manual, 2017. https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf.

[9] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1753–1758, New York, NY, USA, 2017. ACM.

[10] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage and recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[11] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proc. VLDB Endow.*, 10(4):337–348, November 2016.

[12] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 211–224, Berkeley, CA, USA, 2011. USENIX Association.

[13] N. Beckmann and D. Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75, Feb 2015.

[14] Daniel Bittman, Mathew Gray, Justin Raizes, Sinjoni Mukhopadhyay, Matt Bryson, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. Designing data structures to minimize bit flips on nvm. In *The 7th IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, August 2018.

[15] 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, Santa Clara, CA, 2017. USENIX Association.

[16] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[17] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[18] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2014.

[19] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Vaglis. Rewind: Recovery write-ahead system for in-memory non-volatile data structures. *Proceedings of the VLDB Endowment*, 8(5), 2015.

[20] I-C. K. Chen, C-C. Lee, and T. N. Mudge. Instruction prefetching using branch prediction information. In *Proc. of the International Conference on Computer Design*, 1997.

[21] Sangyeun Cho and Hyunjin Lee. Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

[22] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[23] 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 22nd ACM Symposium on Operating Systems Principles*, 2009.

[24] 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.

[25] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Frederic T. Chong. Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in nvms. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 232–244, New York, NY, USA, 2017. ACM.

[26] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. Pdram: A hybrid pram and dram main memory system. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, 2009.

[27] 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.

[28] Kshitij Doshi, Ellis Giles, and Peter Varman. Atomic persistence for scm with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 77–89. IEEE, 2016.

[29] Paul J Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. *Advanced Micro Devices*, 2007. http://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf.

[30] 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 9th European Conference on Computer Systems*, 2014.

[31] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 15:1–15:16, New York, NY, USA, 2016. ACM.

[32] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. Increasing pcm main memory lifetime. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10.

[33] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, June 2005.

[34] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Badgertrap: A tool to instrument x86-64 tlb misses. *SIGARCH Comput. Archit. News*.

[35] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Failure-atomic synchronization-free regions, 2018. http://nvmw.ucsd.edu/nvmw18-program/unzip/current/nvmw2018-final42.pdf.

[36] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 46–61, New York, NY, USA, 2018. ACM.

[37] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 319–331, Boston, MA, 2012. USENIX.

[38] SAP HANA. Bringing persistent memory technology to sap hana: Opportunities and challenges, 2016. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2016/20160810_FR21_Caklovic.pdf.

[39] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, December 1989.

[40] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan. Unified address translation for memory-mapped ssds with flashmap. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 580–591, June 2015.

[41] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, Santa Clara, CA, 2017. USENIX Association.

[42] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 967–979, Boston, MA, 2018. USENIX Association.

[43] 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, 2018. USENIX Association.

[44] Intel. Intel microarchitecture codename nehalem performance monitoring unit programming guide (nehalem core pmu). https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf.

[45] Intel. Intel 64 and ia-32 architectures software developer's manual, 2018. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[46] Intel and Micron. Intel and micron produce breakthrough memory technology, 2015. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

[47] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: Building reliable systems from nanoscale resistive memories. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.

[48] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[49] Yongsoo Joo, Dimin Niu, Xiangyu Dong, Guangyu Sun, Naehyuck Chang, and Yuan Xie. Energy- and endurance-aware design of phase change memory caches. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, 2010.

[50] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25, September 2010.

[51] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, Feb 2017.

[52] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient persist barriers for multicores. In *Proceedings of the international symposium on Microarchitecture*, 2015.

[53] 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 &Amp; IEEE International Conference on Embedded Software*, EMSOFT '06, pages 161–170, New York, NY, USA, 2006. ACM.

[54] 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)*, pages 993–1005, Boston, MA, 2018. USENIX Association.

[55] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 521–534, New York, NY, USA, 2017. ACM.

[56] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *2007 25th International Conference on Computer Design*, pages 245–250, Oct 2007.

[57] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 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 '18, pages 297–312, New York, NY, USA, 2018. ACM.

[58] 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, May 2002.

[59] Hideaki Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[60] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P.M. Chen, and T.F. Wenisch. Delegated persist ordering. In *Proceedings of the 49th International Symposium on Microarchitecture*, 2016.

[61] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 481–493, New York, NY, USA, 2017. ACM.

[62] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Tarp: Translating acquire-release persistency, 2017. http://nvmw.eng.ucsd.edu/2017/assets/abstracts/1.

[63] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[64] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[65] 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 Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[66] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 339–353, Santa Clara, CA, 2016. USENIX Association.

[67] H. L. Li, C. L. Yang, and H. W. Tseng. Energy-aware flash memory management in virtual memory system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(8):952–964, Aug 2008.

[68] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei 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, 2017.

[69] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. Nvm duet: unified working memory and persistent store architecture. In *Proceedings of the international conference on Architectural Support for Programming Languages an Operating Systems*, 2014.

[70] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, 2017. USENIX Association.

[71] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti. Laser: Light, accurate sharing detection and repair. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–273, March 2016.

[72] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[73] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[74] Paul Menage. Memory resource controller, 2016. http://elixir.free-electrons.com/linux/latest/source/Documentation/cgroup-v1/memory.txt.

[75] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. Enabling efficient and scalable hybrid memories using fine-granularity dram cache management. *IEEE Comput. Archit. Lett.*

[76] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, pages 1:1–1:17, New York, NY, USA, 2013. ACM.

[77] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. In *Proceedings of the Twenty-Second International Conference*

on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, pages 135–148, New York, NY, USA, 2017. ACM.

[78] Faisal Nawab, Dhruva Chakrabarti, Terence Kelly, and Charles B. Morey III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. Technical Report HPL-2014-70, Hewlett-Packard, December 2014.

[79] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecom application transaction processing benchmark, 2011. http://tatpbenchmark.sourceforge.net/.

[80] T. Nguyen and D. Wentzlaff. Picl: A software-transparent, persistent cache log for nonvolatile main memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 507–519, Oct 2018.

[81] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. Sofort: A hybrid scm-dram storage engine for fast data recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, 2014.

[82] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[83] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceedings of the 41st International Symposium on Computer Architecture*, 2014.

[84] S. Phadke and S. Narayanasamy. Mlp aware heterogeneous memory system. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[85] Moinuddin K. Qureshi, Michele M. Franchescini, Vijayalakshmi Srinivasan, Luis A. Lastras, Bulent Abali, and John Karidis. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

[86] Moinuddin K. Qureshi, Andre Seznec, Luis A. Lastras, and Michele M. Franchescini. Practical and secure pcm systems by online detection of malicious write streams. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, 2011.

[87] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[88] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, 2011.

[89] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, 2016.

[90] D. L. Schuff, B. S. Parsons, and V. S. Pai. Multicore-aware reuse distance analysis. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010.

[91] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 53–64, New York, NY, USA, 2010. ACM.

[92] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[93] Nak Hee Seong, Dong Hyuk Woo, Vijayalakshmi Srinivasan, Jude A. Rivers, and Hsien-Hsin S. Lee. Safer: Stuck-at-fault error recovery for memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, 2010.

[94] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, 2016.

[95] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: A flexible and fast software supported hardware logging approach for nvm.

In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 178–190, New York, NY, USA, 2017. ACM.

[96] Peter Snyder. tmpfs: A virtual memory file system. In *In Proceedings of the Autumn 1990 European UNIX Users' Group Conference*, pages 241–248, 1990.

[97] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: Architecture of a real-time operational dbms. *Proc. VLDB Endow.*, 9(13):1389–1400, September 2016.

[98] Transaction Processing Performance Council (TPC). Tpc benchmark C, 2010. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5-11.pdf.

[99] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the USENIX Conference on File and Storage Technologies*, February 2011.

[100] 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, 2014.

[101] Haris Volos, Andres Jaan Tack, and Michael M. Swift E. Mnemosyne: Leightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[102] 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, Santa Clara, CA, 2017. USENIX Association.

[103] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, 2015. USENIX Association.

[104] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, June 2014.

[105] Xiaojian Wu and A. L. Narasimha Reddy. Scmfs: a file system for storage class memory. In *In Proceedings of the International Conference for High Performance Computing*, 2011.

[106] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the challenges of crossbar resistive memory architectures. In *In Proceedings of the International Symposium on High Performance Computer Architecture*, 2015.

[107] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, 2016.

[108] 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, pages 478–496, New York, NY, USA, 2017. ACM.

[109] 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 17)*, pages 15–28, Santa Clara, CA, 2017. USENIX Association.

[110] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P. Jouppi, and Mattan Erez. Free-p: Protecting non-volatile memory against both hard and soft errors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[111] H. C. Yu, K. C. Lin, K. F. Lin, C. Y. Huang, Y. D. Chih, T. C. Ong, J. Chang, S. Natarajan, and L. C. Tran. Cycling endurance optimization scheme for 1mb stt-mram in 40nm technology. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 224–225, Feb 2013.

[112] Lunkai Zhang, Brian Neely, Diana Franklin, Dmitri Strukov, Yuan Xie, and Frederic T. Chong. Mellow writes: Extending lifetime in resistive memories through selective slow write backs. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 519–531, Piscataway, NJ, USA, 2016. IEEE Press.

[113] Michael Zhang and Krste Asanovic. Highly-associative caches for low-power processors. In *Kool Chips Workshop, MICRO*, volume 33, 2000.

[114] Wangyuan Zhang and Tao Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures.

In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, 2009.

[115] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[116] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.

[117] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of 46th International Symposium on Microarchitecure*, 2013.

[118] Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM Trans. Archit. Code Optim.*

[119] Ping Zhou, Bo Zhao, Jun Yang, and Yutao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[120] Yanqi Zhou, Ramnatthan Alagappan, Amirsaman Memaripour, A Badam, and D Wentzlaff. Hnvm: Hybrid nvm enabled datacenter design and optimization. *Microsoft, Microsoft Research, Tech. Rep. MSR-TR-2017-8*, 2017.

Storage Gardening: Using a Virtualization Layer for Efficient Defragmentation in the WAFL File System

Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra

NetApp, Inc

As a file system ages, it can experience multiple forms of fragmentation. Fragmentation of the free space in the file system can lower write performance and subsequent read performance. Client operations as well as internal operations, such as deduplication, can fragment the layout of an individual file, which also impacts file read performance. File systems that allow sub-block granular addressing can gather intra-block fragmentation, which leads to wasted free space. This paper describes how the NetApp® WAFL® file system leverages a storage virtualization layer for defragmentation techniques that physically relocate blocks efficiently, including those in read-only snapshots. The paper analyzes the effectiveness of these techniques at reducing fragmentation and improving overall performance across various storage media.

1 Introduction

File systems typically allocate physically contiguous blocks in storage devices to write out logically sequential data and metadata. This strategy maximally uses the write bandwidth available from each storage device since more blocks can be written to it using fewer write I/Os, and it allows for optimal performance when that data or metadata is later read sequentially. Common operations such as file creations, resizes, and deletions age a file system, resulting in *free space fragmentation* and *file layout fragmentation*. Free space fragmentation results in reduced contiguous physical allocations, which in turn lowers file system write throughput [32]. Furthermore, it limits the system's ability to optimally lay out logically sequential data and metadata, thereby contributing to file layout fragmentation [33, 35]. Fragmentation impacts the I/O performance of both hard drives (HDDs) and solid state drives (SSDs), although in different ways.

File sizes rarely align with the file system block size, thus there is potential for intra-block wastage of storage space. Some file systems provide the ability to address sub-block chunks to avoid such wastage and improve storage efficiency [25, 31, 37]. However, such sub-block indexing introduces the potential for *intra-block fragmentation*, which occurs as chunks within a block are freed at different times.

A copy-on-write (COW) file system never overwrites a block containing active data or metadata in place, which makes it more susceptible to fragmentation [12]. WAFL [14] is an enterprise-grade COW file system that is subject to free space, file layout, and intra-block fragmentation. In this paper, we present techniques that efficiently address each form of fragmentation in the WAFL file system, which we refer to collectively as *storage gardening*. These techniques are novel because they leverage WAFL's implementation of virtualized file system instances (FlexVol® volumes) [9] to efficiently relocate data physically while updating a minimal amount of metadata, unlike other file systems and defragmentation tools. This virtualization layer provides two advantages: (1) The relocation of blocks needs to be recorded only in the virtual-to-physical virtualization layer rather than requiring updates to all metadata referencing the block. (2) It even allows relocation of blocks that belong to read-only snapshots of the file system, which would be ordinarily be prohibited.

Most previous studies of fragmentation predate modern storage media (i.e., SSD drives) [32, 33, 34]. Other studies were performed on commodity-grade systems (with single drives) [5, 16]; these studies draw conclusions that do not apply to enterprise-grade systems. The WAFL file system can be persisted on a variety of storage media, which makes it well-suited for this study on fragmentation. We analyze each form of fragmentation and evaluate our defragmentation techniques with various storage media permutations.

To summarize our findings, we see significant improvements in data layout metrics on HDD- and SSD-based systems using our approaches. These improvements translate into significant performance gains on HDD-based systems, which are typically I/O-bound, as well as mixed-media (HDD and SSD) systems. In contrast, the same approaches generally show negative overall performance impact on all-SSD systems, which are more sensitive to the CPU overhead incurred by defragmentation. We conclude that for SSD-based systems, it is preferable (and advantageous) to perform defragmentation only during periods of low load. Our

lessons are applicable to other file systems as well, especially ones that are COW, such as ZFS [27] and Btrfs [31].

2 An Overview of WAFL

This section presents background on WAFL—an enterprise-grade UNIX-style file system—and the trade-offs inherent in defragmentation.

2.1 File System Layout and Transaction

A Data ONTAP® storage system uses the proprietary WAFL [14] file system, which is persisted as a tree of 4KiB blocks, and all data structures of the file system, including its metadata, are stored in files. Leaf nodes (L_0s) of an inode's block tree hold the file's data. The next higher level of the tree is composed of indirect blocks (L_1s) that point with a fixed span to children L_0s; L_2s point to children L_1s, and so on. The number of levels in the block tree is determined by the size of the file. Each inode object uses a fixed number of bytes to store file attributes and the root of its block tree, unless the file size is tiny, in which case the file data is stored directly within the inode. All inodes for data and metadata are arranged in the L_0s of a special file whose block tree is rooted at the superblock. WAFL is a copy-on-write (COW) file system that never overwrites a persistent block in place. Instead, all mutations are written to free blocks and the previously allocated blocks become free.

Client operations that modify the file system make changes to in-memory data structures and are acknowledged once they have also been logged to nonvolatile memory. WAFL collects and flushes the results of thousands of operations from main memory to persistent storage as a single atomic transaction called a *consistency point* (CP) [9, 14, 19]. This delayed flushing of "dirty" blocks allows better layout decisions and amortizes the associated metadata overhead. During each CP, all updates since the previous CP are written to disk to create a self-consistent, point-in-time image of the file system. A snapshot of the file system is trivially accomplished by preserving one such image. The WAFL *write allocator* assigns available free blocks of storage to the dirty blocks during a CP. The goals of the write allocator are to maximize file system write throughput and subsequent sequential read performance.

We have previously presented the data structures and algorithms used to steer the write allocator toward the emptiest regions of storage, with built-in awareness of RAID geometry and media properties [17]. Prior work has also described how CPs manage free space in order to maximize various performance objectives [8]. In this paper, we extend these concepts further, showing how storage gardening can increase the availability of high-quality regions for writing and recreate the desired layout after file system aging has undone the initial write allocation.

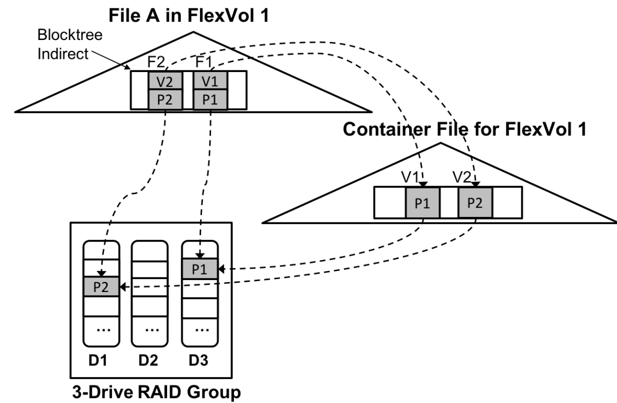


Figure 1: The relationship of a FlexVol volume with its container file and the aggregate. A real-world aggregate has many more drives.

2.2 FlexVol Volumes and Aggregates

WAFL defines collections of physical storage as *aggregates*, which are typically several dozen TiB in size. A WAFL aggregate can consist of different permutations of storage media: HDDs (hard drives) only, SSDs (solid state drives) only, HDDs and SSDs, SSDs and S3-compliant object stores, and LUNs exported from third-party storage. Storage devices with no native redundancy, such as HDDs and SSDs, are organized into RAID [6, 10, 29] groups for resiliency. Multiple aggregates are connected to each of two ONTAP® nodes that are deployed as a high-availability pair. Within each aggregate's physical storage, WAFL houses and exports hundreds of virtualized WAFL file system instances called *FlexVol volumes* [9].

Each aggregate and each FlexVol is a WAFL file system. A block in any WAFL file system instance is addressed by a *volume block number*, or *VBN*. WAFL uses a *Physical VBN* to refer to a block in the aggregate; the Physical VBN maps to a location on persistent storage. A block in a FlexVol is referenced by a *Virtual VBN*. FlexVols are stored within files that reside in the aggregate—the blocks of a FlexVol are stored as the L_0 blocks of a corresponding *container* file. The block number in the FlexVol (Virtual VBN) corresponds to the offset in the container file. Thus, the L_1 blocks of the container file are effectively an array that is indexed by the Virtual VBN to find the corresponding Physical VBN. In other words, container file L_1 blocks form a map of Physical VBNs indexed by the Virtual VBNs, which we call the *container map*. Data structures in the FlexVol store a cached copy of the Physical VBN along with the Virtual VBN pointers. In most cases, the cached Physical VBN helps avoid the extra CPU cycles and storage I/O for consulting the container map. It is possible for the cached Physical VBN in a FlexVol structure to become stale, in which case *the container map is consulted for the authoritative version*. Fig. 1 illustrates the

relationship between the Physical VBNs in an aggregate and the blocks of a File *A* in a FlexVol.

This virtualization of storage for blocks in a FlexVol and the corresponding indirection between Virtual VBNs and Physical VBNs through the container map provide the basis for the storage gardening techniques presented in this paper, as well as a wide range of technologies, such as FlexVol cloning, replication, thin provisioning, and more [9].

2.3 Performance and Defragmentation

Modifications to data and metadata in a COW file system, such as WAFL, fragment both the layout of files and the aggregate's free space. WAFL also supports sub-block addressing, and uses that to *compact* sub-4KiB chunks into a single block. These compacted blocks become fragmented as their constituent chunks are freed. Subsequent sections detail the impact of each form of fragmentation. Defragmentation is typically accomplished by relocating in-use blocks or chunks from badly fragmented regions of a file or file system. Relocating a block is trivial in many cases; the pointer stored in a parent indirect can be fixed up to point to a relocated child's new physical location. Although most file systems prevent relocation of blocks that belong to read-only snapshots, WAFL provides this functionality. Two requirements exist to support block relocation below the file system in the storage layer: (1) the ability to virtualize the address space, which WAFL provides in the form of FlexVol layering, and (2) the ability to detect stale pointers. Each of these abilities is detailed in Sec. 3.3. Although the CP amortizes the overhead associated with re-writing these blocks, defragmentation comes at a cost (CPU cycles and I/Os). An enterprise storage system must consider this cost in the context of the storage media type before it chooses to defragment. This paper explains the defragmentation techniques used by WAFL, and how these trade-offs play out in various Data ONTAP configurations.

3 Free Space Fragmentation

This section discusses the effect of free space fragmentation and the technique used to counter it.

3.1 Background on Space Fragmentation

WAFL groups HDDs and SSDs of an aggregate into RAID groups to protect against device errors and failures. As Fig. 2(A) shows, a *stripe* is a set of blocks, one per device, that share the same parity block. A *full stripe write* is one in which all data blocks in the stripe are written out together such that RAID can compute the parity without additional reads. Fragmentation of free space leads to *partial stripe writes*, shown in Fig. 2(B), which require RAID to read data blocks to compute parity [29]. Writing logically sequential blocks of the file system to consecutive blocks of a storage device reduces the total number of write I/Os to the device and improves sequential read performance, because the blocks can be read with a single I/O [2]. Contiguous free space on devices, such as on D1 in Fig. 2(B), is required to meet this objective, by facilitating long *write chains*. Fragmented free space decreases the availability of contiguous free blocks on each device, as shown on drives D2 and D3.

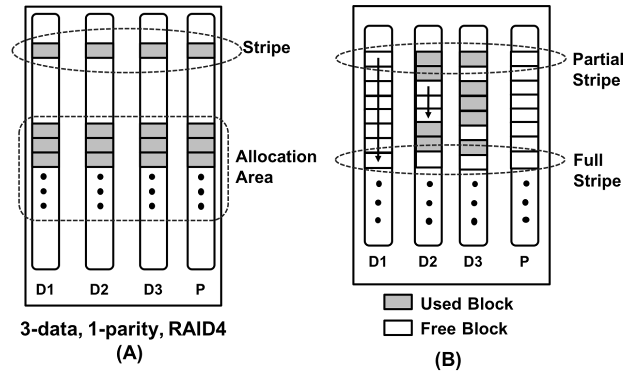


Figure 2: (A) A sample RAID-4 group with 3 data and 1 parity storage device (for simplicity). (B) A sample aged RAID group with free space fragmentation.

The latency of a write operation is not *directly* affected by free space fragmentation because WAFL acknowledges a write operation immediately after it is logged to nonvolatile memory. Fragmentation makes each CP more expensive, which indirectly impacts client operations. First, more CPU cycles are required to find free blocks to allocate [19] and compute RAID parity, which causes the WAFL scheduler to divert more CPU away from client operations so the CP can complete in time. Second, more I/Os of shorter write chains are required to flush out all the dirty blocks of the CP, which takes storage I/O bandwidth away from client operations.

Fragmentation can also impact performance by making free space reclamation more expensive in terms of CPU cycles and metadata updates. Over time, several improvements to free space reclamation have ensured that WAFL now performs efficiently even in the presence of fragmentation [19]. However, this concern still applies in most other file systems.

3.2 Segment Cleaning in WAFL

The goal of free space defragmentation is to make emptier regions of free space available to the write allocator. In-use blocks need to be efficiently relocated to create large areas of free space without violating invariants associated with blocks in FlexVol snapshots. Prior work [17] describes how the WAFL write allocator segments each RAID group into *allocation areas (AAs)* when choosing free Physical VBNs for the CP. As Fig. 2(A) shows, an AA is a set of consecutive RAID stripes; the AA size depends on storage media prop-

erties [17]. Defragmentation operates by *segment cleaning* at the AA granularity. The cleaning of an AA entails consulting free space metadata in WAFL [18] to pick stripes in the AA that are worth cleaning, reading all in-use blocks of such stripes into the buffer cache, and tagging them dirty. WAFL stores a *context* together with each written block [36], which identifies its file and file block offset[1]. Cleaning uses this context to determine the file and offset of the in-use block and marks the buffer dirty in the corresponding file. The subsequent CP processes these dirty buffers (together with all others) and writes them out to new Physical VBNs, thereby freeing the previously used blocks and creating an emptier AA. The parent indirect block of such a rewritten block (much like that of any dirty block) is updated by the CP to reflect its new Physical VBN.

3.3 Blocks in the FlexVol Volume

The vast majority of the blocks in an aggregate belong to its FlexVols, because they contain user data. WAFL leverages the indirection provided by the FlexVol virtualization layer to efficiently relocate FlexVol blocks. In particular, such blocks are relocated by loading and dirtying them as L_0 blocks of the corresponding container file, rather than as blocks in the block tree within the volume. Thus, a relocated block gets a new Physical VBN, but its Virtual VBN remains unchanged and no changes are made *within* the volume. Fig. 3 shows an example in which blocks are moved from within allocation areas AA_x and AA_y. The cleaner determines all in-use blocks (i.e., p_1–p_5) in these AAs and reads each of these blocks into memory, along with its associated context. The context for a block in a FlexVol refers to its container file and Virtual VBN. Thus, p_1–p_3 of File A are marked as dirty L_0s of FlexVol 1's container file, and p_4–p_5 are marked as dirty L_0s of File B (a metadata file in the aggregate). In the subsequent CP, the write-allocator rewrites these blocks together with other dirty buffers to a new AA_z, thereby emptying AA_x and AA_y. Note that File A's indirect blocks continue to point to stale Physical VBNs p_1–p_3 (as discussed in detail later in this section), whereas File B and the container file of the FlexVol are up to date.

Leveraging the virtualization layer provided by the container file yields two key benefits. First, it facilitates relocation of blocks in a snapshot because file system invariants associated with snapshots are preserved within the FlexVol layer. Blocks in a snapshot image of the FlexVol are immutable and therefore forbidden from being dirtied and processed by the CP. Such rules are typical across file system implementations. In theory, it is possible to physically relocate blocks within snapshots without virtualization, but this requires the ability to update metadata within a snapshot to

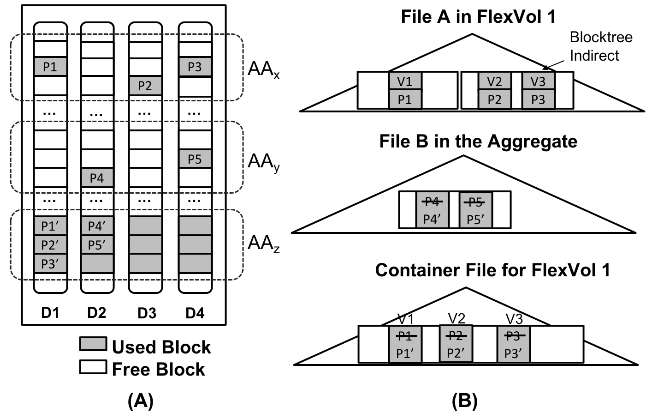


Figure 3: (A) 4 drives with 5 blocks randomly allocated in AA_x and AA_y and the same 5 blocks relocated to AA_z to create empty AAs. (B) Impact of block relocation due to segment cleaning within the L_1s of File A in FlexVol 1 and a File B in that aggregate, and the relevant changes to the container file.

reflect those relocations. In a COW file system, such updates cascade up the file system tree, resulting in further updates. Physical-only relocation via the container preserves the Virtual VBN, and that leaves the FlexVol snapshot image intact, including all file system metadata for block allocation. Given the popularity of FlexVol snapshots, such block relocation is critical to efficient defragmentation[2].

The second benefit is that the requisite metadata updates are minimized. Relocating a block is expensive in a COW file system like WAFL because every ancestor block in the file system tree needs to be rewritten to point to the new location of its child. By leveraging the container map for the FlexVol, blocks within the FlexVol are not rewritten. Further, as described in Sec. 2.1, the file system tree of blocks rooted at the superblock of a FlexVol can be quite tall, whereas the height of the container file is a function only of the size of its Virtual VBN space. Thus, the tree of blocks comprising the container file is significantly shorter, and higher-level indirects of a container file are likely to be already dirty due to the batching effect of the CP.

A file system operation that accesses a relocated block of File A uses the stale pointer in that indirect, such as p_1, to read a block from storage. If WAFL has not yet assigned the previously freed p_1 for a new write, the context check succeeds and the I/O is accomplished. Otherwise, the context check fails and the operation pays a *redirection penalty* to consult the container map, using v_1 to read p'_1. The pointer in File A's parent indirect block can optionally be fixed to p'_1 either via a background scan or opportunistically after the redirected read to avoid the penalty on subsequent accesses. The use of a virtualization layer in this case provides both

[1]This context was introduced originally to protect against lost or misdirected writes [3], so that a subsequent read can detect a mismatch from the expected context.

[2]Snapshots of an aggregate are rare and short-lived, so their interaction with segment cleaning is limited, and is not discussed here.

the ability to defer fix-up work and the option to leave stale pointers in indirects in cases where the update would not be expected to improve performance. Without this virtualization, all references to the physical block would have to be corrected immediately. Although Fig. 3 depicts only L_0s of File A being relocated, any block in the block tree of any file in the FlexVol can be relocated.

3.4 Continuous Segment Cleaning

Segment cleaning was first introduced for all-HDD aggregates as a background scan that walked all AAs in each RAID group. It was expensive and had to be initiated by the administrator during periods of low load. A later release introduced *continuous segment cleaning* (CSC), which runs autonomously and is more efficient. It cleans AAs just in time as they get selected for use by the write allocator. Prior work [17] shows how the WAFL write allocator uses a max-heap to pick the emptiest AA from each RAID group. Cleaning the emptiest AAs minimizes the number of in-use blocks that are required to be relocated, which in turn minimizes the total number of I/Os and CPU cycles required for this activity. This greedy approach also minimizes the subsequent redirection penalty and fix-up work for the file system.

4 File Layout Fragmentation

This section discusses how files become fragmented in WAFL and the approach used to counteract that fragmentation.

4.1 Background on File Fragmentation

The WAFL write allocator attempts to allocate consecutive L_0s of a file sequentially on a single storage device to optimize subsequent sequential read performance. Given that WAFL is a COW file system, this layout may fragment over time. That is, even if sequential file L_0s are initially stored contiguously, continued random overwrites of the L_0s can cause them to be rewritten elsewhere. It should be noted that neighboring offsets in a file need to be overwritten several seconds apart to fragment the file because the CP collects and processes a few seconds' worth of dirty buffers. An example of suboptimal file layout is shown in Fig. 4, in which sequential L_0s of File A are scattered across the aggregate, with Physical VBNs p_1–p_5.

Sequential reads of a fragmented file require an increased number of drive I/Os [2]. Like most file systems, WAFL detects sequential patterns in the accesses to a file, and speculatively prefetches L_0s based on heuristics. (Prefetch heuristics used by WAFL are outside the scope of this paper.) Although prefetching helps sequential read performance, the associated overhead (CPU cycles and storage I/Os) increases with fragmentation of the file layout [2, 34].

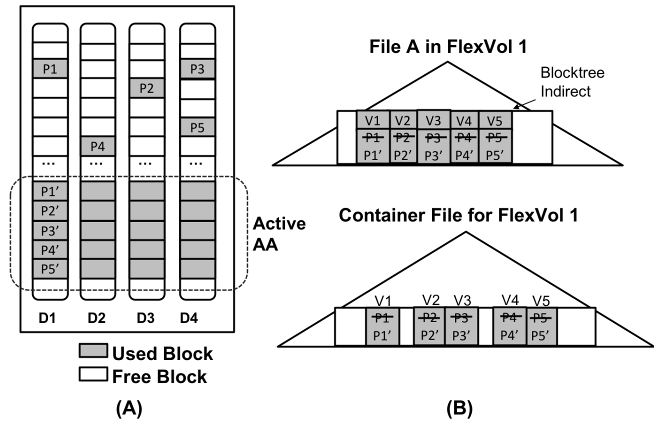


Figure 4: (A) 4 devices with 5 blocks located sequentially within a file but randomly on the storage media due to file layout fragmentation, and the same 5 blocks relocated sequentially on device D1. (B) Impact of this block movement within the L_1s of a File A in a FlexVol volume and the changes to that volume's container file.

4.2 File Defragmentation in WAFL

In theory, file layout defragmentation can be trivially accomplished in any COW file system by dirtying sequential file blocks that are not sequentially stored. These blocks will be written out sequentially during the subsequent write allocation process. File systems have two choices for how to deal with blocks shared with snapshots: (1) Update all indirect blocks pointing to the relocated block, which is not feasible because it requires modifying blocks in a snapshot. (2) Update references to the block in the "active" file system only and leave the block in place in the snapshot, which results in divergence from the snapshot and wasted storage for duplicate copies of these blocks. In WAFL, when a block in a FlexVol is dirtied, it is assigned not just a new Physical VBN but also a new Virtual VBN. The allocation of a new Virtual VBN is reflected in the FlexVol metadata, which results in divergence from the most recent snapshot of the volume. Efficient replication technologies minimize the amount of data transferred in each periodic incremental update [30], which is accomplished by *diff'ing* per-snapshot metadata to efficiently compute changes to the file system.

The need to keep FlexVol metadata intact motivates another physical-only block relocation strategy by leveraging container file indirection. In particular, WAFL tags file-defragmented blocks as *fake dirty*, which conveys that the content of the data block is unchanged and should not diverge from any snapshot to which it belongs. In the next CP, a fake dirty buffer is assigned a new Physical VBN without changing its Virtual VBN. Fig. 4 shows the result of this process on File A; the L_0s retain their Virtual VBNs while getting reallocated sequentially from p'_1 to p'_5. Thus, these relocated blocks do not create false positives during the aforementioned snapshot *diff* process, and WAFL replication tech-

nologies remain efficient. Although Physical VBNs cached in snapshot copies of indirect blocks become stale, a failed read is redirected through the container map to the new location of the block.

File layout defragmentation in WAFL is similar to free space defragmentation, in that a relocated block only acquires a new Physical VBN. However, file defragmentation is different in two ways that make fake dirties more effective for this use case. First, the file blocks being dirtied are by definition contiguous in the file block space, so the COW-related overhead for the block tree in the FlexVol is amortized across multiple fake dirty blocks. Second, file defragmentation is triggered in cases where future sequential file accesses are anticipated (as discussed in Sec. 4.3), so it is desirable to "fix up" the block tree indirects right away rather than deferring the effort.

Relocating L_0 blocks that are shared with other files as a result of deduplication or file cloning does not create inconsistencies. For example, if some File B shares the first L_0 of File A, the parent L_1 in File B is not changed by the defragmentation of File A and therefore points to v_1 and p_1 even after the L_0 of File A is relocated to p'_1. As described in Sec. 3.3, once the now-free p_1 is reused, any subsequent read via that stale pointer in File B fails the context match, and is redirected to p'_1 via the container map. It should be noted that relocating blocks p_1–p_5 in File A could potentially fragment File B if it shares some of these blocks but at different offsets. However, fragmentation resulting from deduplication is unlikely in real-world datasets and as far as we know has not been encountered in our customer deployments. We find that multiple L_0s are shared in the same order between files, so defragmentation of one helps all such files[3].

4.3 Write After Read

File defragmentation was originally introduced as an administrator-initiated command to kick off defragmentation of a specific file or all user files in a FlexVol. When invoked, the Physical VBN pointers in the L_1s of the file are inspected to determine whether defragmentation could result in improved read performance. If so, the L_0 blocks are read into the buffer cache and tagged fake dirty. Autonomous file defragmentation—*write after read* (WAR)—was introduced in a later release. When enabled, it uses heuristics to defragment files that get accessed sequentially by client operations, but only when the system has sufficient availability of CPU cycles and I/O bandwidth. These techniques can also be applied to metadata such as directories[4].

[3]WAFL deduplication code paths track how many consecutive file blocks are detected as duplicates and replaced. Statistics from customer deployments show this number to be mostly in the 4 to 8 range.

[4]WAFL uses several techniques to optimize metadata access, some of which are described in prior work [19]. Such optimizations have made it unnecessary to employ WAR on metadata.

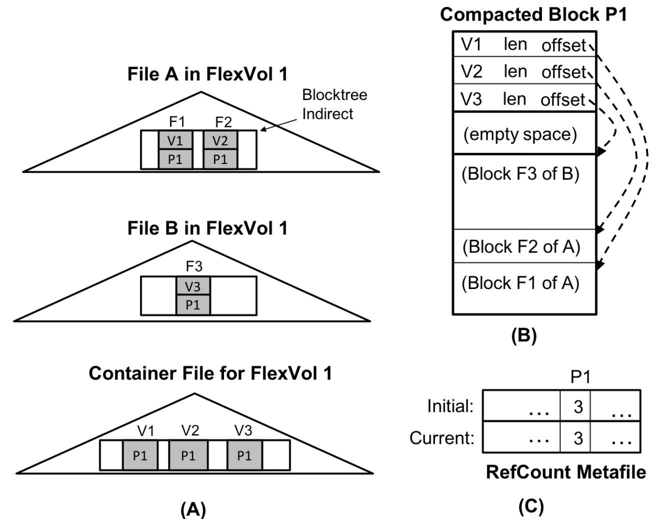


Figure 5: (A) Files A and B with sub-block chunks that share a physical block and the corresponding container file. (B) Format of the physical block containing the three chunks. (C) The refcount file with an "initial" and "current" value of 3 for physical block p_1.

5 Intra-Block Fragmentation

This section describes intra-block fragmentation in WAFL and its mitigation by using the virtualization layer. There are two sources of intra-block wastage in WAFL. First, when the size of a file is not an exact multiple of 4KiB (the block size used by WAFL), the unused portion of the last L_0 of that file is wasted. This may result in significant wastage of storage space, but only if the dataset contains a very large number of small files. Second, data compression in WAFL can result in intra-block wastage even for large files. WAFL uses various algorithms to compress the data stored in two or more consecutive L_0s of a file, and writes the compressed result to the file. Thus, a user file L_1 points to fewer L_0s and some "holes" to indicate blocks saved by that compression. Each set of such L_0s is called a *compression group*. The compressed data rarely aligns to the 4 KiB block boundary; therefore space is almost always wasted in the tail L_0 of every compression group. For example, 8KiB of data may compress down to 5KiB. This would consume two 4KiB blocks of storage where 3KiB of the second block is wasted. Compression has become ubiquitous in Data ONTAP deployments, with significant savings reported for large-file datasets. *Sub-block compaction* enables WAFL to pack multiple sub-block chunks from tails of one or more compression groups and/or files into a single physical block.

5.1 Sub-Block Compaction

Sub-block chunk addressability leverages FlexVol virtualization to remain transparent to the FlexVol layer. Blocks within

a FlexVol retain their Virtual VBNs, but now multiple Virtual VBNs can share one Physical VBN. The corresponding physical block contains the sub-block chunk associated with each Virtual VBN. Similarly, a container file can contain multiple references to the Physical VBN of a single compacted block. Fig. 5(A) shows how a compacted block p_1 is pointed to by three Virtual VBNs from a container file. These chunks are from two different files, A and B, and can be either tails of compression groups or uncompressed but partially filled blocks. As Fig. 5(B) shows, compacted block p_1 starts with a vector of tuples, which allows for chunks of different lengths to be compacted together. When a block is read, the tuples are parsed to locate the desired data. Each WAFL instance has a *refcount metadata file* that tracks references to a block [19]. The refcount file of an aggregate tracks in-use chunks in a compacted Physical VBN[5].

This design offers several clear benefits. First, compaction through the container file keeps it independent of the FlexVol. Thus, although blocks in a snapshot are immutable, they can be compacted or recompacted via the container file. Second, without compaction, compression is beneficial only if the compression group yields at least one block in savings, whereas compaction can exploit savings of less than 4KiB. Third, there is no fixed chunk size, which means that sub-blocks can be compacted together based on workload-aware criteria rather than on their sizes. For example, client-access heuristics can be used to compact together "hot" blocks that might get overwritten soon versus "cold" blocks.

5.2 Recompaction

Over time, one or more chunks within a compacted physical block may get freed due to overwrites or file truncations, which results in intra-block fragmentation within previously compacted blocks, providing an opportunity to reclaim wasted space via *recompaction*.

Recompaction in WAFL is performed by a *recompaction scanner* that walks the container map in Virtual VBN order and chooses blocks to defragment. The per-Physical VBN entry in the refcount file contains two sub-counts: r_i, the initial number of chunks in the Physical VBN when it was first written out, and r_c, the current number of chunks referenced by the container map. As shown in Fig. 5(C), both sub-counts are initialized to the number of chunks when a compacted block is written out. As chunks are freed, r_c is decremented. The recompaction scanner uses the two sub-counts to predict whether a block is worth recompacting. If $\frac{r_c}{r_i}$ is below some threshold (specified by the administrator based on desired aggressiveness), the block is read in from storage and its contents are examined to determine the actual free space in the block. If the block is truly worth recompacting, each chunk is marked dirty as a standalone container L_0 block. The subsequent CP applies compaction to all such blocks and writes them out in newly compacted blocks. In this way, Physical VBNs are changed transparently under the FlexVol virtualization layer, leaving stale Physical VBNs cached in FlexVol data structures.

A fourth type of fragmentation occurs in Data ONTAP that is similar to intra-block fragmentation. Recently introduced all-SSD FabricPool aggregates collect and tier cold blocks as 4MiB objects to an object store, for example to a remote hyperscaler such as AWS. As with compacted blocks, objects may become fragmented due to block frees, and objects can be freed only once all used blocks are freed. Object defragmentation consists of marking all blocks within sparsely populated objects dirty and rewriting them into new objects. It leverages the container file indirection to avoid changes within the FlexVol when the block is moved. Defragmentation is triggered by comparing the monetary cost of wasted storage versus the cost of GETs and PUTs to rewrite the defragmented data. Thresholds of allowed free blocks in an object are defined per-hyperscaler such that the cost of GETs and PUTs to defragment objects breaks even within a month when compared to savings from the reduced storage.

6 Interactions between Techniques

CSC, WAR, and recompaction can run concurrently and with very little adverse interaction. Segment cleaning generates empty AAs for use by the write allocator, which naturally facilitates efficient file block reallocation. No additional requirement is placed on CSC because of WAR. CSC and recompaction both operate by generating dirty L_0s of container files. A block being relocated by CSC may be a compacted block, in which case it may be unnecessarily recompacted in order to reclaim the old Physical VBN.

In theory, defragmentation techniques could invalidate a large number of cached Physical VBNs, which may affect performance because of more fix-up work and/or read redirection penalty. As described in this paper, defragmentation is used with care and only when the associated overhead is justified; as far as we know, no customer systems have been impacted by any pathological scenarios of defragmentation[6].

7 Evaluation

It is not practical to formulate an apples-to-apples comparison of the defragmentation techniques in WAFL with that in other file systems, due to the configurations, sizes, and its large feature set. Instead, this section provides some historical context and explores the trade-offs inherent to each of

[5]The refcount file supports deduplication; blocks from different files and FlexVols may refer to the same Physical VBN. Prior work [19] studies the performance implications of maintaining a refcount file.

[6]Specific features unrelated to defragmentation, such as Volume Move [1], may create scenarios leading to severe redirections, but purpose-built scanners have been designed to handle them.

our techniques across some key configurations.

Data ONTAP is deployed by enterprises in different business segments for a wide variety of use cases. A typical NetApp storage controller might be hosting datasets for multiple instances of different applications that are actively accessed at the same time. In such multitenant environments, no individual customer workload represents the range of possible outcomes. Instead, we use a set of microbenchmarks and an in-house benchmark that represent specific average and worst-case scenarios, but the conclusions are applicable across the majority of benchmarks that we track. The IOPS mix—random reads, random and semisequential overwrites—of the in-house benchmark is designed to be identical to that of the industry-standard SPC-1 benchmark [7], and models the query and update operations of an OLTP/DB application. We generate load by using NFS or SCSI clients based on convenience, but the choice of protocol does not make any material difference to the results presented. Unless otherwise specified, all experiments use our midrange system, which has 20 Intel Xeon E5v2 2.8GHz 25MB-cache cores (10-cores x 2 sockets) with 128GiB of DRAM and 16GiB of NVRAM.

7.1 Free Space Defragmentation

The original ONTAP deployments (20+ years ago) were HDD-only, and WAFL had no defragmentation capability at the time. Because those systems were typically bottlenecked by hard-drive bandwidth, higher performance required attaching more HDDs. Segment cleaning was designed into WAFL soon after the FlexVol layering was introduced. However, it could be initiated only by the administrator, based on observed system performance. In most cases, background defragmentation scans were configured to run during weekends or known times of low load. Just-in-time segment cleaning (CSC) was introduced later, as discussed in Sec. 3.2. Here, we evaluate the performance benefits of CSC across different storage media configurations.

To evaluate the benefits of CSC on all-HDD aggregates, we directed a load of 3K ops/sec of the in-house OLTP/DB benchmark using clients connected over Fibre Channel to a low-end 8-core system with an aggregate composed of 22 10K RPM HDDs of 136GiB each. Fig. 6 presents the results of our test over a 60+ day window[7]. Without CSC, client observed latency continues to increase over time due to increasing fragmentation in the file system. As shown in Fig. 6(A), CSC carries some initial overhead that results in higher client latency for the first 35 days, but it eventually delivers layout benefits that yield a stable and lower client latency over time. Both write chain length and parity reads are greatly improved by using CSC (Fig. 6(B)), as write chain length converges toward a worst-case value of 1 without CSC.

[7]It takes a long while to fragment a real-world-sized all-HDD aggregate, given its low IOPS capability; all-SSD aggregates can be fragmented faster.

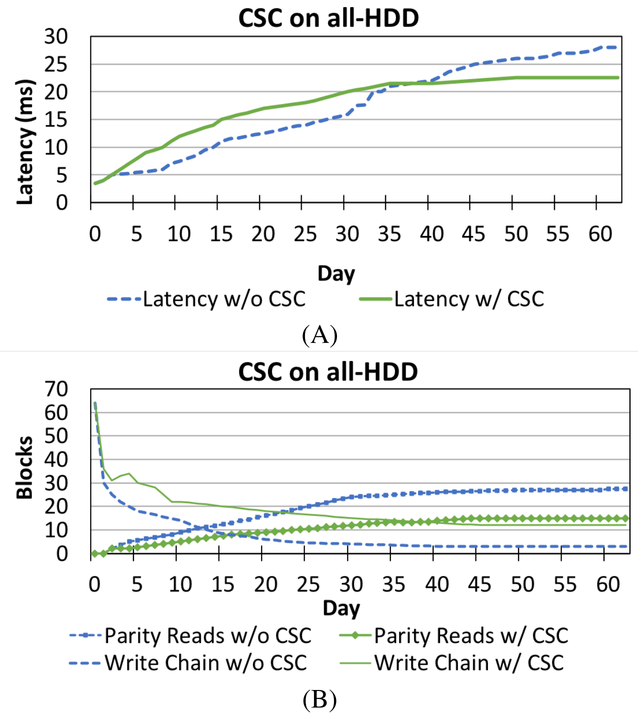


Figure 6: (A) Client observed operation latency and (B) parity reads/sec and write chain lengths with and without CSC on an all-HDD aggregate over 62 days running an OLTP/DB benchmark.

Introduced in 2012, NetApp Flash Pool® aggregates mix RAID groups of SSDs together with RAID groups of HDDs. At that time, enterprise-quality SSDs were 100 to 200GiB in size and relatively expensive. Therefore, based on cost-benefit analysis for ONTAP systems, SSDs could make up at most 10% of an aggregate's total capacity. WAFL used heuristics to determine where a particular block was to be stored. For example, "hot" (based on access patterns) data and metadata blocks were stored or even cached in the SSDs, and "cold" blocks were stored in or tiered down to HDDs.

An interesting effect of biasing hot blocks to SSDs was that fragmentation was mostly isolated to the SSD tier, and infrequent deletion of cold blocks in the HDD tier was insufficient to fragment the HDD tier. This was verified by repeating the OLTP/DB experiment previously described, but at a higher load on a Flash Pool aggregate composed of 12 SSDs and several HDDs. Hot spots of the working set stayed in the SSD tier, and write chains to HDDs declined very slowly, leveled out at around 48 blocks after 22 days, and remained stable for the remainder of the measurement interval (60+ days), without any need for CSC; the graph for this experiment is not shown.

On the other hand, the SSD tier of a Flash Pool aggregate fragments very quickly. We studied this by running the OLTP/DB benchmark benchmark on the midrange system with 12 200GiB SSDs and a large number of HDDs.

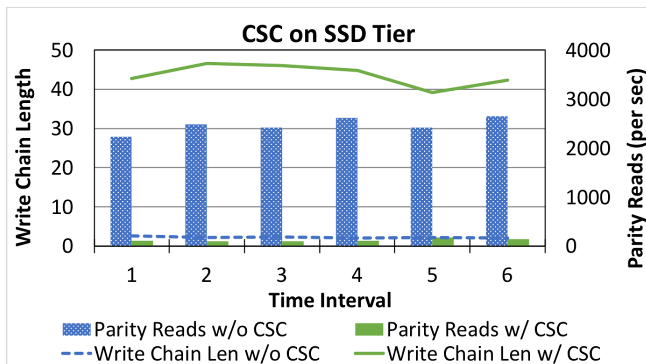


Figure 7: Parity reads per second and write chain lengths for the SSDs in a mixed aggregate during the in-house OLTP/DB benchmark, with and without CSC on the SSD tier.

The SSD tier is fragmented by running a heavy load (100K op/sec) for 2 hours, followed by another 2 hours of a more moderate and recommended load (50K op/sec) for that configuration. Fig. 7 shows write chains and parity reads per second within the SSD tier during the latter period. Both metrics show a marked improvement with CSC, with write chain lengths of about 40 instead of 2. Although CSC results in a small increase in client latency—from 0.79ms to 0.82ms (not shown here)—it was still beneficial for this earlier generation of SSDs, which were more prone to wear out. SSDs have a flash translation layer (FTL) that generates empty *erase blocks* for new writes and evenly wears out the SSD by moving data around within the SSD [28]. It is well known that shorter and more random write chains lead to higher *write amplification* on SSDs, which impacts SSD lifetime. Prior work [17] explains how the choice of the AA size in WAFL minimizes negative log-on-log behaviour [41] in devices using translation layers such as SSDs or SMR drives.

Following architectural improvements to the WAFL I/O path, ONTAP systems with all-SSD aggregates were introduced in 2015. As the size of the enterprise-quality SSD has increased from 100GiB to 16+ TiB in less than 5 years (remarkably), and the promise of new interconnect technologies such as NVMe [39, 40] has become a reality, the performance bottleneck in these storage systems has shifted from the media to the available CPU cycles to maximally use storage I/O bandwidth. In addition, emphasis has shifted away from SSD lifetimes and avoiding burnout due to write amplification in favor of total cost of ownership benefits as vendors manufacture SSDs with larger *drive writes per day* [4]. RAID-style fault tolerance also provides protection from burnout. Thus, write amplification due to free space fragmentation on enterprise-class SSDs is a performance problem that manifests mostly when a storage server has an excess of free CPU cycles but insufficient SSD I/O bandwidth. This is unlikely on systems with RAID-based redundancy, which require a minimum number of SSDs to amortize the

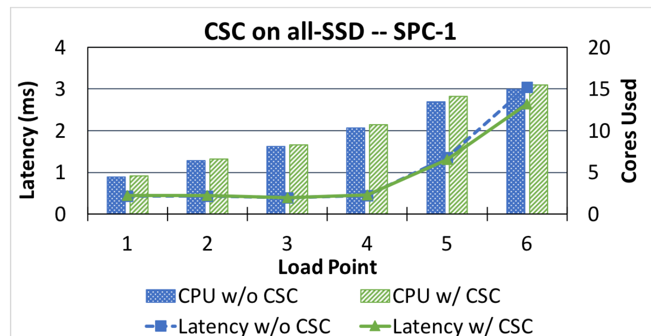


Figure 8: Latency and CPU utilization with the in-house OLTP/DB workload on an all-SSD aggregate, with and without CSC.

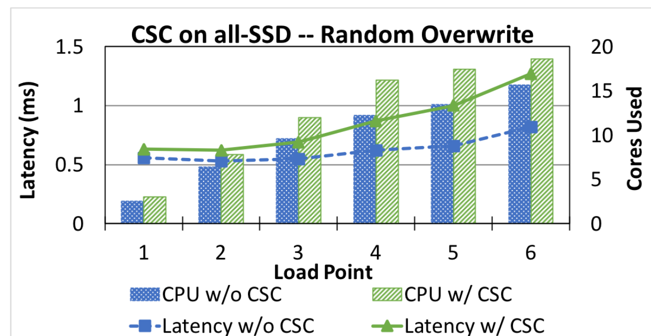


Figure 9: Latency and CPU utilization with a pure random overwrite workload on an all-SSD aggregate, with and without CSC.

space needed for storing RAID parity; ONTAP aggregates contain anywhere from 12 to 20+ SSDs. Much higher performance with consistently low operational latency is required of all-SSD systems, and therefore these systems are sensitive to changes in available CPU cycles.

We first evaluate CSC on all-SSD systems by running the OLTP/DB benchmark on the midrange system with an aggregate comprising 21 SSDs of 1TiB each. The aggregate was first filled to 85% of its capacity and subjected to severe load for approximately 1 day, until fragmentation metrics plateaued. Fig. 8 presents the achieved latency and CPU utilization at discrete increasing levels of load on this pre-aged dataset. The use of CSC dramatically improves write chain length, from 10.9 to 60.6 blocks, and nearly doubles the read chain length, from 2.2 to 3.7 blocks, by providing emptier AAs for write allocation; this is not shown in the figure. The CPU overhead of CSC is limited to a fraction of a core, because writes represent only a portion of the load to the system and therefore demand for clean AAs is limited. Despite the layout improvements with CSC, performance is unaffected until the maximum load is requested, where CSC reduces latency from 3.0ms to 2.6ms.

Given the variety of workloads and the prevalence of multitenancy on deployed systems, performance cannot be fairly evaluated by any one benchmark. Thus, we next target a

pure random overwrite to the same pre-aged setup as just described to increase demand for clean AAs and determine *worst-case* CSC overhead, as shown in Fig. 9. These results do not represent the expected behavior in practice, but they can inform the decision of whether to enable the feature by default. Without CSC, write chain lengths quickly degrade to 3, but with CSC they never fall below 12. Parity reads/sec without CSC are around 10 times those with CSC. However, CSC consumes significant CPU cycles in this workload—almost 3 out of the 20 cores—which results in higher latencies, especially at higher load.

Given the ubiquity of SSDs these days, all-HDD aggregates are now mostly used for backup and archival purposes. Such systems do not experience sufficient free space fragmentation to benefit from CSC, so it is disabled by default. CSC can be (and is) enabled on customer deployments that target more traditional I/O loads to achieve the benefits we describe. In hybrid SSD and HDD aggregates, CSC is enabled only on the SSD tier to provide reduced write amplification and extend device lifetimes. Finally, our results show that free space fragmentation plays a smaller role in all-SSD configurations (with sufficient SSDs) than do CPU bottlenecks. Given the expectation of consistent low latency at higher IOPS from all-SSD systems, and the higher endurance of modern enterprise-quality SSDs, CSC is disabled by default on such systems. Free space defragmentation can still be enabled on a case-by-case basis or performed by a background scan as needed at known periods of low load; Sec. 7.4 discusses this further.

7.2 File Layout Defragmentation

As discussed in Sec. 4.3, WAFL uses WAR to defragment file layouts, and that reduces the number of drive I/Os required to satisfy a client request. We use the *read chain length*—the number of consecutive blocks read by a single request to a drive—as a primary metric for measuring file fragmentation. In these experiments, we used an internal tool to generate pre-fragmented datasets on the 20-core midrange system. The tool randomizes the Physical VBNs assigned to the L_0s of a set of files, which efficiently mimics file fragmentation.

First, we analyze all-HDD aggregates. Sequential reads of 64KiB each from several clients were aimed at the system with an all-HDD aggregate. Fig. 10 presents both the average latency and read chain lengths at increasing levels of that load. Without WAR, both metrics remain stable at around 8ms and 1.8 blocks, respectively. With WAR enabled, the incoming read operations trigger WAR to improve file layout, which translates into longer read chains and reduced latency in successive load points. Overall client read throughput (not shown in the graph) also improves when using WAR, from 1GiB/sec without WAR to 2.6GiB/sec with WAR at the higher load points. At the drive level, the number of read I/Os decreases from 639 to 345 per second, in

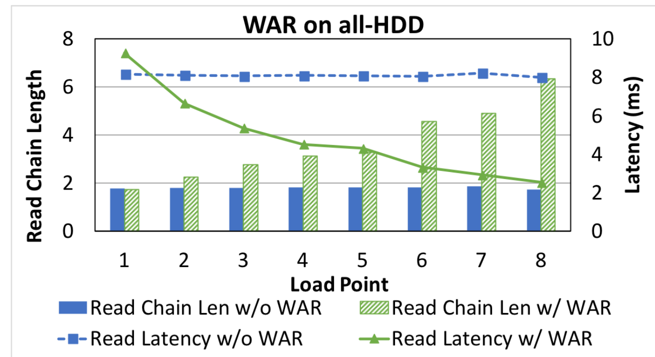


Figure 10: Read chain length and read operation latency on an all-HDD aggregate, with and without WAR. Increasing loads of sequential read are run for fixed intervals to fragmented files.

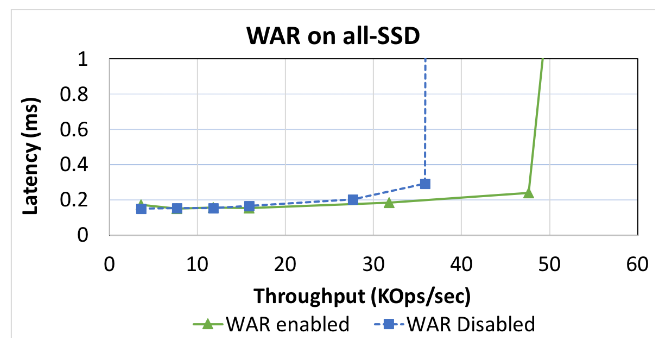


Figure 11: Latency versus achieved throughput of sequential reads to a pre-aged dataset. With WAR enabled, measurements were taken after WAR had completed defragmenting file layout.

spite of the significantly higher read throughput. This reduced load translates to a reduction in I/O latency, measured at the drive, from 3.9ms to 1.2ms. As expected, WAR writes to the storage to relocate fake dirty blocks, but the reduction in the number of drive-reads outweighs these writes.

We now evaluate WAR for all-SSD aggregates by replacing the HDDs in the previous experiment with 21 1TiB SSDs. We first isolate the benefits of file layout improvements on the performance of sequential reads by measuring throughput and latency *after* WAR has completed defragmenting the pre-aged dataset, as shown in Fig. 11. While the file fragmented test sees average read chain lengths of 1.7 blocks, post-WAR the system sees read chains of 32 blocks. Thus, file defragmentation significantly reduces SSD read I/Os per second and lowers CPU cycles needed by the storage driver code in ONTAP. As a result, with WAR, the system is capable of much higher throughput before the system saturates and latency climbs to unacceptable levels. This experiment demonstrates that file layout can still have a major impact on read performance on SSDs, even though random read performance is less of a factor than on HDDs.

As mentioned earlier, all-SSD systems are CPU-bound,

and operational latencies are more sensitive to CPU consumed by other activity. To evaluate the *worst-case* performance impact of WAR overhead and inform the enablement of this feature by default, we issued a mixed workload of sequential read and sequential write on the same fully pre-aged dataset (graph not shown). Writes are more CPU intensive than reads and so are a better indicator of CPU interference. In this test, WAR overhead pushes operation latency up from 1.7ms to 2.5ms and throughput is lowered from 2GiB/sec to 1.7GiB/sec. The WAR interference particularly comes from the increase in drive writes and 1.6 extra cores used in an already CPU-saturated system.

All-HDD backup and archival systems typically experience sequential writes (backup transfer streams) and sequential reads (restore transfer streams), and they get fragmented by the deletion of older snapshots and archives. Thus, WAR is beneficial to such deployments. WAR is disabled on all-SSD platforms due to its performance overhead, but can be enabled as needed during periods of low activity to achieve the demonstrated file layout benefits.

7.3 Compaction and Recompaction

Sec. 5 presented the compaction technology in WAFL to pack multiple compressed blocks within a single block on persistent storage. In our evaluation, we first created 2.2TiB of data across several files in the midrange system with 21 1TiB SSDs. The data written to these files was designed to be highly compressible. Once created, this data set consumed only 511GiB of physical storage, representing a 1.7TiB (77%) savings due to the cumulative effect of compression and compaction. In particular, compaction was able to store an average of 4.8 chunks per block. These were large files, so the benefit of compaction was primarily due to the tail-end of compression groups being compacted together.

The compacted dataset was then fragmented by using random overwrites until the storage savings were reduced to 27%, indicating sparsely compacted blocks and significant intra-block fragmentation. Overwrites of compressed user L_0s resulted in the freeing of chunks within compacted blocks. Then we initiated a moderate random read load (80MB/sec) with and without a recompaction scanner to measure the rate of intra-block defragmentation and the associated interference to the client workload. We observed storage space being reclaimed at a rate of 3.78GiB/min, with somewhat significant impact on client latency. In particular, we saw the average latency of the client read operations rise from 0.63ms to 2.07ms, which comes from an additional 1.25 cores worth of CPU cycles, as well as the additional blocks written to the SSD drives. The primary reason for the latency increase is a background scan[8] that runs after recompaction, but at a coarser parallelism that precludes client operations. Until that limitation is fixed and recompaction is made sufficiently lightweight, it should be initiated by the administrator only at known time periods of low load. Once recompaction is made lightweight, it can run autonomously.

[8] This scan is mentioned in Sec. 3.3, and is used to fix up stale Physical VBNs in indirect blocks of the FlexVol.

7.4 Customer Data and Summary

We mined data from customer deployments running a recent Data ONTAP release on all-SSD configurations, with different space utilization levels (aggregate fullness); Fig. 12(A) presents the distribution of write chain length observed. It shows that higher space utilization adds to the fragmentation effect caused by file system aging—higher utilization shows smaller write chain lengths. About 40% of systems that were at least 75% full have write chain lengths below 11; such systems stand to benefit from CSC. Logs collected over a recent 3-month period also showed that 17% of all-SSD systems experienced the less-efficient administrator-invoked segment cleaning (versus 7% for all-HDD systems), justifying the need for CSC. A similar analysis across all-HDD systems (not shown here) reveals that write chain length distribution skews to higher values. There are two reasons for this. One, it takes much longer to fragment an all-HDD aggregate given its lower IOPS capability, while our data was from a recent release. Two, a sizeable fraction of these systems are archival, hosting "secondary" FlexVols that are replicas of FlexVols accessed by customer applications. Incremental updates to such FlexVols [30] are slow to fragment free space because they overwrite (and free) large ranges of blocks.

To determine available CPU headroom, we next mined CPU utilization during a particularly busy hour of a weekday, but specifically for all-SSD systems with write chains less than 11. Fig. 12(B) shows that about 85% of systems had CPU utilization of less than 50%; a similar trend was seen across all-HDD systems as well. This indicates that enabling CSC would not really impact client operations. Further data mining also showed that CPU utilization of systems varies during a day, depending on the workload and the time zone, as well as the customer workflow. Several features in Data ONTAP autonomically detect periods of low user activity to selectively enable themselves; our data indicates that CSC, WAR, and recompaction can behave similarly.

To summarize this section, CSC and WAR are consistently able to deliver improved layout, and these improvements successfully translate to significant performance gains for HDD-based systems that are typically storage bound, and where layout plays a more critical role. The CPU and I/O overhead of data relocation on all-SSD systems occasionally outweighs improvements in layout, motivating autonomic defragmentation during periods of low load. Other media such as QLC flash and SMR are being analyzed. There is insufficient customer data to analyze recompaction at this time.

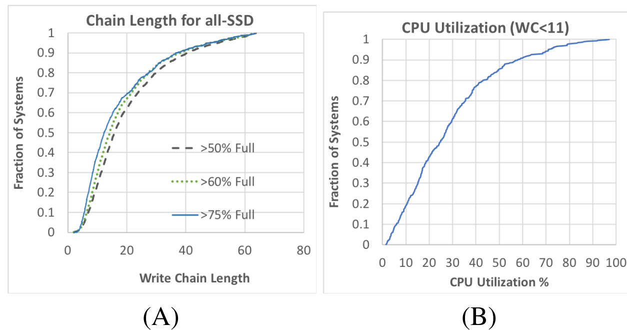


Figure 12: (A) Distribution of write chain lengths on customer deployments at different levels of fullness, and (B) Distribution of CPU utilization for systems with write chain length less than 11.

8 Related Work

The original LFS work [32] introduced log structuring and evaluated several policies for performing segment cleaning to constrain the associated overhead. Seltzer, et al. [34] analyze the performance impacts of free space fragmentation in FFS and the overheads associated with cleaning in LFS. It was shown that a policy based on grouping similarly aged blocks into new segments is efficient. Our technique targets areas that have the least cost, and because the emptiest segments (AAs) generally have the oldest blocks, cold blocks become colocated when rewritten out together. The SMaRT file system employs free space defragmentation on SMR drives [13], using either background or on-demand garbage collection based on a set of SMR-specific heuristics.

F2FS is a log-structured file system that is optimized for SSD [21]. Similar to our work, F2FS has the ability to perform both foreground and background segment cleaning and seeks to minimize the impact of cleaning on system performance. Converting random overwrites into sequential write requests in the block device driver can provide the benefits of log-structured writes without paying the cost of segment cleaning [20, 22, 43]. Geriatrix is an aging tool to fragment both files and free space [16]; it reports that free space fragmentation significantly affects file system performance on SSDs. These findings were not on enterprise-grade systems with large storage arrays and do not conflict with our results.

Sequential file read performance degrades as a file's data becomes fragmented [2, 12, 33]. Betrfs is a file system whose format inherently reduces fragmentation; the authors found that performance was sustained better over time than other file systems [5]. Aging can also be partially avoided through preallocating space for a file, multiblock allocations, and delayed allocation [24]. Unfortunately, some degree of aging is inevitable in a log-structured, copy-on-write file system [12]. The DFS file system relocates data in order to reduce fragmentation [2] and improve the subsequent read performance. Recent work has found similar negative effects of file fragmentation on mobile storage and tuned defragmentation for such platforms [11, 15]. A study of deduplication using Windows desktops showed that file fragmentation does not impact performance because a large fraction of their files are not overwritten after creation, and the background defragmenter patches up the fragmented files [26].

The problem of intra-block fragmentation is most commonly solved by using *tail packing*, in which the non-block-size aligned ends of files are persisted to a shared block [5, 31, 37]. The most popular form of this consists of defining some *fragment* size less than the block size, which becomes the minimum unit of allocation [25, 42]. ReconFS [23] dynamically compacts sub-block sized updates to metadata in order to reduce the number of drive writes on flash. Our approach is more general, in that there is no minimum or fixed chunk size. Further, we first compress data so that blocks within large files can also benefit from this technique. Our process of recompaction is similar in concept to garbage collection in the NOVA file system [38], in which log entries in nonvolatile memory are written compactly to a new log when less than 50% of log entries are active.

9 Conclusion

We investigated various forms of fragmentation in the WAFL file system, and showed that it can have significant implications on both performance (as in the cases of free space and file block fragmentation) and storage efficiency (as in the case of intra-block fragmentation). We then presented *storage gardening* techniques that leverage the FlexVol virtualization to counteract each type of fragmentation. Although the techniques dramatically improved data layout across a variety of workloads, performance gains did not universally follow. I/O-bound HDD systems showed significant benefits. However, operational latency on all-SSD storage systems is very sensitive to the availability of CPU cycles, and therefore CPU and I/O overhead of defragmentation may outweigh its benefits. Intra-block defragmentation provided significant storage savings, but with a performance penalty.

10 Acknowledgements

We thank the many WAFL engineers who contributed to these designs over the years; they are too many to list. We thank Pawan Rai for helping gather field data. We thank Keith A. Smith for his input into this work. We also thank our reviewers and shepherd for their invaluable feedback.

References

[1] NetApp cDOT - Volume Move. `https://www.storagefreak.net/2017/07/netapp-cdot-volume-move`.

[2] Ahn, W. H., Kim, K., Choi, Y., and Park, D. DFS: A defragmented file system. In *Proceedings. 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. (MASCOTS)* (2002), pp. 71–80.

[3] BARTLETT, W., AND SPAINHOWER, L. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on dependable and secure computing 1*, 1 (2004).

[4] BJØRLING, M., GONZÁLEZ, J., AND BONNET, P. LightNVM: The linux open-channel SSD subsystem. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2017).

[5] CONWAY, A., BAKSHI, A., JIAO, Y., JANNEN, W., ZHAN, Y., YUAN, J., BENDER, M. A., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., ET AL. File systems fated for senescence? nonsense, says science! In *Proceedings of Conference on File and Storage Technologies (FAST)* (2017).

[6] CORBETT, P., ENGLISH, B., GOEL, A., KLEIMAN, T. G. S., LEONG, J., AND SANKAR, S. Row-diagonal parity for double disk failure correction. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2004).

[7] COUNCIL, S. P. Storage performance council-1 benchmark. `www.storageperformance.org`.

[8] CURTIS-MAURY, M., KESAVAN, R., AND BHATTACHARJEE, M. Scalable write allocation in the WAFL file system. In *Proceedings of the Internal Conference on Parallel Processing (ICPP)* (2017).

[9] EDWARDS, J. K., ELLARD, D., EVERHART, C., FAIR, R., HAMILTON, E., KAHN, A., KANEVSKY, A., LENTINI, J., PRAKASH, A., SMITH, K. A., ET AL. FlexVol: flexible, efficient file volume virtualization in WAFL. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2008).

[10] GOEL, A., AND CORBETT, P. RAID triple parity. In *ACM SIGOPS Operating Systems Review* (2012), vol. 46, pp. 41–49.

[11] HAHN, S. S., LEE, S., JI, C., CHANG, L.-P., YEE, I., SHI, L., XUE, C. J., AND KIM, J. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2017).

[12] HE, J., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The unwritten contract of solid state drives. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2017).

[13] HE, W., AND DU, D. H. SMaRT: An approach to shingled magnetic recording translation. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2017).

[14] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter Technical Conference* (1994).

[15] JI, C., CHANG, L.-P., SHI, L., WU, C., LI, Q., AND XUE, C. J. An empirical study of file-system fragmentation in mobile storage systems. In *Procedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2016).

[16] KADEKODI, S., NAGARAJAN, V., GANGER, G. R., AND GIBSON, G. A. Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2018).

[17] KESAVAN, R., CURTIS-MAURY, M., AND BHATTACHARJEE, M. Efficient search for free blocks in the WAFL file system. In *Proceedings of the Internal Conference on Parallel Processing (ICPP)* (2018).

[18] KESAVAN, R., SINGH, R., GRUSECKI, T., AND PATEL, Y. Algorithms and data structures for efficient free space reclamation in WAFL. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2017).

[19] KESAVAN, R., SINGH, R., GRUSECKI, T., AND PATEL, Y. Efficient free space reclamation in WAFL. *ACM Transactions on Storage (ToS) 13* (October 2017).

[20] KIM, H., SHIN, D., JEONG, Y., AND KIM, K. H. SHRD: Improving spatial locality in flash storage accesses by sequentializing in host and randomizing in device. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2017).

[21] LEE, C., SIM, D., HWANG, J. Y., AND CHO, S. F2FS: A new file system for flash storage. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2015).

[22] LEE, Y., KIM, J.-S., AND MAENG, S. ReSSD: a software layer for resuscitating SSDs from poor small random write performance. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (2010).

[23] LU, Y., SHU, J., WANG, W., ET AL. ReconFS: A reconstructable file system on flash storage. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2014).

[24] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium* (2007).

[25] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for unix. *Transactions on Computer Systems 2*, 3 (1984), 181–197.

[26] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *Proceedings of the 9th USENIX conference on File and storage* (2011).

[27] MICROSYSTEMS, S. ZFS at OpenSolaris Community. `http://opensolaris.org/os/community/zfs/`.

[28] MITTAL, S., AND VETTER, J. S. A survey of software techniques for using non-volatile memories for storage and main memory systems. In *IEEE Transactions on Parallel and Distributed Systems* (Jan 2015).

[29] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the International Conference on Management of Data (SIGMOD)* (1988).

[30] PATTERSON, H., MANLEY, S., FEDERWISCH, M., HITZ, D., KLEIMAN, S., AND OWARA, S. SnapMirror: File-system-based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), USENIX Association.

[31] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (2013), 9.

[32] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10* (1992), 1–15.

[33] SATO, T. ext4 online defragmentation. In *Proceedings of the Linux Symposium* (2007), vol. 2, pp. 179–86.

[34] SELTZER, M., SMITH, K. A., CHANG, H. B. J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (1995).

[35] SMITH, K. A., AND SELTZER, M. I. File system aging—increasing the relevance of file system benchmarks. In *ACM SIGMETRICS Performance Evaluation Review* (1997), vol. 25, ACM, pp. 203–213.

[36] SUNDARAM, R. The Private Lives of Disk Drives. https://atg.netapp.com/?p=13640, 2006.

[37] WIKIPEDIA. Reiserfs. Wikipedia, the free encyclopedia, 2017. (Online; accessed 18-April-2018).

[38] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2016).

[39] XU, Q., SIYAMWALA, H., GHOSH, M., AWASTHI, M., SURI, T., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance characterization of hyperscale applicationson on nvme ssds. In *ACM SIGMETRICS Performance Evaluation Review* (2015), vol. 43, ACM.

[40] XU, Q., SIYAMWALA, H., GHOSH, M., SURI, T., AWASTHI, M., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR)* (2015), ACM.

[41] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., AND SUNDARARAMAN, S. Don't stack your log on my log. In *INFLOW* (2014).

[42] ZHANG, Z., AND GHOSE, K. yFS: A journaling file system design for handling large data sets with reduced seeking. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2003).

[43] ZUCK, A., KISHON, O., AND TOLEDO, S. LSDM: Improving the performance of mobile storage with a log-structured address remapping device driver. In *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on* (2014).

NETAPP, the NETAPP logo, and the marks listed at http://www.netapp.com/TM are trademarks of NetApp, Inc. Other company and product names may be trademarks of their respective owners.

Pay Migration Tax to Homeland: Anchor-based Scalable Reference Counting for Multicores

Seokyong Jung, Jongbin Kim, Minsoo Ryu, Sooyong Kang, Hyungsoo Jung*
Hanyang University
{*syjung, jongbinkim, msryu, sykang, hyungsoo.jung*}@*hanyang.ac.kr*

Abstract

The operating system community has been combating scalability bottlenecks for the past decade with victories for all the then-new multicore hardware. File systems, however, are in the midst of turmoil yet. One of the culprits behind performance degradation is reference counting widely used for managing data and metadata, and scalability is badly impacted under load with little or no logical contention, where the capability is desperately needed. To address this, we propose PAYGO, a reference counting technique that combines per-core hash of local reference counters with an anchor counter. PAYGO imposes the restriction that decrement must be performed on the original local counter where the act of increment has occurred so that reclaiming zero-valued local counters can be done immediately. To this end, we enforce migrated processes running on different cores to update the anchor counter associated with the original local counter. We implemented PAYGO in the Linux page cache, and so our implementation is transparent to the file system. Experimental evaluation with underlying file systems (i.e., `ext4`, `F2FS`, `btrfs`, and `XFS`) demonstrated that PAYGO scales file systems better than other state-of-the-art techniques.

1 Introduction

Reference counting is a general technique, originally introduced by Collins [8] almost six decades ago, to determine the liveness of an object for automatic storage reclamation. Since the early version of UNIX kernel used reference counting to manage data (e.g., page cache) and metadata (e.g., inode), reference counting has gained widespread acceptance in the systems community thereafter, e.g., file systems, HBase [21], RocksDB [22] and MariaDB [23].

However, a recent study by Min et al. [17] found that reference counters, among many other factors, in modern file systems are not scalable, thus leading file systems to suffer performance degradation on multicore hardware, even with applications with little or no logical contention. For example, the traditional way of referencing (let us call it 'traditional reference counter'), which is currently being used in the Linux operating system for page cache, uses a single shared atomic counter. By using atomic operations, an object can be safely referenced even when multiple threads update at the same time. The traditional reference counters, however, degrade the performance of applications on multicores due to excessive atomic operations on a shared counter.

In order to be a good reference counter for concurrent applications, there are important properties to consider; 1) updates on reference counters must be scalable, 2) reading an accurate (zero or positive) counter value should be cheap, 3) reference counters should be space-efficient and 4) all these should be guaranteed without incurring extra delay to manage reference counters. We denote overheads required for achieving the four properties as *counting overhead*, *query overhead*, *space overhead*, and *time overhead*, respectively.

Counting overhead. The counting overhead, which is the most important property for scalable counting, indicates the cost of updating (REF/UNREF) a reference counter itself when there is a heavy load on referencing an object. Since the counting overhead is a crucial hurdle for achieving scalability, all reference counting techniques strive hard to eliminate it first. The traditional reference counter which uses a single shared counter has the highest counting overhead due to the hardware-based synchronization bottleneck [13].

Query overhead. The query overhead measures the cost of query operation which checks if the reference counter of an object is zero and so we can safely reclaim the object from memory. The traditional technique can detect zero by reading a single atomic counter.

Space overhead. The space overhead indicates how much space they use for reference counting. In terms of space overhead, the traditional reference counter is a (de facto) optimal technique since it does not require any other data structure than one atomic counter per object.

Time overhead. The time overhead represents any other delay than the counting overhead introduced by a reference

*Contact author and principal investigator

counting technique to manage all data structures it maintains. The traditional reference counter has minimal time overhead since it maintains only per-object atomic counters. However, some reference counting techniques that exploit distributed local reference caches have the synchronization overhead between local counters and a global counter. This synchronization plays two roles: 1) the global counter becomes ready (i.e., up-to-date) for zero detection and 2) the local counter, if it resides in hash, can be reclaimed. We generally denote this type of overhead as the *time overhead*.

Our analysis of prior proposals (§2.1) suggests that it is challenging to achieve all four properties, possibly due to tradeoffs between different properties. In this work, we propose *pay-as-you-go* (PAYGO[1]) reference counting that ensures scalable counting and space efficiency with negligible time overhead. Although based on a well-known per-core hash technique, PAYGO introduces a novel concept of an *anchor counter* that enables the immediate reclamation of local zero-valued counter entries, which is pivotal to reducing the forceful eviction of the conflicting hash entries when the number of objects accessed in a core becomes large. The instant reclamation is indeed a critical feature for escaping performance degradation that may otherwise occur due in large part to the heavy cost of operations for resolving collisions, including forceful evictions.

We implemented PAYGO and applied it to page cache in Linux, leaving existing file systems almost intact. To see the applicability of PAYGO to user applications, we also implemented new PAYGO system calls that can be used for reference counting user-level objects. Evaluation results with various underlying file systems (i.e., `ext4`, `F2FS`, `btrfs` and `XFS`) demonstrated that PAYGO shows substantial improvements against state-of-the-art reference counting techniques.

2 Related Work and Motivation

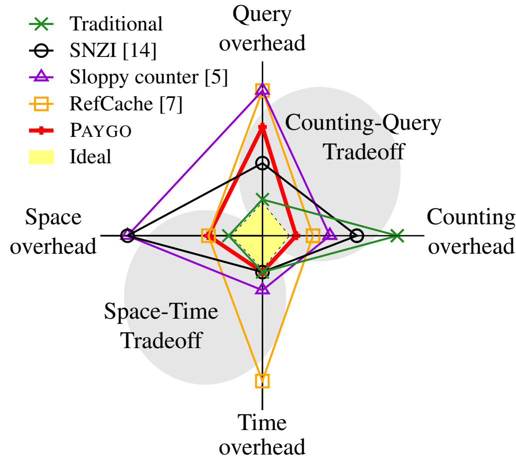
2.1 Related Work

There have been many proposals attempting to address some of the properties introduced in §1, and the techniques available so far utilize at least one of the following features:

Contention distribution. One of the major factors impeding the scalability of a reference counter is cache line contention: updating the same reference counter *atomically* by many threads results in high contention. SNZI [14] mitigates the contention by dispersing the reference counters at compile time. It manages distributed counters using a binary tree with a fixed-sized depth. While it shows better scalability than the traditional reference counter, it is still slower than other techniques due to the possible contention on a particular counter that changes frequently. However, it can perform zero detection in constant time by checking the indicator of the root node in the binary tree, although determining the global count value is impossible. Other techniques [1, 18] alleviate the contention problem by distributing reference counters according to the degree of contention at runtime, but they empirically judge the degree of contention and distribute reference counters so they cannot relieve the contention for a reference counter under general workloads where we can hardly predict the degree of contention. Carrefour [12] also distributes contention dynamically, but hardware profiling is required to verify memory traffic. Proposals in this category still rely on atomic instructions for updating reference counters, so they seldom achieve linear scalability.

Cache affinity. Another factor that hinders scalability of reference counters is cache misses. To reduce the cache misses, a local reference counter is used in a way that an object has per-core local counters and updates are made to the local counters *nonatomically*. The downside of this approach is the overhead needed for summing all local counter values to obtain the global count. To alleviate this side effect, there is a way to obtain the global count in advance and store it in the central counter [9, 5, 10], which incurs extra time overhead. Sloppy counter [9, 5] updates only the local counter if the updated value is less than a certain threshold. If the value exceeds the threshold, the local counter value is transferred to the central counter. The central counter is therefore an approximation of the global count. Before transferring the local counter value, it acquires the global lock for the central counter, which incurs extra time overhead. `percpu_ref` [10], a variant of the sloppy counter, implemented in Linux for managing memory objects in several device drivers, also primarily changes the local counter. The techniques exploiting cache affinity have the *counting-query tradeoff*: nonatomic updates on local reference counters earn good scalability in exchange for longer query time to read a global count by collecting the sum of local counters. They also have bad space efficiency due to the per-object, per-core local counters.

Per-core hash. To improve the space overhead of cache affinity-based techniques, recent years have seen attempts to use per-core hash of reference caches that would fulfill the main duty of reference counting with much less space overhead [7, 4, 3]. They can substantially reduce the space overhead by using per-core hash which keeps the local reference counters for only those objects in use. Techniques based on per-core hash inevitably face the problem of reclaiming a hash table entry whose local counter is zero (i.e., the corresponding object is not in use). Existing techniques address this using quiescent period-based synchronization which is widely used in Linux to reclaim objects, such as *read-copy-update* (RCU) [16]. The reference counting algorithm exploiting per-core hash with quiescent period-based synchronization cannot avoid the *space-time tradeoff*: they achieve better space efficiency in exchange for time overheads not only in synchronization between local and global counters but also in hash entry reclamation. RefCache [7], which is one of quiescent period-based techniques, manages its local counters in per-core hash, and the counter values are flushed



	Traditional*	SNZI	Sloppy counter	RefCache	PAYGO
Counting overhead	atomic ops.	atomic ops.‡	global lock	–	–
Space overhead†	$O(N)$	$O(M \cdot N)$	$O(M \cdot N)$	$O(M \cdot C + N)$	$O(M \cdot C + N)$
Query overhead§	$O(1)$	$O(1)$	$O(M)$	$O(1) + 2 \cdot epoch$	$O(M)$§§
Time overhead	–	–	every threshold	every epoch and collision	–

* A single atomic reference counter
† N: # of objects, M: # of local counters per object, C: # of hash entries
‡ SNZI recursively updates the counter of the parent node whenever the counter of the child node changes from 0 to 1 and vice versa.
§ Time to determine if the reference counter of a *single* object is zero or not
§§ PAYGO has practically less query overhead than Sloppy counter (§3.4).

Figure 1: A comparison of reference counting techniques under workloads accessing a shared counter.

into a central counter every epoch. OpLog [4] generalized RefCache's idea by using operation logs for the shared data structure with a local timestamp. In descending order of the timestamp, per-core logs are applied to the data structure.

2.2 Motivation

The design of PAYGO is motivated by two observations:

Observation 1. Our analysis of existing algorithms in §2.1 is summarized in Figure 1. Noticeable is the observation that attaining the counting scalability (i.e., low counting overhead) demands a sacrifice of two other properties due to the counting-query and space-time tradeoffs. By escaping those tradeoffs we can attain more good properties; for example, escaping the space-time tradeoff enables us to make both space and time overheads low while achieving scalability.

Observation 2. Another and more important observation is that the excessive time overhead may eventually incur severe performance degradation. As described in §2.1, techniques based on per-core hash sacrifice time overhead to reduce space overhead. The time overhead under consideration in such techniques is the overhead of reclaiming hash entries, when the number of objects accessed in a core becomes large so that frequent hash collisions occur and therefore forceful evictions for the conflicting hash entries need to be exercised to make room for newly accessed objects. The eviction of a hash entry needs to flush the local counter value to the global counter and therefore causes additional synchronization overhead between local and central counters.

For example, RefCache [7], designed for a new virtual memory system, is perfectly scalable when n threads are repeatedly performing `mmap/munmap` on a single shared physical page (see Figure 8 in [7]), since the forceful eviction of hash entries due to collisions seldom occurs and so may not be a serious design consideration in virtual memory systems. However, if we use RefCache in *page caches* under file system benchmarks that may access far more objects, frequent evictions that internally acquire/release object locks to protect the flush of local counter values to the central counters, may result in serious time overhead, leading to performance degradation. To confirm it, we conducted a preliminary experiment that measures the throughput of RefCache for page caches. The experimental environment is shown in §6.1, and we ran the FXMARK microbenchmark so that 96 threads read 64 bytes (or 4 KiB) on a shared file, with a sequential access pattern and varying the file size. Figure 2 shows the result that confirms our conjecture. The throughput decreases as the file size (i.e., the number of objects) gets bigger, due to increased hash collisions triggering more forceful evictions. We found that the hash collisions start slightly occurring from the point when the file size is 1 MiB (i.e., 256 objects) and become excessive as the file size increases. Hence, reclaiming garbage hash entries in a timely manner is critical for avoiding the performance degradation of per-core hash-based reference counting techniques.

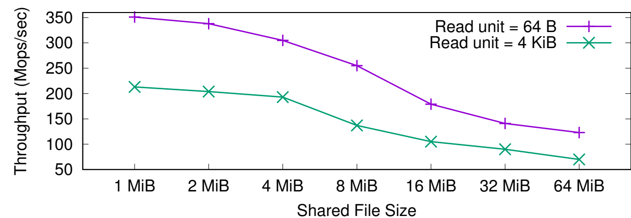


Figure 2: Performance of RefCache: hash table size = 4,096 entries (default size), `ext4` file system.

The above observations guide us to conclude that escaping the space-time tradeoff is crucial for scalable reference counting techniques. By escaping the space-time tradeoff, we can achieve true scalable counting keeping both space and time overheads low, which is our design goal of PAYGO whose comparative properties are depicted in Figure 1.

3 PAYGO

Of counting-query and space-time tradeoffs, we aim at escaping the *space-time tradeoff* while embracing the other. PAYGO achieves this by using a per-core hash-based reference cache with a new technique called *anchoring*. PAYGO is designed on the following assumptions; (i) objects are referenced and unreferenced by the same process and (ii) the lifetime of references is reasonably short not to put the static size of per core *hash* in jeopardy (see §3.5).

3.1 Design Overview

Design rationale. To escape the space-time tradeoff, per-core hash-based techniques must ensure the safety condition such that a local reference cache entry can be reclaimed immediately upon releasing all references to an object, all done without sacrificing other properties. In this respect, RefCache earned the counting scalability in exchange for the increased time overhead required for reclaiming obsolete cache entries. For addressing this issue, our main design rationale behind PAYGO lies in a simple goal; we make a local reference cache zero (i.e., ready to be reclaimed) right after all references are released. To this end, we enforce the restriction that a process, once referencing an object, must be *anchored* to the original reference cache to inform of any unreferencing to the object irrespective of which core the process runs on. For this purpose, PAYGO's per-core hash entry consists of a local counter and an *anchor* counter fields, and the sum of two represents a local count for an object initially accessed in that core.

Access rules. First, we establish ground rules in accessing a pair of local and anchor counters to preserve the correctness, that is '*never miscount*'. Access rules for `(UN)REF` are described as follows. For the `REF` operation, a process always accesses a local reference cache and updates the local counter field nonatomically. At this time, the process is logically anchored to this core (homeland) and anchor core ID is recorded in a `task_struct`. For the `UNREF` operation, acting on local or anchor counters depends on whether migrated or not in between `REF` and `UNREF`; if the process remains at the same core (homeland), `UNREF` is done on the same local counter nonatomically. Otherwise, the migrated process *atomically* updates the anchor counter of the original reference cache in the homeland core. The use of an atomic operation on an anchor counter is indeed for correct counting even with multiple processes in concurrent environments.

We summarize the access rules in Table 1, and we ensure that a local reference cache becomes zero upon the completion of `REF`/`UNREF` operations. This allows PAYGO to reclaim zero-valued local reference caches immediately from hash, thus retaining the hash space efficiency without time overhead. The common rule governing both `REF` and `UNREF` is, we disable *preemption* while performing two operations in

Table 1: Access rules for `REF` and `UNREF` from homeland and foreign land. (✓: *nonatomic*, ⓥ: *atomic*, ×: no-op)

Type	PAYGO Entry			
	local counter		*anchor counter*	
	`REF`	`UNREF`	`REF`	`UNREF`
Homeland	✓	✓	×	×
Foreign land	×	×	×	ⓥ

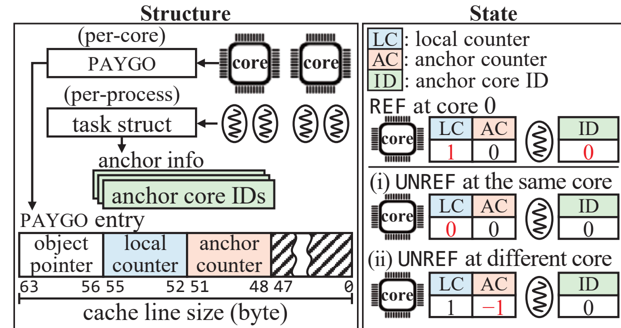


Figure 3: An overall structure of PAYGO and state of the structure when referencing and unreferencing an object.

order to prevent *malicious data races* on a local counter. Of course, there are other ways of doing this, such as kernel spin locks (i.e., `spinlock_t`), but disabling/enabling preemption is by far the fastest method we found it suitable for our purpose and has been used in prior work [7, 10]. An in-depth performance analysis will be presented in §6.2.5.

Overall structure of PAYGO. Next, we describe the overall structure of PAYGO. Figure 3 shows the structure of PAYGO and the state of the data when an object is referenced and unreferenced. For each core, there is per core hash of reference caches, each entry of which consists of an object pointer and two counters, a *local* and an *anchor* counters. The space overhead for this hash table is much smaller than the Linux sloppy counter and larger than the traditional one, but it is similar to RefCache. Given hash of reference caches, the `UNREF` operation atomically decreases the anchor counter of the anchored core only when process migration occurs, by the access rules in Table 1. To do this, each process stores anchor information that bookkeeps the core IDs in which an object is referenced. The anchor information internally maintains multiple anchor core IDs to deal with a case where a process references an object multiple times on different cores without unreferencing it. Matching anchor core ID is removed after `UNREF` is done on the corresponding core. Unlike hash, PAYGO requires extra memory space for storing anchor information in a task structure, and this is surely regarded as additional memory overhead.

On the right side of Figure 3 shows the state of the data when a process references and unreferences an object. When a process references an object at core 0, it raises the local counter of core 0 and keeps core 0 in the process's anchor

information. When the process unreferences the object, it first searches the current core ID in its anchor information. If found, the process decreases the local counter of the current core; otherwise, it decreases the anchor counter of any core in the anchor information *atomically*.

3.2 PAYGO Operations

PAYGO has three operations: REF/UNREF operations to increase/decrease a reference counter and READ-ALL operation to read the global value of the reference counter which is equivalent to the query operation.

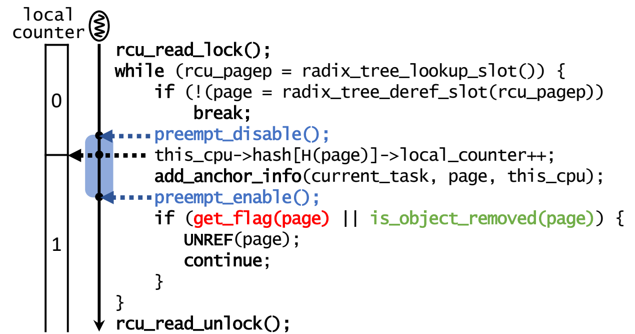
REF **operation.** When a REF operation of an object is invoked, it finds the PAYGO entry for the object in the hash of the current core. If the PAYGO entry is found, its local counter is increased. If the PAYGO entry is not found, a new PAYGO entry is created in the hash and the local counter is increased, and then the current core ID is stored in the process' anchor information. The REF operation is executed while preemption is disabled to prevent multiple processes from updating the same hash entry concurrently.

UNREF **operation.** When an UNREF operation of an object is invoked, it first checks the anchor information of the process. If the core ID stored in the anchor information is the same as the current core, the process finds the PAYGO entry for the object in the hash of the current core and decreases the local counter. If the process has migrated to another core, it finds the PAYGO entry for the object in the hash of the anchored core and atomically decreases the anchor counter. The UNREF operation is also performed with preemption being disabled for the same reason as the REF operation.

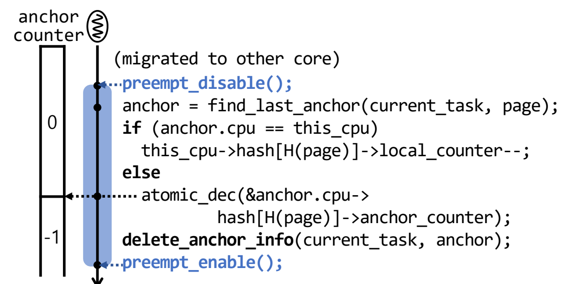
READ-ALL **operation.** When a READ-ALL operation is invoked, it finds all the PAYGO entries for the object in all per-core hash tables and computes the sum of the local and the anchor counters of all valid PAYGO entries. The READ-ALL operation is performed while the preemption is disabled in order to prevent any scheduling delays that may slow down the process. Nevertheless, this does not guarantee to read the correct sum since the REF and the UNREF operations may modify the counters during the READ-ALL operation. Object reclamation therefore needs a delicate design (§3.3).

3.3 Object Reclamation

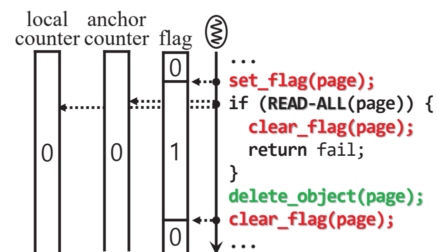
Objects are a target of reference counting, and operating systems often reclaim objects that are not referenced by any process in order to keep memory pressure under control. Once an object is chosen to be reclaimed, the reclaiming process should prevent any additional reference to the object and check again the zero value of the reference counter. In traditional reference counting, this can be done atomically by comparing the shared atomic counter with zero and swapping it to a negative value. The synchronization used in the traditional method is based on *atomic read-modify-write* operation (e.g., CMPXCHG).



(a) REF operation



(b) UNREF operation



(c) READ-ALL operation (reclaimer)

Figure 4: Code snippets of how PAYGO's REF, UNREF and READ-ALL are implemented and used in the Linux page cache, where $H()$ is a hash function.

In PAYGO, it needs more steps to correctly handle the case. Since the READ-ALL operation cannot acquire the sum in one snapshot, it uses a flag to indicate its commencement, which helps prevent the additional reference to the object. The synchronization method we use here is based on the *read-after-write* (RAW) pattern [2]. Important to notice is the invariant that at least one of a reclaiming process and referencing processes, if they run concurrently, must detect both events and then retreat itself for safety, thus never allowing malicious data race. We enforce these checking conditions to be verified at the end of REF and READ-ALL routines.

Figure 4 shows the code snippets of how REF, UNREF and READ-ALL operations are implemented in the Linux page cache with a special flag indicating that the current page is accessed for reclamation. Accessing the special flag may

cause contention only if the same page is repeatedly reclaimed (or flushed in the Linux page cache) while many processes read it, which we seldom, if ever, witnessed in Linux. In Figure 4a, the code executed while preemption is disabled denotes the `REF` operation. In Figure 4b, the whole code is the `UNREF` operation. `READ-ALL` operation which iterates all core's hash, finds the PAYGO entry and collects the sum of all entries again with the preemption being disabled, is only shown as a function call in Figure 4c. Notice that there is additional code around the `REF` and `READ-ALL` operations for the correct implementation of reclaiming page caches.

As shown in Figure 4a, the entire routine for referencing a page is protected by `rcu_read_lock()` and `rcu_read_unlock()`. The `REF` operation starts by obtaining an `rcu` reference to the page. Once it obtains the `rcu` reference, it retrieves the page object and then performs the `REF` operation. After that, a flag is checked to see if a reclaiming process is being tried. If the flag is clear, then the page is checked whether or not it is removed. This makes sure that the page is not already reclaimed before the flag is checked. Only if both conditions are passed, the page is safely referenced. Otherwise the flag is set, then the process retries until the reclaiming process clears the flag. If the page was already removed, the reference process fails. For the reclaiming process, the `READ-ALL` operation is performed after setting a flag. If the page is not referenced by any thread, the page object is safely removed and the flag is cleared. If the page is already referenced by other thread, it is not removed and the flag is cleared, thus failing to reclaim the page object.

3.4 Anchoring in Action

Reference counting techniques exploiting per-core hash, such as RefCache [7], allow processes to update nonatomic local counters of the running core. This means that when a process at core 0 increases the local counter of core 0, migrates to core 1, and decreases the local counter of core 1, then we have two local counters with values of 1 and -1, respectively. The spread-out local counters are problematic if per-core hash is used to reduce space overhead. As an example, RefCache uses background threads to flush the local counters every epoch, which inevitably delays the reclamation of zero-valued reference cache entries.

The anchoring technique in PAYGO enforces the `REF` and `UNREF` operations to act on the same PAYGO entry, thus guaranteeing the sum of its local and anchor counters to eventually become zero. Any zero-valued PAYGO entry can be recycled immediately when another `REF` operation accesses the same hash bucket. Figure 5 shows an example of an object accessed by multiple threads in a system with four cores. At core 0, a red thread references and unreferences the object by increasing and decreasing the local counter of core 0. At core 1, a blue thread followed by a green thread reference the object. Then, a yellow thread also references the object at core

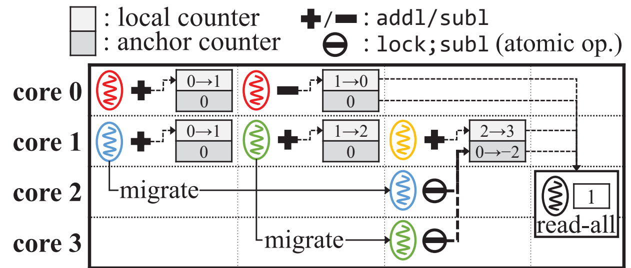


Figure 5: Usages of an anchor counter.

1. Since the yellow thread is using core 1, the blue thread and the green thread have to migrate to other cores (namely, core 2 and core 3, respectively), and unreference the object using the same anchor counter of core 1. As shown in this example, an anchor counter has the risk of being modified by multiple threads in parallel, so we use an atomic operation.

The anchoring technique gives us another opportunity of reducing the query overhead. Since the sum of local and anchor counters in a core can never be negative, during the zero detection (i.e., query), upon seeing a positive value of the sum in a core, we can immediately stop zero detection safely concluding that the object is currently being used by at least one process.

Discussion. Since decreasing an anchor counter uses an atomic operation, there is a performance concern when systems have processes that are all accessing the same anchor counter, thus hitting the hardware-based synchronization bottleneck. This is the worst case that can occur when processes are migrated frequently between `REF` and `UNREF`. But, the general design rationale for the OS scheduler usually inhibits such frequent process migration unless there are compelling reasons, such as severe load imbalance.

Nonetheless, the chance of migrating a process can increase if the interval between `REF` and `UNREF` becomes distant. Even if it occurs, atomic operations on anchor counters would not have bad impact on performance, since the price for process migration is much larger than the pure cost of reference counting itself. Hence, the performance degradation caused by atomic operations can be neglected (see anchoring overhead in §6.2.3). To alleviate any possible bottleneck on the same anchor counter, the OS scheduler can give a temporary CPU affinity to processes that are in between `REF` and `UNREF`, to prevent process migration.

One may raise concern about the overhead of searching the matching core ID in anchor information when a process references an object multiple times or numerous objects without unreferencing. Since PAYGO stores the same anchor ID in anchor information even if the same object is referenced again, this issue will surely impact performance due to the search cost, but we have not discovered such cases yet inside file systems or data management systems. If the case is found, then augmenting an additional search structure must be necessary.

3.5 Table Overflow

The table overflow problem of hash tables is a fundamental issue that per-core hash-based counting techniques should address. In the context of reference counting, the table overflow occurs when there are a large number of live objects. For instance, if a process opens many files, then the corresponding `dentry` objects will be alive in per-core hash until closed. Conventional methods, such as table doubling, are hard to use or to be efficiently designed due mainly to high concurrency. We deal with the overflow similar to the way Linux swap space is managed. First, an object that uses PAYGO has a list, called an *overflow counter list*, protected by an object lock. When a live object needs to be evicted from per-core hash, we acquire the object lock, evict the entry from hash, add the evicted counter information to the overflow counter list and then release the lock. Later, the owner process of the evicted entry can reload the evicted counter information from the overflow list while holding the object lock. Further improvements can be made to the shared overflow list, but we hold off until it really matters since '*premature optimization is the root of all evil*' [15]. What really matters here is the lifetime of the referenced object, and the concerned place (i.e., page cache) suffering bottlenecks has short-lived objects that begin and end its lifetime inside `read`/`write` system calls. PAYGO scales file operations well under such conditions.

4 PAYGO Implementation

We implemented PAYGO in Linux kernel version 4.12.5 and applied it to the page cache that can affect many concrete file systems suffering scalability issues. For experiments, we take the code base implementing RefCache and SNZI from sv6 [6] and adapt it to Linux page cache. Noticeable is the observation that other latent contention often arises after PAYGO eliminated contention on reference counters.

The Linux page cache is implemented using a radix tree, and its operations are made lockless for the performance benefits [20]. However, `read` system calls using a page cache still have scalability issues, such as the usage of atomic reference counter to synchronize between reading a page from a page cache (`REF`/`UNREF`) and flushing the page from memory to storage (`READ-ALL`). Therefore, threads trying to read the same page contend on the same reference counter and have poor performance under such loads [17].

In the original implementation of a reference counter in a page (`_refcount`) has two purposes. First, it is used as a status variable. If the value is zero, it means that the page is unused. If the value is two, the page is active and is stored in a page cache, but it is not referenced by any threads. The `_refcount` of a value above two is used as a reference counter. For example, the `_refcount` value of three indicates that there is one process referencing the page. Here, we left the `_refcount` to be used as a status variable and use PAYGO to replace the referencing part of `_refcount`.

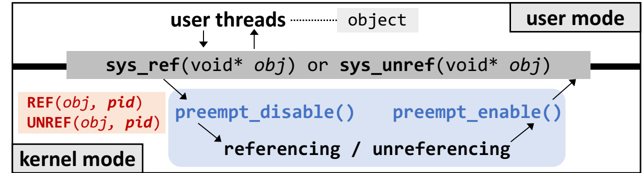


Figure 6: User-level PAYGO in Linux.

5 User-Level PAYGO

PAYGO, although being motivated by pressing concerns in file systems and intended to address it, can be easily extended to a user-level reference counting method for applications above the kernel. The development of scalable user-level reference counting is more demanding indeed, since there are many latent use cases where contention may arise once the present performance matters are all cleared away. For example, managed language runtimes (such as the JAVA runtime) use referencing counting for collecting garbage objects, and the same is true in the database land; popular database systems, such as HBase [21], RocksDB [22] and MariaDB [23], also use reference counting for managing memory objects. To the best of our knowledge, they use either hardware based atomic operations or lock based synchronization primitives to safely orchestrate concurrent accesses to the shared counter variables, but both methods are all vulnerable to performance bottlenecks in highly concurrent environments.

To make applications benefit from the better scalability of PAYGO, we implement three system calls, `sys_ref()`, `sys_unref()` and `sys_readall()`, which enable applications to exploit core kernel-level PAYGO operations without difficulty for user-level reference counting (Figure 6). Despite there being the inherent overhead required in switching between user mode and kernel mode, reference counting on user-level objects through PAYGO system calls enables applications to achieve far better scalability than their legacy reference counting techniques. Furthermore, PAYGO will exhibit much less overhead for managing garbage entries in per-core hash, and this is a required feature especially when applications hold a large number of live references.

Enabling applications to directly exploit the reference counting technique in the kernel via system calls poses two nontrivial issues. First, the system call overhead should be sufficiently minimized to benefit from the original performance of the kernel-level reference counting technique. To this end, we make PAYGO system calls lightweight such that they just wrap core kernel-level PAYGO routines with `preempt_disable()`/`preempt_enable()` executed beforehand/afterward. The wrapped routines here basically refer to the code segments bounded by `preempt_disable()` and

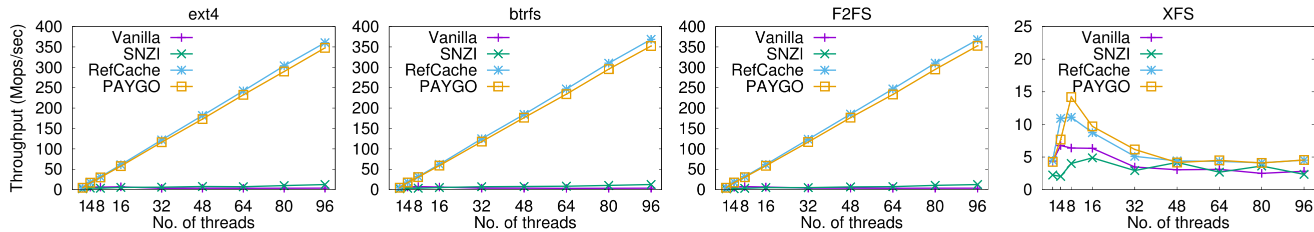


Figure 7: Scalability comparison under strongly contending workloads: the Linux page cache.

`preempt_enable()` in Figure 4. One subtle matter is, we have to transform the virtual address of a user object into a unique one inside per-core hash, by combining it with the `pid` of a user process. Second, since applications are not as reliable as kernel, the abnormal termination of applications may leave the kernel data structures for reference counting incorrect. When an application terminates after referencing an object but before unreferencing it, the corresponding counter in the kernel can never become zero. To resolve the problem, when terminating a process, the `task_struct` needs to be checked to detect any left-over counter values in the corresponding PAYGO entries in per-core hash tables. Any such left-over counters, if found, must be decreased.

6 Evaluation

In this section, we measure the overall performance and scalability of PAYGO, especially in page cache, and compare with other reference counting techniques including RefCache, SNZI and traditional reference counter under various file systems. For analyzing the performance of user-level PAYGO, we compare PAYGO with existing user-level reference counting techniques.

6.1 Experimental Setup

We perform all of the experiments in Linux kernel version 4.12.5 on our 96-core system equipped with four 24-core Intel Xeon E7-8890 v4 CPUs and 1 TiB DDR4 DRAM. We run FXMARK microbenchmark [17] with a RAM disk and filebench [11, 25] with a Samsung SM1725 NVMe SSD. To show the general applicability of PAYGO, we conduct experiments under four different file systems (i.e., `ext4`, `btrfs`, `F2FS` and `XFS`). In `ext4`, we used the default *journaling* mode and did not see any lock contention in the journaling subsystem observed in the prior study [17]. Page structures cached in memory are freed before every experiment, and the Linux security module is turned off to avoid the unrelated performance degradation.

6.2 Scalability

This section explores the multicore scalability of concerned file systems under file system benchmarks, with the degree of contention being varied from strongly contending to weakly contending. Our evaluation methodology follows similar approaches used in [17], and the important metric is the number of `REF/UNREF` (i.e., file reads) with the degree of contention on reference counters being controlled by the size of files accessed by benchmark threads. Experiments under this controlled environment may reveal the weakness and strength of tested schemes that may overlook at the time it was proposed.

6.2.1 Strongly Contending Workloads

To evaluate the performance of file operations under strongly contending scenarios, we ran the shared block read workload (i.e., DRBH) in FXMARK, a microbenchmark that is intended to stress file systems. For the evaluation, a varying number of threads repeatedly read the same 4 KiB data block, thus stressing the reference counting part enormously. This workload is known to reveal the contention resilience of any reference counting approach, since the stock Linux suffers the most. Figure 7 shows the results. With this workload, all file systems under consideration in stock Linux (i.e., vanilla) undergo severe scalability bottlenecks arising from contention on the same reference counter. SNZI shows slight improvements over the vanilla scheme that uses the traditional reference counter. PAYGO and RefCache perfectly scale the throughput of `ext4`, `btrfs` and `F2FS`. The main reason for slightly lower performance of PAYGO than RefCache is because the number of instructions executed by PAYGO is slightly greater than RefCache. By profiling on clock cycles, we obtain the cycle difference that matches the performance difference we observe here.

Interesting is the performance degradation that has been consistently observed in `XFS` primarily due to contention on the semaphore inside an `inode` structure, which completely renders all reference counting methods useless. Although a further investigation is needed, it is worthwhile putting effort to redesign this coarse-grained locking so that `XFS` can reap performance benefits from better counting techniques.

6.2.2 Weakly Contending Workloads

To evaluate the performance of file systems under weakly contending scenarios, we used filebench, a benchmark that

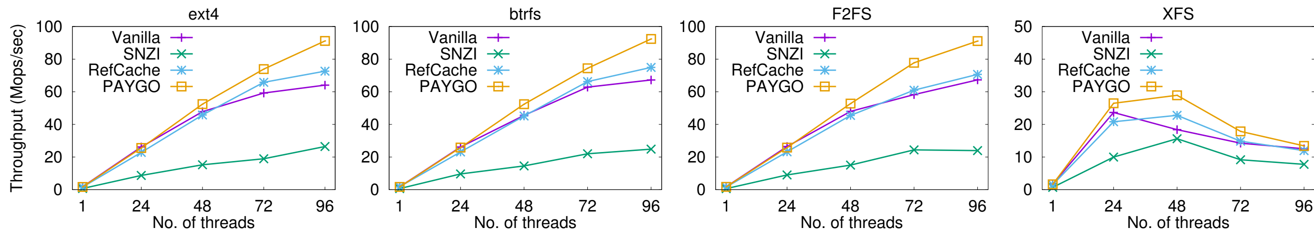


Figure 8: Scalability comparison under weakly contending workloads: the Linux page cache.

can flexibly add and test workloads to file systems and storage. Before we run the filebench, we modified the filebench code to experiment with more flexibility on multicores. Originally, filebench is implemented with a lock for each file and only one thread can access the file at a time. We eliminated the file lock so that multiple threads can access the file concurrently. For the evaluation, we ran the randomread workload with participating threads performing 64 bytes random reads from one of ten 128 MiB files. Since weakly contending workloads disperse contention on reference counters, it may expose any latent overhead (or downside) of given counting techniques that has been overlooked in exchange for resolving high contention arising under strongly contending scenarios.

The throughput results are shown in Figure 8. Strikingly the vanilla scheme deployed in the stock Linux page cache performs well after it reduces hotspot contention; it outclasses SNZI all the time and sometimes outperforms RefCache with a slight margin. As the thread count increases, the throughput gap between PAYGO and RefCache widens due to a large number of garbage entries that increase hash collisions in RefCache's per-core hash, which were not observed under strongly contending workloads.

Again, none of the tested counting techniques scale the performance with `XFS` due to bottlenecks inside `XFS`, and this will be discussed in detail in the following section.

6.2.3 In-Depth Analysis

In order to reveal detailed information about various system activities, we perform an in-depth analysis with moderately contending workloads being profiled over different metrics.

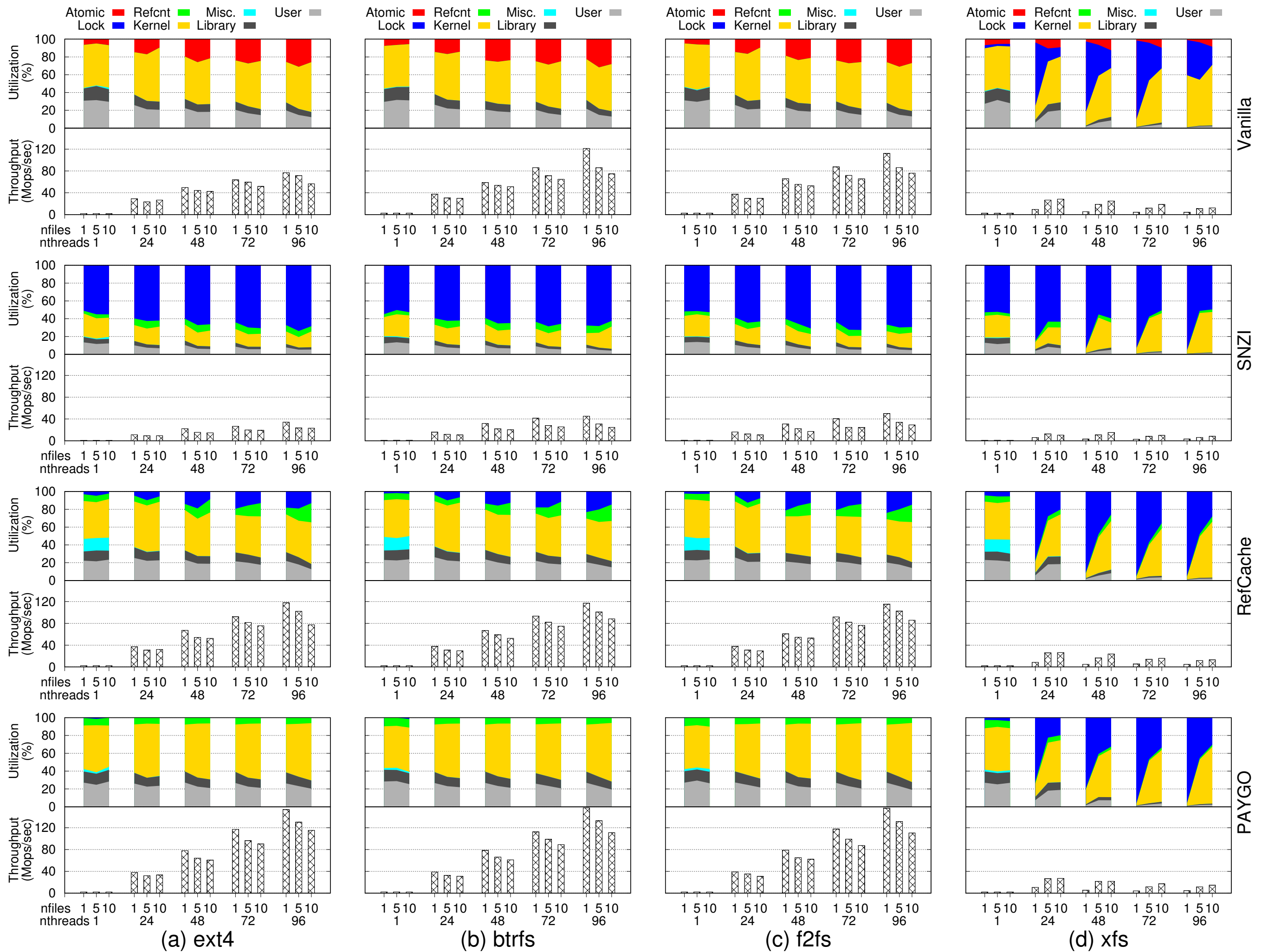
Stressing page cache. We first ran the randomread workload of the filebench microbenchmark, by varying the number of files whose size is set to 32 MiB. We chose the moderately contending workload as a good proxy for stressing reference counting schemes with a reasonable balance of contention and the count of objects referenced. Figure 9 shows the throughput and the CPU breakdown of the benchmark.

First, the in-depth profiling gives clear explanations for two strange observations in XFS and SNZI. The first observation is the poor scalability of XFS. The main culprit for this problem is due to severe lock contention inside the file system; `xfs_ilock()` and `xfs_iunlock()` on the `inode` of a file. Lock contention mainly depends on the number of files, not the file size. High contention on the `inode` lock indeed leads to severe performance degradation regardless of reference counting schemes. This perhaps needs attention from our community. The second observation is the poor scalability of SNZI, and SNZI also has a similar culprit for the issue; it scales poorly regardless of file systems at this time. Since the only publicly available code base for SNZI can be taken from sv6 [6], we show the results as is.

The vanilla scheme can scale the performance of `ext4`, `btrfs` and `F2FS` quite well as the thread count increases. Although the overhead of atomic instructions grows in proportion to the thread count, the dispersed contention cancels out the negative impact of atomic operations we have seen in Figure 7. With 72 threads, the vanilla scheme performs almost on a par with RefCache. An in-depth analysis of performance over different contention levels will be discussed in the next experiment.

RefCache shows worse core scalability than PAYGO, and this is mainly because of the increased overhead of handling hash collisions (i.e., atomic lock operations) in RefCache. Also, noticeable is the slight performance degradation of the vanilla, RefCache and PAYGO as the file count increases. This is due to the increased memory access overhead for reading files larger than cache memory, which is not observed in experiments with the same number of smaller files, although results are omitted here due to the space limitation.

Performance spectrum over degree of contention. We further investigate the performance spectrum over different contention levels to fully grasp the nature of the space-time tradeoff. For this evaluation, we modified the FXMARK benchmark in a way that 96 threads perform 64 bytes sequential reads per 4 KiB page on a single file whose size is varied from 1 MiB to 64 MiB, with `ext4` mounted. Figure 10 shows the performance spectrum of three concerned schemes. The most noticeable result is the sharp throughput decrease in RefCache as the file size grows, which clearly shows the negative effect of a large time overhead to scalability and so the necessity of the instant reclamation of hash entries. PAYGO effectively addresses the problem and undergoes no performance overhead for that issue. The gradual degradation of the throughput in PAYGO is due to the file data overflow in cache memory, resulting in the increased memory access overhead, which also occurs in other schemes.



(a) ext4 (b) btrfs (c) f2fs (d) xfs

Figure 9: The performance and the CPU breakdown of file systems (i.e., column labels (a)-(d)) with different reference counting schemes (i.e., row labels on the right side) under moderately contending workloads: the Linux page cache.

As we analyzed earlier, the vanilla scheme is ill-suited for the strongly contending condition (i.e., 1 MiB). But its performance rebounds quickly as soon as the degree of contention is alleviated, and it outperforms RefCache once it passes a break even point (i.e., 16 MiB file in our case). In-depth looking through profiling reveals that the acquire/release of an object lock in RefCache to handle hash collisions incur more overhead than the atomic operations in the vanilla scheme when hash collisions occur frequently due to a large number of objects accessed. After the break even point, the throughput of the vanilla scheme starts to decrease because the increased memory access overhead due to the file data overflow in cache memory becomes larger than the merit of dispersed contention.

Anchoring overhead. Since the anchor counter can be contended by only migrated threads, the frequency of thread migration determines the anchoring overhead. As described in §3.4, the design rationale for the OS scheduler usually inhibits frequent process migration. To confirm it, we ran the openfiles workload of the filebench on all cores that could cause thread migration between `REF` and `UNREF` operations, and counted the number of migration. For this experiment, we created 2,000 threads running the openfiles workload on 36 physical cores (disabling 60 cores), which hopefully causes frequent thread migration due to the load imbalance. However, during the 60 seconds experiment, less than 10,000 times of migration occurred.

Moreover, regardless of how the Linux scheduler is implemented, the more frequent the thread migration occurs, the less effective the CPU time is due to the long latency of the context switch. The latency of the context switch can be as short as 1 microsecond [24, 19] which is still relatively larger than the overhead of the atomic operation [13]. Recently, there has been an effort to reduce the latency of the context switch to several tens of nanoseconds by emulating a thread at the user level [24], but no such study has been

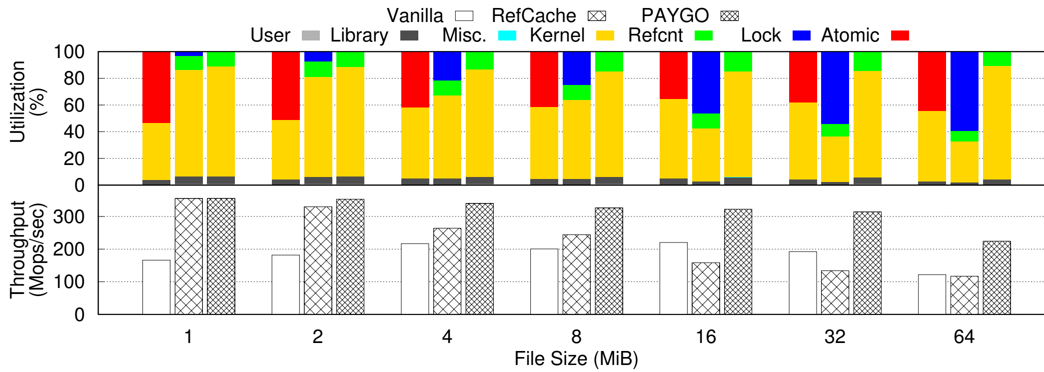


Figure 10: Performance spectrum of the vanilla, RefCache and PAYGO over varying contention levels on `ext4`.

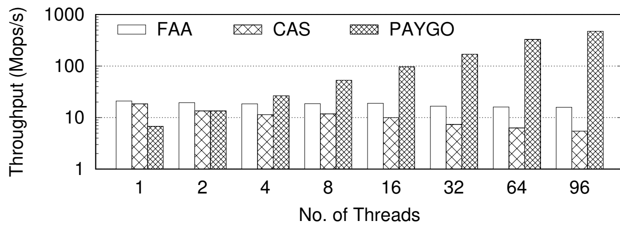


Figure 11: Throughput under the strongly contending workload (user-level reference counting).

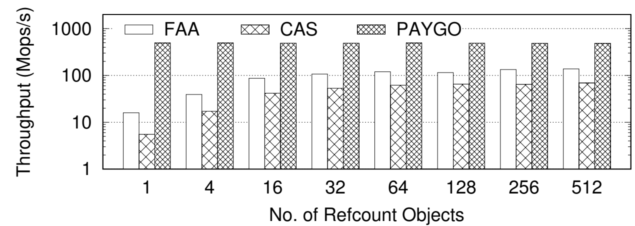


Figure 12: Performance spectrum of *FAA*, *CAS* and user-level PAYGO with a varying number of objects.

found in the kernel level. Therefore, there is practically no reduction in system throughput due to frequent changes of the anchor counter in PAYGO.

6.2.4 Scalability of User-level PAYGO

Next, we evaluate the performance of user-level PAYGO system calls to see its applicability. For the evaluation, we use a microbenchmark that has a varying number of threads (un)referencing user-level objects repeatedly. For comparison, we implement two methods based on our observation. The first one is to use atomic `fetch_add` and `fetch_sub` for reference counting. We call it *FAA*, and this is a typical implementation widely adopted in many systems. Note that this technique does not show performance collapses, but it cannot scale performance mainly due to hardware-based synchronization bottlenecks. The second one is to implement what is being used in the Linux page cache, which is based on the atomic compare-and-swap instruction. We call this *CAS*.

Figure 11 shows the throughput (i.e., the number of fetch_add/fetch_sub and `REF/UNREF` operations per second) of three schemes as we increase the number of threads, all of which access a single shared user-level object. As we manifested, the mode switch overhead of user-level PAYGO is quite noticeable and expected, considering the performance of *FAA* and *CAS* until 2 threads in our system. The performance number *FAA* and *CAS* achieve with 1 thread, however, is the peak number obtainable for reference counting a single shared object. After 2 threads, both *FAA* and *CAS* are either saturated or slowly degrading. Meanwhile, our user-level PAYGO scales the performance with no contention overhead. Figure 12 shows the performance spectrum of three methods as we increase the number of referenced objects with 96 threads. As shown in the figure, *FAA* and *CAS* suffer from synchronization bottlenecks initially when all of 96 threads access a small number of objects, but they slowly gain throughput up to a certain point as contention is dispersed. We believe that the saturation point observed here (i.e., 137 Mops/s) reaches the maximum capacity that our 96-core server can support. On the other hand, our user-level PAYGO could sustain the maximum throughput regardless of the count of objects.

Impact on application performance. As demonstrated above, user-level PAYGO may have a profound impact on application performance especially on multicore hardware. We have been conducting an in-depth code-level analysis of latent bottlenecks caused by reference counting in MongoDB, MariaDB, Boost.Asio, etc. What we have learnt from our preliminary study on such systems is that many applications using user-level reference counting mostly suffer performance bottlenecks that start occurring much earlier before the reference counting is responsible for severe performance degradation. For example, database systems we analyzed have recently undergone major changes to enhance its multicore scalability. As the systems community is battling pressing concerns, the contention around reference counting will soon appear as a primary bottleneck in achieving scalable performance.

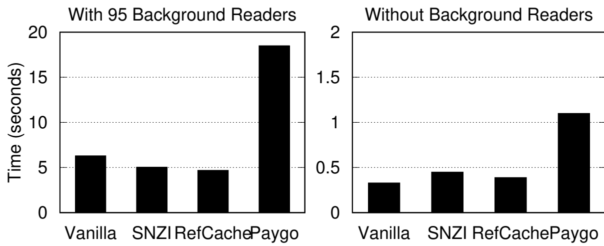


Figure 13: Query overhead comparison.

6.2.5 Comparing `preempt_disable()` and spin lock

As we briefly mentioned in §3.1, the use of `preempt_disable()`, instead of kernel spin locks or something similar, to prevent data races in `REF/UNREF` needs concrete justification. Hence, we compared the overheads of both methods by measuring the average clock cycles per each function pair (i.e., `preempt_disable()/preempt_enable()`, `spin_lock()/spin_unlock()`) by iterating them up to a billion times. The results show that the clock cycles for `preempt_disable()` and `spin_lock` converge to 14 cycles and 50 cycles, respectively. Throughout the experiments, the overhead of `preempt_disable()` remains a constant fraction (∼30%) of that of `spin_lock` regardless of the number of iterations performed. The main reason for the higher overhead of `spin_lock` is because it internally invokes `preempt_disable()` and executes additional code segments including an atomic instruction for cross-core communication supporting mutual exclusion on a global object. This is undoubtedly overkill for our case where we also use an atomic operation to safely decrease an anchor count from remote cores. In conclusion, `preempt_disable()` is a fast and safe method, as it has shown its usefulness in prior work, for preventing data races on local counters in our `REF/UNREF` implementations.

6.3 Query Overhead

In this section, we conduct the performance evaluation of the `READ-ALL` operation. To evaluate the query overhead of PAYGO, we measure the time to flush a 4 GiB file in the Linux page cache with and without background readers on `ext4`. The experiment first reads the entire file so that file blocks are all loaded in page caches. Then, it measures the time taken to drop the file from page caches. Since page caches are all clean (i.e., unmodified), dropping page caches is comprised of pure CPU activities. Figure 13 shows the completion time of different reference counting techniques. With background readers, RefCache surprisingly outpaced all other competitors, since RefCache may read a batch of global counters for multiple pages safely if their hash entries were flushed two epochs ago and no referencing occurred in between. Although PAYGO has less query overhead than RefCache for a single reference counter, the benefit of syncing the entire hash of dirty reference caches to global counters predominates the time overhead of two epochs with a large number of objects. Meanwhile PAYGO exhibits the overhead of reading a large number of local counters for each page and the vanilla scheme suffers contention due to atomic operations. Without background readers, the vanilla scheme is better than RefCache, but PAYGO still shows the same overhead of reading local counters. Nevertheless, the query overhead of PAYGO does not commensurate with the number of cores owing to its early detection of positive reference counter values (§3.4).

7 Limitations and Future Work

The limitations of PAYGO can be summarized as follows. First and foremost, PAYGO is not completely free from the counting-query tradeoff. We do not have a clue on whether it is possible or not. A proposal achieving low overhead in all directions must be a major breakthrough in systems research. Second, the way we handle the table overflow is rather naive, and one may find practical use cases that can stress PAYGO in that the overflow counter list is spotted as a bottleneck point. Our ongoing work is to apply user-level PAYGO to language runtime systems that surely benefit from user-level PAYGO.

8 Conclusion

Reference counting in modern file systems is not scalable on multicores, even under workloads with little or no logical contention. Through in-depth survey of present reference counting techniques designed for scaling file I/O operations, we found that there are space-time tradeoff and query-counting tradeoff in designing scalable reference counting techniques. In this paper, we have presented a novel reference counting scheme, PAYGO, that escapes the space-time tradeoff by using an anchor counter. PAYGO provides scalable counting and space efficiency with negligible time delay for the reclamation of hash entries. We have implemented PAYGO in the page cache in Linux. Our evaluation with different file system benchmarks demonstrated that PAYGO is practically useful in addressing severe contention arising in other reference counting techniques.

Acknowledgements. We would like to thank our shepherd, Vijay Chidambaram, and the anonymous reviewers for helping us improve this paper. This work was supported by the National Research Foundation of Korea grant (2017R1A2B4006134) and the Ministry of Science and ICT (MSIT), Korea (R0114-16-0046, Software Black Box for Highly Dependable Computing), and (2016-0-00023, National Program for Excellence in SW) supervised by the Institute for Information and communications Technology Promotion (IITP), Korea.

References

[1] ACAR, U. A., BEN-DAVID, N., AND RAINEY, M. Contention in structured concurrency: Provably efficient dynamic non-zero indicators for nested parallelism. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2017), ACM, pp. 75–88.

[2] ATTIYA, H., GUERRAOUI, R., HENDLER, D., KUZNETSOV, P., MICHAEL, M. M., AND VECHEV, M. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, ACM, pp. 487–498.

[3] BHAT, S. S., EQBAL, R., CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scaling a file system to many cores using an operation log. In *Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles* (2017), ACM.

[4] BOYD-WICKIZER, S. *Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels*. PhD thesis, Massachusetts Institute of Technology, 2014.

[5] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., ZELDOVICH, N., ET AL. An analysis of linux scalability to many cores. In *OSDI* (2010), vol. 10, pp. 86–93.

[6] CLEMENTS, A., ZELDOVICH, N., ET AL. sv6: Posix-like scalable multicore research os kernel. `https://github.com/aclements/sv6`, 2014.

[7] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 211–224.

[8] COLLINS, G. E. A method for overlapping and erasure of lists. *Commun. ACM 3*, 12 (Dec. 1960), 655–657.

[9] CORBET, J. The search for fast, scalable counters. `https://lwn.net/Articles/170003/`, 2006.

[10] CORBET, J. Per-cpu reference counts. `https://lwn.net/Articles/557478/`, 2013.

[11] CORBET, J. Filebench. `https://github.com/filebench/filebench/wiki`, 2017.

[12] DASHTI, M., FEDOROVA, A., FUNSTON, J., GAUD, F., LACHAIZE, R., LEPERS, B., QUEMA, V., AND ROTH, M. Traffic management: a holistic approach to memory placement on numa systems. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 381–394.

[13] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 33–48.

[14] ELLEN, F., LEV, Y., LUCHANGCO, V., AND MOIR, M. Snzi: Scalable nonzero indicators. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (2007), ACM, pp. 13–22.

[15] KNUTH, D. E. Structured programming with go to statements. *ACM Comput. Surv. 6*, 4 (Dec. 1974), 261–301.

[16] MCKENNEY, P. E., BOYD-WICKIZER, S., AND WALPOLE, J. Rcu usage in the linux kernel: One decade later. *Technical report* (2013).

[17] MIN, C., KASHYAP, S., MAASS, S., AND KIM, T. Understanding manycore scalability of file systems. In *USENIX Annual Technical Conference* (2016), pp. 71–85.

[18] NARULA, N., CUTLER, C., KOHLER, E., AND MORRIS, R. Phase reconciliation for contended in-memory transactions. In *OSDI* (2014), vol. 14, pp. 511–524.

[19] PETER, S., LI, J., ZHANG, I., PORTS, D. R., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS) 33*, 4 (2016), 11.

[20] PIGGIN, N. A lockless page cache in linux. In *Proceedings of the Linux Symposium* (2006), vol. 2.

[21] Apache HBase. `https://github.com/apache/hbase/blob/rel/2.1.0/hbase-server/src/main/java/org/apache/hadoop/hbase/io/hfile/bucket/BucketCache.java#L484`.

[22] Facebook RocksDB. `https://github.com/facebook/rocksdb/blob/v5.14.3/utilities/persistent_cache/hash_table_evictable.h#L62`.

[23] MariaDB Server. `https://github.com/MariaDB/server/blob/10.3/storage/innobase/buf/buf0buf.cc#L4350`.

[24] SEO, S., AMER, A., BALAJI, P., BORDAGE, C., BOSILCA, G., BROOKS, A., CASTELLO, A., GENET, D., HERAULT, T., JINDAL, P., ET AL. Argobots: A lightweight threading/tasking framework. Tech. rep., Argonne National Laboratory (ANL), 2016.

[25] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A flexible framework for file system benchmarking. *USENIX; login 41* (2016).

Notes

[1] PAYGO: *pay* migration tax (i.e., the anchoring overhead) *as you go* to other core.

Speculative Encryption on GPU Applied to Cryptographic File Systems

Vandeir Eduardo[1,2], Luis C. Erpen de Bona[1], and Wagner M. Nunan Zola[1]

[1]Federal University of Paraná
[2]University of Blumenau

Abstract

Due to the processing of cryptographic functions, Cryptographic File Systems (CFSs) may require significant processing capacity. Parallel processing techniques on CPUs or GPUs can be used to meet this demand. The CTR mode has two particularly useful features: the ability to be fully parallelizable and to perform the initial step of the encryption process ahead of time, generating encryption masks. This work presents an innovative approach in which the CTR mode is applied in the context of CFSs seeking to exploit these characteristics, including the anticipated production of the cipher masks (speculative encryption) in GPUs. Techniques that demonstrate how to deal with the issue of the generation, storage and management of nonces are presented, an essential component to the operation of the CTR mode in the context of CFSs. Related to GPU processing, our methods work to perform the handling of the encryption contexts and control the production of the masks, aiming to produce them with the adequate anticipation and overcome the extra latency due to encryption tasks. The techniques were applied in the implementation of EncFS++, a user space CFS. Performance analyzes showed that it was possible to achieve significant gains in throughput and CPU efficiency in several scenarios. They also demonstrated that GPU processing can be efficiently applied to CFS encryption workload even when working by encrypting small amounts of data (4 KiB), and in scenarios where higher speed/lower latency storage devices are used, such as SSDs or memory.

1 Introduction

In the era of storing data in cloud services, where they end up being written to server disks scattered around the world, it is increasingly important to deal with data confidentiality before it is actually stored. Cloud storage services typically provide secure communication channels in the data transfer process. However, they do not bother encrypting the files before writing and transferring them, which usually results in data being stored in clear text format.

One way to get around the problem of storing files in clear text is to use a Cryptographic File System (CFS). By using cryptographic techniques, they act transparently, encrypting the data before they are actually stored. CFSs can apply encryption functions at different levels of the data storage and retrieval process, and can encrypt individual files, directories, partitions, and entire disks. CFSs typically encrypt the contents of files in blocks, in order to allow accesses at random without having to decipher them completely. They use block ciphers and modes of operation that dictate specific rules on how the encryption process should be performed.

CFSs have intensive processing demands due to the volume of data and the cost of cryptographic functions. Parallel processing techniques on CPUs or GPUs can be used to meet these demands. As a way to achieve better performance on these systems, you can use the parallel processing capabilities offered by multiprocessor computers and servers, whether in the form of multiple CPUs or GPUs.

The acceleration of symmetric GPU encryption algorithms is a well-studied subject, including the Advanced Encryption Standard (AES) [1] [19] [18] [13] [6] [12], in addition to its application in the context of CFSs [26] [14] [10] [25]. One of the studies related to the acceleration of AES in GPUs resulted in the AES (WAES) *Warped*, presenting significant performance gains [28]. This study also resulted in the creation of the WAESlib library, which can be used to facilitate the integration of applications with the use of AES cryptographic processing in GPUs.

A major feature of WAES, apart from GPU processing, is the use of the operating mode called *Counter* (CTR). The CTR mode has two particularly useful features: the ability to be fully parallelizable in both encryption and decryption operations and the ability to perform the initial step of the encryption process in advance, generating what we call the *encryption masks*.

In addition to exploring the first characteristic, WAES also explores the second, making it capable of computing encrypted data in advance. So when an application actually needs encrypted data, it will already be available, ready to

be used. This feature may prove useful in a number of situations, allowing effective GPU encryption of small amounts of data, something indicated to be impracticable in previous research [26] [14] [10] [25]. Since most file system have blocking factor of 4 KiB or lower, using WAES can bring interesting results.

However, endowing a CFS with the ability to process its cryptographic functions by a parallel processor is not restricted to the simple use of a library. For this to be done efficiently, there are significant challenges to overcome.

A first concern would be about how to implement the CTR mode in a CFS respecting the security requirements required by the mode. It is necessary to treat issues related to an essential element to the operation of the CTR mode called *nonce*. These issues pertain to their generation, storage and management. The resulting implementation needs to ensure that the use of the CTR mode does not cause negative impacts to the CFS performance, which could nullify the gains from parallel processing the cryptographic functions.

Another point with direct connection to the parallel processing of the encryption workload regards to how to manage the CFSs encryption contexts, controlling the production of encryption masks on the multicores or GPU for subsequent use in data encryption and decryption tasks. The read and write operations, either sequential or random, have different characteristics, which requires the creation of different techniques for managing these contexts in order to produce the encryption masks with adequate time in advance. A related issue is about how to aggregate enough work for the cores, for parallel processing from typically sequential workloads generated by each CFS client application, without compromising latency on the CFS operations.

This paper presents techniques that try to overcome these challenges and constitute the main contribution of this work. The authors are unaware of previous work that has used the CTR mode in the context of CFSs, mainly seeking to exploit the advantage of being able to produce the encryption masks in a speculative way. The main objective of such an approach is to obtain a better performance of the CFS in order to achieve higher throughput and more efficient use of CPUs.We specifically deal with aspects related to the counter mode encryption. Authentication issues could be dealt with at FS level or also in conjunction to the encryption processes. In the latter case, our techniques could also be applied in conjunction to other authenticated encryption methods (e.g. GCM [3] or OCB [22]) that are parallelizable and work with similar counter mode constraints.

The techniques presented were validated through the creation of the EncFS++ CFS, based on the preexisting CFS called EncFS [7]. The results show that it was possible to obtain significant performance improvements (in throughput and CPU efficiency) in several scenarios, even in very low latency environments where the base file system (FS) was stored in memory. In SSD disk micro-benchmarks this improvement reached a maximum of 269% in throughput and 112% in CPU efficiency for sequential writing of large blocks. Indeed, in the most adverse scenario for SSDs, the random reading, it was possible to achieve gains of 18% in throughput and 18% in CPU efficiency. The results also demonstrate that the applied techniques allowed performance improvements even when processing small amounts of data (requests of 4 KiB). Our approach was also evaluated in a scenario of low latency "devices", by locating the tested file system in memory (RAM). We also present synthetic macro-benchmarks performed in solid-state disks.

The rest of this paper is organized as follows: In Section 2 we introduce background aspects related to CTR encryption mode, cryptographic storage systems and the GPU encryption library used in our work. We discuss related work in Section 3. Issues related to CTR implementation and storage on CFSs are shown in Section 4. The management of encryption contexts is presented in Section 5. We show micro and macro-benchmark evaluations of our proposed scheme in user space CFS EncFS++ in section 6. Finally we present our conclusions in Section 7.

2 Background

This section presents some concepts relevant to the understanding of this work. The operation of the CBC and CTR modes is discussed, emphasizing the advantages of the CTR mode. Examples of cryptographic storage systems and the different levels at which they can act on the Linux IO subsystem are also presented. EncFS is also previously presented considering that the techniques of this work are implemented on this. Finally, we present WAES, and its WAESlib library, which exploit the advantages of CTR mode and allow the execution of cryptographic functions in GPUs.

2.1 CBC and CTR Operation Modes

CFSs encrypt files in fixed-size blocks so that you can access parts of them without having to encrypt or decrypt them completely. The blocks are coded using block ciphers that subdivide these blocks into smaller blocks (usually 64 or 128 bits), processing them individually [16] [20] [23]. The processing of these smaller blocks during encryption must follow specific rules which are known as operation modes. Among these modes are the CBC and CTR [2].

In CBC mode, each cipher block depends on the clear text block (P), the key (K), and the previous ciphertext block (C). In the encryption process, before a block is encrypted, it undergoes an operation of `XOR` with the previously encrypted block. The encryption operation can be seen in Figure 1a. The CTR mode employs a counter that is incremented for each new block processed. This counter, called *nonce*, is encrypted and then used in a `XOR` operation with the clear text to produce the ciphertext, as in Figure 1b.

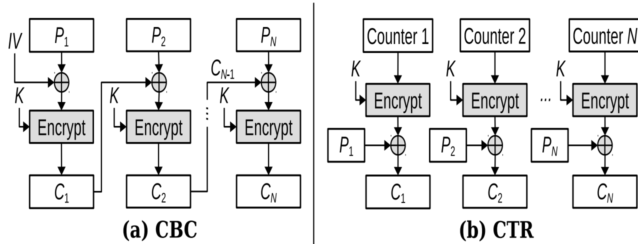


Figure 1: CBC and CTR encryption process.

The CBC mode requires the use of an initialization (IV) vector. Data in IV should not be predictable, ie an attacker with use of some clear text should not be able to predict the IV that will be used in the future process of encryption. The choice of data to be used to populate the IV should follow specific rules. Generally, a pseudorandomly generated number is used, which is encrypted with the same key used in the remainder of the encryption process [20] [23] [4] [2].

In CTR mode, the use of *nonce* enforces a unique nonce and key pair. That is, the same *nonce* can not be used in different encryption operations with a given key in the encryption process. This leads to the explicit need to control nonce increment between different encryption processes that use the same key [2].

With regard to the parallel execution capability of the blocks, in CBC this is not possible. This is due to the fact that the encryption of a particular block depends on the encryption of the previous block. Only the decryption process can be parallelized. In contrast, the CTR mode is highly parallelizable since there is no dependence between the blocks. In addition, since encryption can be done only on *nonce* it is possible to generate encryption masks in advance. Thus, these masks will be ready to be used when the data is known.

Besides the efficiency characteristics of CTR, it has proven security bounds. In fact, the concrete security bounds one gets for CTR-mode encryption, using a block cipher, are no worse than what one gets for CBC encryption [15].

2.2 Cryptographic Storage Systems

Software-based storage systems can operate at different levels of the Linux I/O subsystem. For this reason, the integration of cryptographic resources into this system can be done in several ways, giving rise to different types of cryptographic storage systems.

There are file systems that act on user space, communicating with kernel space through libfuse and the FUSE module [27]. Because they run in user space, they offer greater flexibility, not requiring elevated user privileges to be configured, used, and even developed and tested. On the other hand, by constantly calling from the user to kernel space, they cause a lot of context changes, which compromises their performance. An example of a CFS of this category is EncFS [7].

Other systems are implemented as kernel modules, communicating directly with the Virtual File System (VFS). By running entirely in kernel space, they significantly reduce the need for context switches, improving their performance. An example of this category is eCryptfs [8].

Another approach is to act on the block layer through the device mapper. In this way the encryption will occur only in an agnostic form acting on blocks to the file system itself. They present the best performance, at the cost of not being as flexible as the systems described above. Dm-crypt exemplifies such a system.

The positioning in the Linux IO subsystem of each of these three system examples can be seen in the Figure 2.

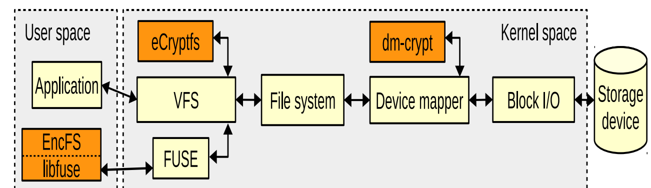


Figure 2: Cryptographic storage systems placement on Linux I/O subsystem.

2.3 EncFS

The implementations of this work were realized on a cryptographic FS called EncFS [7]. The main reason for choosing EncFS was that it is well known and run in user space, which has two important advantages: it facilitates development / testing and allows direct access to the CUDA [17] library, which is only available in user space.

In its encryption processes EncFS can use AES, Blowfish and Camellia symmetric block ciphers available in the OpenSSL library. In order to be able to randomly access data within an encrypted file, its contents are encrypted in blocks with fixed sizes. The size is set during FS creation and can range from 1 to 4 KiB. In full block encryption, it uses CBC mode of operation.

Three different types of IVs are used in block encryption. The first is the IV contained in the volume key (IVV). The second is the IV of the file (IVA) and the third is the IVs for encryption of the blocks within the files (IVB). The IVV is stored in the final bytes of the volume key (VK), which is generated randomly in the FS creation. IVA is also generated randomly in creating a new file. $NumBlock$ corresponds to the block number within the file. IVB is obtained by applying the following cryptographic hash function:

$$IVB = HMAC_CTX(VK, IVV \parallel (NumBlock \bigoplus IVA))$$

Because EncFS uses CBC mode, the IVs used in block encryption (IVBs) must meet the unpredictability requirement. This is why the concatenation of different IVs and the

application of the cryptographic hash function is done. Considering the same file stored in a given EncFS system, VK, IVV and IVA remain constant, the only variable being the block number. This feature has advantages over performance, as it allows the IVBs to be dynamically calculated, and it is not necessary to store them. In addition, the same IVB can be reused in rewriting processes of the same block within a same file.

2.4 WAES and WAESlib

WAES (*Warped AES*) [28] is the implementation of a high performance algorithm for heterogeneous parallel processing employed for data encryption using GPU. WAES was implemented with 128-bit and 256-bit keys in CTR mode. The heterogeneous part of WAES refers to the possibility of performing all the steps of the CTR operation mode on the GPU or performing the final part of `XOR` with the text to be encrypted or decrypted in the CPU.

WAES explores the characteristic of the CTR mode of operation to implement a technique called speculative encryption. From the moment you have the encryption key and *nonce*, WAES can compute and pre-populate *buffers* with encrypted data. In this way, these data are available in advance and are called encryption masks.

Anticipating the production of encryption masks in conjunction with performing the final `XOR` step of CTR mode on the CPU are critical to efficient processing when working with small amounts of data in GPUs. This anticipation allows you to compensate for the latency involved in GPU operations such as data transfer and GPU kernel activation. Performing the final `XOR` step on the CPU avoids the need to transfer the data to be GPU encrypted.

WAES also allows buffer aggregation, which are processed in the same WAES kernel activation. The buffers aggregation technique decreases the latency and increases the utilization of the GPU processing cores, consequently resulting in higher bandwidth, thus being a promising technique to be used with file systems submitted to workloads composed of requests in the 4 KiB range.

The techniques proposed by WAES were implemented and made available in a library called WAESlib. Calling this library will prepare the CUDA environment and launch the WAES *kernel* which will pre-compute the encryption masks. Calls to masks made through WAESlib are asynchronous, freeing the application to perform other tasks while the encryption masks are computed on the GPU.

WAESlib also offers a priority feature that can be used to control the production order of encryption masks. Contexts defined with higher priority have their masks produced before. This feature can be explored by trying to sort the production of the masks according to the order in which blocks of a file are accessed.

3 Related Work

The GPU acceleration of symmetric block ciphers is a well-known subject in the literature [1] [19] [18] [13] [6] [12]. Some of these researches aim to take advantage of the acceleration of these ciphers in cryptographic storage systems. In this section we first present research that worked with systems running in user space and in the sequence those that worked with systems running in kernel space.

In user space. The Engine-CUDA [21] project features an OpenSSL engine capable of performing symmetric GPU encryption operations. The project is used in the development of the work done in [6], where the engine has been improved and extended, being implemented other ciphers such as DES, Blowfish, IDEA, Camellia and CAST5. Results show that GPU processing becomes effective above 16 KiB and can be eight times better when approaching 8 MiB.

In [5], CrystalGPU is presented, which is a framework that aims to facilitate the integration of GPU processing into applications. This framework is used in the implementation of a FUSE-based CFS called CRSFS [26] [14]. In this work the encryption algorithms used are the AES operating in the ECB and CBC modes. Experimental results show that it is advantageous to use CPU processing for data sizes up to 4 KiB and GPUs for data above 16 KiB.

In kernel space. The OpenBSD Cryptographic Framework (OCF) [11] is a framework developed with the goal of providing a service virtualization layer within the kernel by offering an API that hides the specific details of each accelerator. The work presented in [9] uses the Linux version of the OCF to integrate the GPU processing resources into this *framework*, allowing to execute in NVIDIA GPUs symmetric encryption operations using AES in ECB mode.

Gdev [10] is a runtime system designed to run in the kernel space to manage the use of the GPU in a way that is similar to the processes running on the CPU. Its main feature is to control the GPU without relying on proprietary drivers and access libraries that run in user space. One of the practical applications demonstrated is the eCryptfs adaptation to perform encryption on the GPU. For writing operations the gain was up to 2x.

The GPUStore [24] [25] is a runtime and framework system designed to facilitate the integration and efficient use of GPU processing for data storage systems that run in *kernel* space. In [25] as a practical application, the GPUStore is used to accelerate code inside the kernel from the dm-crypt, and eCryptfs. For data encryption above 256 KiB, the performance was 36 times better.

Two points can be highlighted in relation to the state of the art presented. First, when applying the processing acceleration of the cryptographic functions in GPU to file systems none of the work took advantage of the CTR mode. To our knowledge, the work proposed in this paper is the first to explore the CTR mode in this context. The second is that results in the reviewed

work indicate that GPU processing only begins to be efficient when working with larger size requests (> 16 KiB).

The use of the CTR mode for CFS proposed here allows new forms of performance gain not exploited in previous work: exploring the ability to produce the encryption masks speculatively in the GPU; and to avoid transferring the data to be encrypted or decrypted from the CPU to GPU. These techniques allow to compensate for the additional latency inherent to the GPU processing that are especially impacting when the size of the requests is small, according to the research data presented. Moreover, we show that the data space needed for nonce storage is very small and that it can be retrieved and operated without compromising performance. Our approach is applicable, with some adaptation, to other *counter mode* encryption methods that include authentication and are parallelizable, including on multicores.

4 CTR Implementation on CFSs

Before we can exploit the benefits of using CTR in the context of CFSs, it must be adapted to operate in CTR mode. One essential element for the operation of the CTR mode is the nonce. We must carefully think the ways to generate and store these nonces as these factors can impact the performance of the CFS and possibly cancel any impending gains from the application of speculative encryption using GPU processing. This section presents some techniques that address such issues.

4.1 Nonces Generation

Considering the issue of *nonces* generation and seeking to satisfy the demand for *nonce* uniqueness [3] in the CTR mode, this work proposes a deterministic form for its generation. The technique relies on a single global counter that is incremented on each write or rewrite of a data block in the FS. The numbers generated by this counter are used as *nonce* in the encryption and decryption functions. Considering, for example, the use of a 128-bit block cipher, the *nonce* must also have that size. Therefore, a 128-bit counter can be used.

In implementations such as the OpenSSL library, the least significant nonce bits are used as an internal counter incremented every 16 bytes of encrypted data. The amount of bits required for this counter is given by the formula $log_2(x/16)$, where x is the number of bytes to be encrypted with this *nonce*. Since *nonces* must be unique, the amount of blocks that can be written or rewritten is given by 2^{128-x}, where x is the amount of reserved bits.

Figure 3 illustrates the format of the global counter and the least significant bits reserved for the internal counter of the CTR mode.

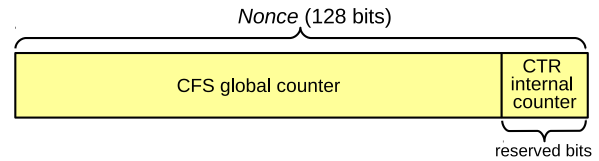


Figure 3: Nonce obtained from CFS global counter.

4.2 Nonces Storage

Regarding the storage of *nonces*, when a FS block is written, the *nonce* used in its encryption process is obtained from the FS global counter. In a future process of reading the same block, in order to be able to decipher its contents, it is necessary that the same *nonce* be passed as a parameter to the decryption functions. Therefore, it must be stored for future retrieval since any rewritten block needs a different *nonce* and file blocks may have been updated in any given order.

A naive method would be to store the *nonces* individually, prepending each block. However, this approach is inappropriate for random reading because it makes it difficult to read *nonces* in advance in order to trigger the anticipated generation of encryption masks. In addition, FS block sizes are already page aligned and doesn't have the space to include the *nonces*.

One way to work around this problem is to store *nonces* separately from the encrypted file. Thus, they occupy contiguous regions on disk, streamlining the processes of reading and writing. Also, since each *nonce* has only 16 bytes, accessing it individually is not efficient. In addition to storing them in separate files, it is also interesting that they be read and written in a clustered fashion.

However, grouped and separate storage is not the ideal solution for all cases. In a scenario that works with very small files, containing few blocks of data, reading and writing an entire group of *nonces* can lead to wasted memory and disk space. Ideally, we would store the initial *nonces* of all files in a single storage location. Only when a file occupies a greater number of blocks, its other *nonces* begin to be stored in a separate and exclusive file.

To deal with the storage situation of the initial file *nonces*, this paper proposes a solution based on the Unix *inodes* storage system. This structure was called *nonce node* (*nnode*). There is a single file responsible for storing the *nnodes* of all files stored in the FS. The general structure of this file can be seen in Figure 4a.

At the beginning of the *nnodes* structure (file) the FS global counter is stored. Following is a counter that controls the amount of *nnodes* used, as well as a bitmap that is intended to control the allocation of *nnodes*. The *nnodes* themselves are in the sequence. Each file stored in the FS has an *nnode* allocated to it and stored in that structure. In the expansion of an *nnode*, shown in Figure 4b, you can see the information contained in it. They are the inode number of the file (used to

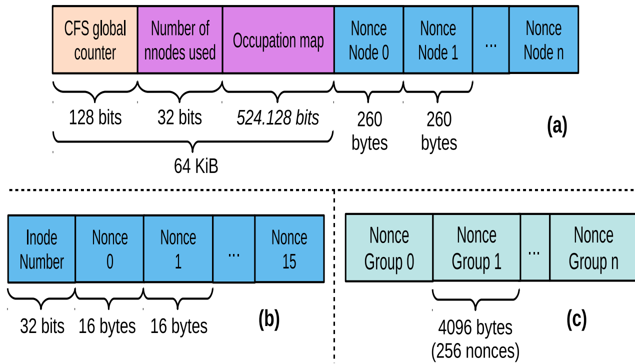


Figure 4: Nonce nodes (nnodes) file format (a). Detail of a nnode (b). Exclusive nonce file format (c).

index the file to *nnode*), followed by the first 16 *nonces* of the file.

Considering that an *nnode* is used to store the first 16 *nonces* of a file, assuming for example that the FS uses 4 KiB blocking factor and supposing files of up 64 KiB, only the structure of one *nnode* is sufficient to store its *nonces*. This form of storage helps to optimize access to small files as it allows nnodes to be stored in near regions on the disk. In addition, it also helps to avoid wasting disk space by allocating larger structures for storing an entire group of *nonces*.

If a file grows beyond the 16 blocks, the other *nonces* begin to be stored in a separate and unique file for each file stored in the FS. The format of this file can be seen in Figure 4c. *Nonces* are stored in groups of 256 *nonces* to match the size of a 4 KiB memory page. This implies that that one *nonce group* is stored for each $256*4$ KiB segment of a file (i.e. a 1 MiB file segment). This way the overhead, in terms of disk space to keep the nonces, stays negligibly under 0.4% (i.e. at most $4KiB/1MiB$).

5 Encryption Contexts Management

In order to understand some of the techniques proposed in this work, which aim to use WAESlib efficiently in the context of CFSs, it is necessary to briefly describe its operation. One of the main components of the library is the so called encryption contexts: a kind of logical organization used by the library to control the production and application of encryption masks.

Basically, the use of the library is a process that involves calling a context definition function to start the production of the encryption masks and then call the functions that apply these generated masks.

In the use of the context definition function, the identification number of the context being defined, the identification number of the previously defined key, *nonce* and a priority number are given. The call to this function is asynchronous, releasing the application to perform other tasks. After the function call, the library can fire the WAES *kernel* so that it begins computing the encryption masks in advance in the GPU.

Internally the priority is used by WAESlib to aggregate and define the order in the production of the masks. As the masks are generated, they are transferred to the system memory and are available for use in the data encryption and decryption functions.

The other two main functions are used to instruct WAESlib to encrypt and decrypt data. In your call information such as the context identification number, the *buffer* containing the data to be encrypted or decrypted and its size is passed. These are synchronous functions that cause WAESlib to use the previously calculated encryption mask for the context in question, available in system memory, to effectively encrypt or decrypt the data. This final step consists of a bitwise `XOR` operation between the data and the mask generated, being performed in the CPU.

Despite its simplified use, there are significant challenges involved in this process. They mainly concern how to effectively group, define, and use these encryption contexts in writing and reading operations. One of the main contributions of this paper is to present some techniques that show how to do this. They demonstrate how to use encryption contexts to ensure that encryption masks are produced promptly in advance and are readily available at times when they are required to effectively encrypt and decrypt blocks of data. These techniques seek to explore how files stored in the CFS are accessed, for example by examining data locality issues and access patterns.

5.1 Encryption Context Pools

The approach carried out in this work in grouping the encryption contexts was based on the idea of pools that are used differently according to the operation being performed. We identified the need to create at least two group types: a unique *pool* of encryption contexts, at CFS level, used in sequential and random write operations; and several *pools* of contexts for decryption, one per open file, used in sequential and random read operations. The working idea of these pools is described in the following two subsections.

5.1.1 Contexts Pool for Encryption/Writing

In order to hide the latency involved in the process of generating the masks in the GPU, as well as its transference from the memory of the device to the system memory, it is essential that the triggering of the masks generation occurs as soon as possible. This work proposes the use of a single *pool* of contexts used for the generation of masks that can be used in the process of block encryption. Consequently, they can be used in the processes of writing and rewriting blocks of all CFS files.

Using as an example a *pool* containing eight contexts, when the CFS is mounted, eight contexts will be allocated for this *pool*. The current value of the CFS counter is copied and used to define the contexts contained in the *pool* using a *nonce* corresponding to the value of that counter, which is incremented with each defined new context. Respecting the need to keep the least significant bits of the counter reserved and considering a CFS that uses blocks of 4 KiB, the counter increases in the value of 256 (as each 4 KiB data block totals 256 AES blocks).

The queue start (pool beginning) indicator is used to store the context ID containing the oldest mask generated. At the end of this process, shown in Figure 5a, the CFS will have an amount corresponding to the size of the pool in masks ready to be used in block encryption processes.

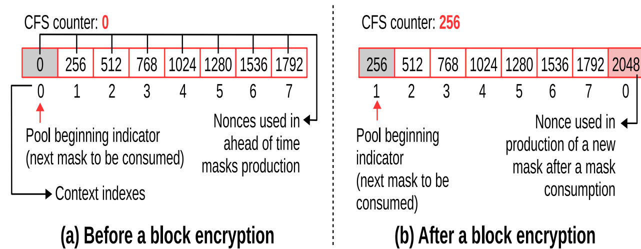


Figure 5: Contexts pool usage example for encrypting blocks in write operations.

However, in order to avoid wasting *nonces* and avoid the need to temporarily store the *nonces* used to generate masks, the CFS counter is only effectively incremented when a mask is used in the encryption process. The *nonce* to be saved to be used in the future process of decrypting the block is obtained from the value of the CFS counter when mask usage occurs. Soon after, the counter is incremented, corresponding to the value used before in the generation of the mask of the subsequent context. The queue start indicator is then moved.

After consuming the mask, the context that contained the newly used mask is reused to trigger the generation of a new mask. Thus, all contexts are always maintained with a new mask. The priority used in this context reset (redefinition) may be as small as possible, since the mask to be produced will only be consumed after all others. The value of *nonce* to be used in the generation of the next mask can be obtained through the formula $x = y + (1 << w) * z$, where x corresponds to the next *nonce* y the current value of the CFS counter, w the amount of reserved bits of CTR mode and z the size of the *pool* of writing contexts. Figure 5b illustrates the state of *pool* after consumption of a mask, followed by generation of a new mask.

5.1.2 Contexts Pool for Decryption/Reading

The *context pool* used for decryption, and consequently in the reading processes, works as a mask window that is shifted according to the region of the file being read. In this technique, each open file has its unique *context pool* whose trigger for masking occurs according to the block number being accessed. The window is positioned on the first read block, generating the masks for the subsequent blocks according to the size of the *pool*. As the blocks are read and the masks consumed in the decryption processes, the window is shifted and new masks are generated.

The Figure 6 illustrates the behavior of the *pool* of contexts containing the generated masks in a sequential read process, initiating at the beginning of the file. Also taking as an example a *pool* containing eight contexts, when the file is opened, the *nonces* of the first eight blocks are used to define the encryption contexts that begin to produce the masks.

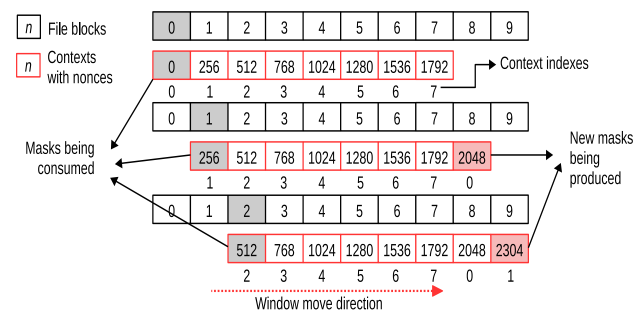


Figure 6: Contexts pool usage example for decrypting blocks in sequential read operations.

The priority feature offered by WAESlib can be exploited to dictate the production order of these masks. Thus, when defining each of the contexts of the window, one can define the context contained at the beginning of the window with the highest possible priority, gradually decreasing this priority in the definition of subsequent contexts. The masks needed to decipher the first blocks are getting ready before the masks of subsequent blocks.

As illustrated in Figure 6, after consumption of the mask referring to the $0th$ block, one can trigger the production of the mask for block eight with the lowest possible priority, since the trend is that it will be used only after consumption of the masks from the previous blocks. In a purely sequential reading, this process repeats itself, with the consumption of the mask at the beginning of the window and the subsequent production of a new mask at its end.

The sliding window technique can also be applied for random access. In this case, its beginning is always moved to the position corresponding to the first block being read. When this displacement occurs, two situations may occur at first: (i) the window is fully shifted to an earlier position, i.e. $(x - y) > z$, where x is the previous initial position of the window, y the new starting position and z the size of the window; (ii) the window is shifted fully forward, where $(y - x) > z$. Both situations are illustrated in Figure 7a and 7b, respectively. Because the new starting positions are completely outside the previous

window, it is necessary to reset (redefine) all contexts for new masks to be generated.

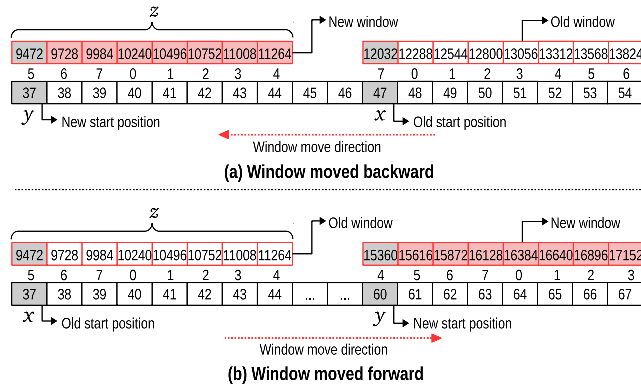


Figure 7: Window moved to a far position with the redefinition of all contexts inside the pool.

The other two situations refer to cases where, after moving the window, part of the new window ends up overlapping the previous one. These situations occur when the window is moved to a near previous position, that is $(x-y) <= z$; and when it is shifted to a near posterior position, that is, $(y-x) <= z$. The figure 8a and 8b illustrates the first and second case, respectively. In these situations, the window that occupied the previous position contains some masks that can be reused, it is not necessary to redefine the contexts of the positions that overlap, making the process more efficient.

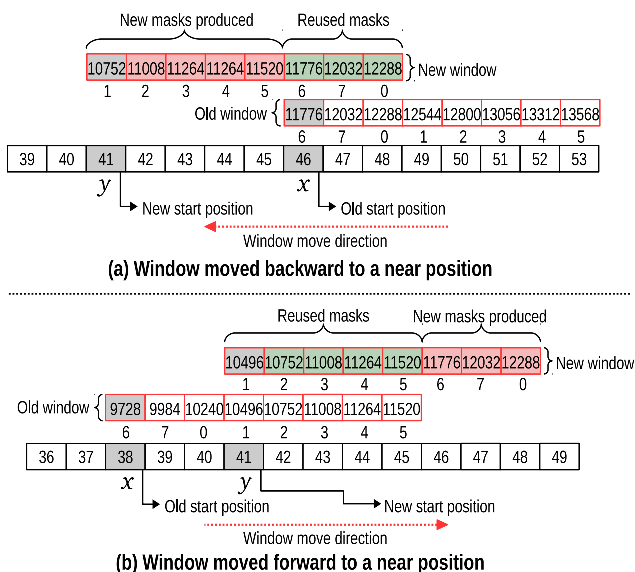


Figure 8: Window moved to a near position with reuse of previously produced masks.

6 Evaluation

As a way of validating the techniques discussed in the previous sections, they were implemented on the EncFS CFS. We have called this new version EncFS++. The performance analyzes in this section compare EncFS and EncFS++. The EncFS version is the original, using CBC mode and with exclusive CPU processing. The EncFS++ version uses the CTR mode implemented according to the ideas discussed in Section 4, as well as the encryption context management techniques for GPU processing discussed in Section 5.

Performance measurements were performed at the micro-benchmark level measuring throughput with 4, 64 and 128 KiB requests in read and write operations both sequentially and randomly. To measure throughput the tool `fio` was used; to measure CPU utilization, the `pidstat` tool was used to exclusively monitor the EncFS and EncFS++ processes and subprocesses. Macro-benchmarks were also performed with the `filebench` tool, as described at the end of this section. The tests were performed on a computer running Linux OS with 4.10 kernel, using Intel Core i7-7700HQ @ 2.8GHz, 32 GiB memory and Western Digital SSD disk model WDS240G1G0A. The libfuse version used was 2.9.4, OpenSSL 1.0.2g and WAESlib 2.01g. The GPU used was an NVIDIA GeForce GTX 1070 (mobile version), in CUDA 9.2 environment. The micro-benchmark measurements were performed on a 16 GiB file, being repeated 10 times and taking the simple arithmetic mean of the results. Among the measurements, the base FS (`ext4`, default settings) was unmounted, re-created and remounted. Among all the repetitions, the caching pages were discarded.

As a way of comparing CPU utilization between versions, an index called CPU utilization efficiency was used. It was calculated using the following formula: $x = (y/z)/(w/k)$, where x corresponds to the efficiency index being calculated, y and z correspond to the throughput and utilization value of CPUs reached in the version being compared; w and k correspond to the value of bandwidth and CPU utilization reached in the version with which the comparison is being made. Thus, it is possible to obtain a comparison between the amount of work performed by each version contrasted with the percentage of CPU usage.

Figure 9 shows the bandwidth values obtained in the two versions of EncFS, in a sequential read scenario with base FS stored on disk (SSD) and in memory. In the secondary y axis (right side) the percentage of the bandwidth variation of EncFS++ is displayed in relation to its original version. Table 1 displays the values obtained. The next to last column shows the percentage increase or decrease in bandwidth. Positive and green values indicate that the EncFS++ version had an increase in bandwidth compared to the original version. Negative values in red means the opposite. The last column shows the calculated efficiency values. Values above 1 were colored green indicating more efficient CPU usage in the

EncFS++ version. Values below 1 were colored red indicating less efficient use.

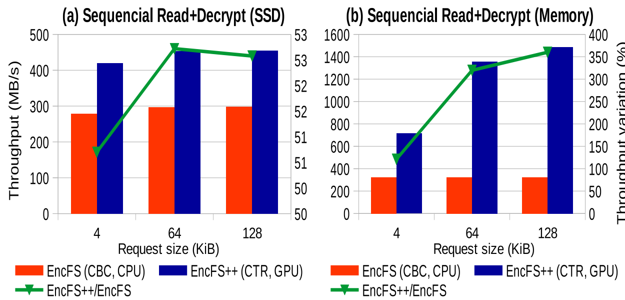


Figure 9: Throughput variation (sequential read+decrypt).

Req. Size (KiB)	Sequencial Read+Decrypt (SSD)						Sequencial Read+Decrypt (Memory)					
	EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/EncFS		EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/EncFS	
	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency
4	278.47	11.32	419.66	14.71	50.70	1.16	323.25	12.20	714.82	21.76	121.14	1.24
64	297.04	12.10	453.64	11.54	52.72	1.60	322.91	12.20	1,355.20	23.90	319.68	2.14
128	297.69	12.11	454.20	11.54	52.58	1.60	323.03	12.20	1,485.59	23.90	359.90	2.35

Table 1: Throughput and CPU utilization: sequential read+decrypt.

With the base FS stored on disk, it is perceived that the maximum throughput gain in EncFS++ is approximately 52%, with an increase in CPU utilization efficiency around 60%. However, the bandwidth number reached approaches the throughput limit supported by the disk (500 MB/s). With base FS stored in memory (Figure 9b), it can be seen that it is possible to obtain much more significant gains, up to 359% in throughput and 135% in CPU use efficiency.

The significant performance improvement in the reading operations deserve to be highlighted, mainly because they involve the use of the decryption functions. CBC mode also has the same advantage as CTR mode in that it can be parallelized in the decryption processes. Therefore, a simple application of the CTR mode, without exploiting the parallel processing in GPU, would not have many advantages with respect to the improvement in throughput.

The results of the sequential writing operations can be seen in Figure 10 and the values in the Table 2. In the sequential writing scenario, EncFS++ also has a significant throughput gain of up to 269%, using base FS on disk, and 324% with FS in memory. CPU utilization efficiency, is up to 212% and 133%, with base FS stored on disk and memory, respectively. In both cases, direct write to disk (`O_DIRECT`) was not used because it was not supported by EncFS. As a consequence, writing occurs first on cache pages. For this reason, the values written to disk and memory are very close and values exceed the bandwidth limit of the SSD disk.

Significant gains in both sequential reading and sequential writing demonstrate the effectiveness of context pool management techniques in order to be able to produce encryption masks well in advance. Even in a scenario of very low latency as in the case of base FS stored in memory. It is also important to highlight the gains obtained in the measurements involving 4 KiB requests, demonstrating that it is also possible to perform GPU processing efficiently when the CFS works with small amounts of data. As described in Section 3, previous research indicates that this is only feasible with larger requests (> 16 KiB).

The results of the random read can be seen in Figure 11 and the values in the Table 3.

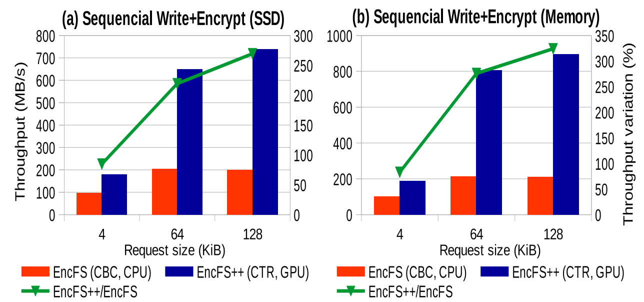


Figure 10: Throughput variation (sequential write+encrypt).

Req. Size (KiB)	Sequencial Write+Encrypt (SSD)						Sequencial Write+Encrypt (Memory)					
	EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/ EncFS		EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/ EncFS	
	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency
4	97.43	10.41	179.86	11.75	84.61	1.64	102.88	10.42	188.11	12.19	82.83	1.56
64	203.79	10.55	650.71	14.16	219.30	2.38	214.71	11.00	806.92	18.59	275.82	2.22
128	199.87	9.93	738.43	11.75	269.44	3.12	211.56	10.43	897.25	18.94	324.12	2.33

Table 2: Throughput and CPU utilization: sequential write+encrypt.

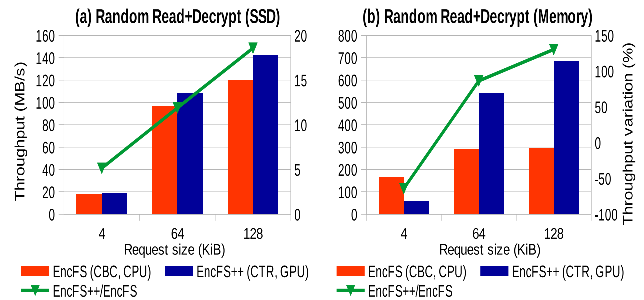


Figure 11: Throughput variation (random read+decrypt).

Req. Size (KiB)	Random Read+Decrypt (SSD)						Random Read+Decrypt (Memory)					
	EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/EncFS		EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/EncFS	
	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency
4	17.80	1.93	18.71	3.75	5.14	0.54	166.68	10.37	59.72	8.30	-64.17	0.45
64	96.48	4.37	107.95	5.35	11.88	0.91	290.74	11.40	541.54	17.12	86.26	1.24
128	120.17	5.34	142.50	5.38	18.57	1.18	297.11	11.45	684.10	17.37	130.25	1.52

Table 3: Throughput and CPU utilization: random read+decrypt.

Before discussing the results of random reading, it is important to explain how the random reading generated by the

fio tool occurs. When generating the random accesses, the tool determines the next block to be randomly accessed based on a uniform distribution. While all blocks of a file are not accessed, the reading of a previously held block does not repeat. Since the CFS was configured to encrypt blocks of 4 KiB, requests of 4, 64 and 128 KiB result in access to 1, 16 and 32 blocks, respectively.

The context pool for reading is designed to be used for both sequential and random reading. In sequential reading this pool was configured to contain 64 contexts (window size). That means, in every read request, 64 encryption masks will be produced. In sequential reading, this is not a problem, since the blocks are read in sequence and all masks are used. However, considering the random access of 4 KiB, if a window of 64 contexts is used, mostly the first mask produced is consumed, as the chances of the next block to be read in the sequence is low in random read patterns. Consequently, a significant overhead would have occurred in the production of 63 masks that are not harnessed. Therefore, in random readings, the size of the window was adapted according to the size of the request. For the 4, 64 and 128 KiB requests, windows containing 2, 8 and 16 encryption contexts were used.

First, by analyzing the results with the base FS stored in memory, in the 64 and 128 KiB requests, there are gains in throughput (86% and 130%), but they are smaller than the values obtained in sequential reading (319% and 359%). If you use smaller windows, you also reduce the level of advance that masks are generated when the request is being served. It is also important to highlight the issue of full window offset when a new random read request occurs. All window contexts need to be reset and there is less time for the masks corresponding to the initial blocks of the request to be ready, causing the application to wait for its production.

In the random 4 KiB requests and FS in memory there are significant throughput losses (64%) and low CPU use efficiency. Considering the random access where it is not possible to predict the next block to be accessed and due to the use of a small window (2 contexts), there is no way to shoot the mask production well in advance. Therefore, with each block accessed, the application needs to wait for the mask to be ready, adding latency to the decryption process. The scenario is aggravated by the fact that the base FS is stored in memory, because in this case there is neither the time to read the block on disk, in which, in parallel, the mask could be generated. In Figure 11a, it can be seen that there are no throughput losses in 4 KiB requests, which demonstrates how disk access time can be exploited as a way to compensate for processing latency in GPU kernel launches. One solution to circumvent this outcome is to either lower the latency in GPU decryption library setup or to resort to pure CPU CTR decryption when 4K random read patterns are presented to the FS.

In contrast, according to the performance observed in the 64 and 128 KiB requests, even without having the access time to disk to produce the masks, the use of a slightly larger window (8 and 16 context) and the technique of launching the production of a mask for the block n positions ahead (where n corresponds to the size of the window), are enough to achieve a certain level of advance in the production of the masks. Thus, even in this scenario of very low latency (without access to disk), we still have gains in throughput.

However, although disk access time can be exploited as a way to compensate for the latency of GPU processing, it also acts against WAESlib's ability to aggregate encryption contexts, mainly in the random read scenario. As a consequence, the efficiency in the use of GPU processing resources is reduced and the overheads inherent in GPU processing (GPU kernel launching and mask transfer) become more significant. Even with these adversities, there are still gains in this case with the base FS stored on disk, although modest, from 5% up to 18% in throughput boost.

Regarding CPU utilization efficiency, in the disk FS scenario, there is a reduction of about 46% (i.e. 0.54 efficiency) in the 4KiB requests, the worst case. In practice, this means that it would not be possible to ameliorate CPU utilization in these small requests sizes, given the throughput levels achieved. We have observed that polling mechanisms, either internal in CUDA or in the GPU encryption library, generate slightly higher CPU overhead when waiting longer for disk operations to complete. This is an issue to be further investigated. One possible solution was dicussed before, for the small 4KiB random read case from memory.

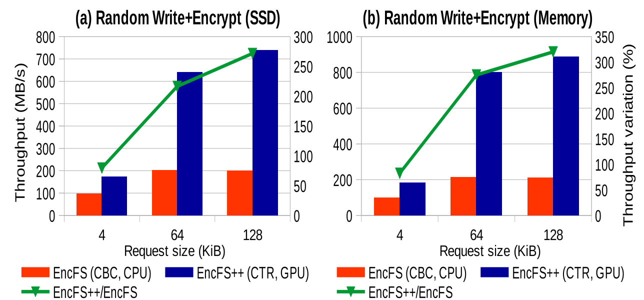


Figure 12: Throughput variation (random write+encrypt).

Req. Size (KiB)	Random Write+Encrypt (SSD)						Random Write+Encrypt (Memory)					
	EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/ EncFS		EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/ EncFS	
	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency
4	96.61	10.36	173.46	11.54	79.54	1.61	99.81	10.28	182.13	11.94	82.47	1.57
64	202.53	10.50	640.83	14.06	216.42	2.36	213.83	10.97	801.79	18.57	274.96	2.21
128	198.91	9.91	739.30	11.60	271.69	3.17	211.47	10.43	888.46	18.89	320.13	2.32

Table 4: Throughput and CPU utilization: random write+encrypt.

The results of the random writing can be seen in Figure 12 and the values in the Table 4. The outcomes in this scenario are very close to the sequential writing scenario. This is because writing occurs in memory regardless of whether the base FS is on disk or memory, as explained before. In

addition, the pool of contexts used in sequential and random writing is the same, including with respect to its mechanism of operation. There are significant gains with both disk (271%) and memory (320%) FS, citing the maximum numbers. There are also improvements in CPU efficiency of up to 217%and 132%, respectively.

On the synthetic macro-benchmark analyses, two example workloads from filebench were used: *fileserver.f* (characterized by a mix of read and writes operations) and *webservef.f* (characterized mainly by read operations). EncFS++ performance was compared to the original EncFS and with eCryptfs. The hardware accelerated (AESNI) as well as the CPU version of eCryptfs were used due to the fact that it is also stacked over *ext*4 FS (figure 2), but operating in kernel space. Measurements with simultaneous access by 1, 2 and 4 threads were performed. In this scenario, the base FS (ext4) was stored on a SSD disk.

Greater throughput gains occurred in the *webserver.f* workload (Figure 13 and Table 5). The gains are up 145% when compared to EncFS and ranges from 44% to 130% with respect to eCryptfs. In the *fileserver.f* workload, the gains were more modest: between 9% and 27% when compared to EncFS and and between 2% and 33% contrasted to eCryptfs. It is important to note that with the base FS stored in SSD, the maximum throughput supported by the disc imposes a limit to the gains. Despite the theoretical limit of 540/465 MB/s (read/write) supported by the SSD, in practice this value is 400/200 MB/s (measured by `dd` tool with `bs=128k`, same size as *iosize* parameter used in the filebench tool). Regarding CPU use efficiency, there were also gains. Compared to EncFS in the *fileserver.f* workload, these gains ranged between 120% and 170% and in *webserver.f* it was between 60% and 80%. The gains against eCryptfs in the *fileserver.f* workload stayed between 110% and 190%, and in *webserver.f* ranged from 90% to 120%.

A comparison with eCryptfs(AESNI), a CFS executing AES encryption in hardware, is a fair experiment with macro-benchmark workload. In particular because the latency at submitting encryption tasks is much lower than spawning GPU work, and this FS works in kernel space. Even in this case, the performance of EncFS++ proved competitive with respect to throughput capacity. EncFS++ showed gains between 8% and 32% in *webserver.f* workload. In the *fileserver.f* workload, the throughput practically stayed the same. The negative point in this scenario was in the efficiency of CPU usage, especially in the *webserver.f* workload where, in the worst case, there was a reduction of 60%.

Nevertheless, it is important to highlight the low GPU utilization in the experiments (Table 5). GPU utilization was only up to 25% in the *webserver.f* benchmark and around half of this amount in the *fileserver.f* workload. Also, this GPU use is lowering with more threads, as expected, because in this case, more threads competing for disk access generate more delays at this level, consequently generating less encryption work. This means that there is potential margin for scalability at the GPU side, provided bottlenecks at the FS/storage layers are resolved.

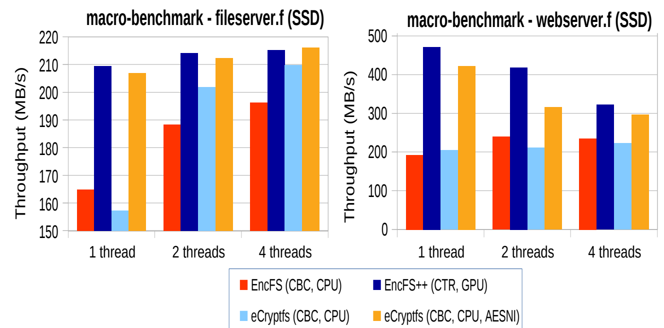


Figure 13: Macro-benchmarks (SSD).

Threads	EncFS (CBC, CPU)		eCryptfs (CBC, CPU)		eCryptfs (CBC, CPU, AESNI)		EncFS++ (CTR, GPU)			EncFS++/ EncFS(CBC)		EncFS++/ eCryptfs(CPU)		EncFS++/ eCryptfs(CPU,AESNI)	
Macro-benchmark – fileserver.f (SSD)										**Macro-benchmark – fileserver.f (SSD)**					
	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	GPU (%)	Thrput Var. (%)	CPU use efficiency	Thrput Var. (%)	CPU use efficiency	Thrput Var. (%)	CPU use efficiency
1	164,8	9,9	157,2	8,8	206,9	4,6	209,4	4,7	10,5	27,1	2,7	33,2	2,5	1,2	1,0
2	188,2	11,4	201,9	11,5	212,4	4,4	214,0	5,8	12,8	13,7	2,2	6,0	2,1	0,8	0,8
4	196,2	12,3	209,7	16,1	216,1	10,9	215,1	5,8	11,4	9,6	2,3	2,6	2,9	-0,4	1,9
Macro-benchmark – webserver.f (SSD)										**Macro-benchmark – webserver.f (SSD)**					
Threads	EncFS (CBC, CPU)		eCryptfs (CBC, CPU)		eCryptfs (CBC, CPU, AESNI)		EncFS++ (CTR, GPU)			EncFS++/ EncFS(CBC)		EncFS++/ eCryptfs(CPU)		EncFS++/ eCryptfs(CPU,AESNI)	
	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	GPU (%)	Thrput Var. (%)	CPU use efficiency	Thrput Var. (%)	CPU use efficiency	Thrput Var. (%)	CPU use efficiency
1	192,3	8,0	204,9	10,9	422,5	3,6	471,8	11,2	25,5	145,4	1,8	130,3	2,2	11,7	0,4
2	239,7	10,8	211,1	11,2	316,1	3,6	418,3	10,5	23,3	74,5	1,8	98,2	2,1	32,3	0,5
4	234,8	10,0	222,9	11,7	296,2	7,0	322,3	8,8	19,1	37,3	1,6	44,6	1,9	8,8	0,9

Table 5: Macro-benchmarks: numbers (SSD).

7 Conclusion

CFSs have intensive processing demands due to the volume of data and the cost of cryptographic functions. Parallel processing techniques on CPUs or GPUs can be used to meet these demands. As a way to achieve better performance on these systems, you can use the parallel processing capabilities offered by multiprocessor computers and servers, whether in the form of multiple CPUs or GPUs.

In the application of the encryption functions, different modes of operation can be used, among them the CTR mode. It has interesting advantages, including the ability to be fully parallelizable and allowing the generation of encryption masks in advance. This work sought to explore both, including the anticipated generation of encryption masks in the GPU. The authors of this work are unaware of previous works that have explored them in the context of CFSs, which makes our approach innovative. While this work applies such techniques with the use of GPUs for parallel encryption tasks, they could also be exploited using CPUs only or in heterogeneous (CPU+GPU) solutions.

In this work, techniques were proposed to overcome two important challenges. The first concerns the use of the CTR

mode in the context of CFSs. Issues related to the management of nonces were addressed, including techniques for their generation and storage. The main objective was to ensure that the management of the nonces did not cause a significant decrease in the performance of the CFS, which could nullify the later benefits of parallel encryption tasks.

Future work involves macro-benchmark analysis using real workloads primarily to better evaluate the performance of context pools. This will allow to identify the need for improvements in management techniques created, or even the creation of new techniques. There is also the intention to apply the approach carried out in this work on storage systems that run in the kernel space, since these systems also usually demand greater processing capacity and we can circumvent extra delays introduced by user space modules.

Acknowledgement

We thank the anonymous reviewers, in particular we appreciate the help from our shepherd, Swami Sundararaman, for their many suggestions in improving this paper.

References

[1] A. D. Biagio, A. Barenghi, G. Agosta, and G. Pelosi. Design of a parallel AES for graphics hardware using the CUDA framework. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, May 2009.

[2] Morris J. Dworkin. SP 800-38A 2001 Edition. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2001.

[3] Morris J. Dworkin. Special Publication (SP) 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2007.

[4] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010.

[5] A. Gharaibeh and S. Al-Kiswany. Crystalgpu: Transparent and efficient utilization of gpu power. *arXiv preprint arXiv:1005.1695*, 2010.

[6] Johannes Gilger, Johannes Barnickel, and Ulrike Meyer. GPU-Acceleration of Block Ciphers in the OpenSSL Cryptographic Library. In *Proceedings of the 15th International Conference on Information Security*, ISC'12, pages 338–353, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] Valient Gough. EncFS: an Encrypted Filesystem for FUSE. https://github.com/vgough/encfs, note = Accessed 07/03/2017,.

[8] Michael Austin Halcrow. eCryptfs: An enterprise-class encrypted filesystem for linux. In *Proceedings of the 2005 Linux Symposium*, pages 201–218, 2005.

[9] Owen Harrison and John Waldron. GPU Accelerated Cryptography as an OS Service. In *Transactions on Computational Science XI*, pages 104–130. Springer-Verlag, Berlin, Heidelberg, 2010.

[10] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class GPU Resource Management in the Operating System. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.

[11] A. Keromytis, J. L. Wright, and T. Raadt. The Design of the OpenBSD Cryptographic Framework. In *USENIX Annual Technical Conference (General Track)*, pages 181–196, San Antonio, 2003. USENIX Association.

[12] Wai-Kong Lee, Hon-Sang Cheong, Raphael C.-W. Phan, and Bok-Min Goi. Fast Implementation of Block Ciphers and PRNGs in Maxwell GPU Architecture. *Cluster Computing*, 19(1):335–347, March 2016.

[13] Q. Li, C. Zhong, K. Zhao, X. Mei, and X. Chu. Implementation and Analysis of AES Encryption on GPU. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 843–848, June 2012.

[14] Lin, Shang-Chieh and Liao, Yu-Cheng and Hsu, Yarsun. A Reliable and Secure GPU-Assisted File System. In Sun, Xian-he and Qu, Wenyu and Stojmenovic, Ivan and Zhou, Wanlei and Li, Zhiyang and Guo, Hua and Min, Geyong and Yang, Tingting and Wu, Yulei and Liu, Lei, editor, *Algorithms and Architectures for Parallel Processing*, pages 71–84, Cham, 2014. Springer International Publishing.

[15] Helger Lipmaa, David Wagner, and Phillip Rogaway. Comments to NIST concerning AES modes of operation: CTR-mode encryption, 2000.

[16] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[17] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53, March 2008.

[18] Naoki Nishikawa, Keisuke Iwai, and Takakazu Kurokawa. High-Performance Symmetric Block Ciphers on Multicore CPU and GPUs. *International Journal of Networking and Computing*, 2(2):251–268, 2012.

[19] Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. Fast Software AES Encryption. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption*, pages 75–93, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[20] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[21] Margara Paolo. Engine-CUDA. https://github.com/heipei/engine-cuda, 2017. Accessed 06/05/2017.

[22] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, August 2003.

[23] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2014.

[24] Weibin Sun. *Harnessing GPU Computing in System-Level Software*. PhD thesis, School of Computing, University of Utah, Utah - EUA, 2014.

[25] Weibin Sun, Robert Ricci, and Matthew L. Curry. GPUstore: Harnessing GPU Computing for Storage Systems in the OS Kernel. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 9:1–9:12, New York, NY, USA, 2012. ACM.

[26] Chien-Kai Tseng, Shang-Chieh Lin, and Yarsun Hsu. A File System Using GPU-Accelerated File-wise Reliability Scheme. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 12)*, pages 32–38, Las Vegas, 2012. CSREA Press.

[27] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of user-space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, Santa Clara, CA, 2017.

[28] W. M. Nunan Zola and L. C. E. De Bona. Parallel speculative encryption of multiple AES contexts on GPUs. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, May 2012.

Sketching Volume Capacities in Deduplicated Storage

Danny Harnik
IBM Research

Moshik Hershcovitch
IBM Research

Yosef Shatsky
IBM Systems

Amir Epstein*
Citi Innovation Lab TLV

Ronen Kat
IBM Research

Abstract

The adoption of deduplication in storage systems has introduced significant new challenges for storage management. Specifically, the physical capacities associated with volumes are no longer readily available. In this work we introduce a new approach to analyzing capacities in deduplicated storage environments. We provide sketch-based estimations of fundamental capacity measures required for managing a storage system: How much physical space would be reclaimed if a volume or group of volumes were to be removed from a system (the *reclaimable* capacity) and how much of the physical space should be attributed to each of the volumes in the system (the *attributed* capacity). Our methods also support capacity queries for volume groups across multiple storage systems, e.g., how much capacity would a volume group consume after being migrated to another storage system? We provide analytical accuracy guarantees for our estimations as well as empirical evaluations. Our technology is integrated into a prominent all-flash storage array and exhibits high performance even for very large systems. We also demonstrate how this method opens the door for performing placement decisions at the data center level and obtaining insights on deduplication in the field.

1 Introduction

The rise of all-flash storage arrays has also brought deduplication technology to the forefront and many prominent all-flash systems now support deduplication across entire systems or data pools (e.g., [2–5, 7, 9]). While this shift helped reduce storage costs, it also created new storage management challenges for storage administrators. This work focuses on technologies and tools for managing storage capacities in storage systems with deduplication.

Storage Management and Deduplication: Volume placement and capacity management were challenging yet well understood management tasks before deduplication was introduced. A storage volume needs to be allocated appropriate resources and connectivity. In large data centers, spanning multiple storage arrays, managing where to place volumes optimally is a tricky task. It involves satisfying two main measures that characterize a volume: its capacity and workload (IOPS/throughput). The main problem tackled in this paper is that once deduplication is brought in to the equation, the capacity of a volume is no longer a known quantity. Hence a storage administrator is left without clarity about one of the main resources that he needs to manage.

Our solution serves a number of appealing applications that are otherwise hard to accomplish in a deduplicated setting. In a recent paper titled “99 deduplication problems" [31], Shilane et al. present some burning deduplication related problems that need to be addressed. Our methods turn out to be helpful in solving three of the five problem classes that are discussed in this paper (it was actually 5 rather than 99 problems...). In particular, our solution is relevant to the issues of: 1) understanding capacities, 2) storage management and 3) tenant chargeback.

Why is managing volumes with deduplication hard? Volume level capacity statistics are the primary tools for managing the system capacity. However, in a system with deduplication those statistics are no longer naturally available. There are two different aspects that are the main reasons for this:

1. The first is that once cross-volume deduplication is enabled, it is no longer clear which volume owns what data. This brings up a conceptual question of what should actually be reported to the storage administrator? In Section 3.2 we discuss in detail and define what capacity can be attributed to a volume and why this information is useful. More importantly, we point out that a critical and well-defined question about a volume is how much space will be freed in case this volume was removed from the system (termed the *reclaimable* space of a volume in this paper). Note that with deduplication enabled, one could possibly remove the largest volume in the system, yet

*Work was conducted while at IBM Research.

not free up a single byte of user data from the storage.

2. The second aspect is a pure computational challenge: once we decide what we want to report, how can this number be calculated in a typical architecture of a storage system with deduplication. There is a fundamental difference between capacity statistics in the presence of deduplication and traditional capacity statistics (in traditional statistics we also include those of storage that supports compression only without deduplication).
   Traditional statistics are all additive and can be aggregated per each volume - i.e., hold a counter of how much space was held by a volume and update this on every write/overwrite/delete operation.[1] On the other hand, in the case of storage with deduplication, the capacity statistics of a single volume do not depend solely on what happens in this specific volume, but rather could be affected by any write operation to any volume in the system (as long as they are part of the same deduplication domain). Moreover, the reclaimable statistic of a group of volumes is not additive - i.e. the reclaimable space of removing two different volumes does not equal the sum of the reclaimable quantities of the two volumes separately. As a result, the methods for calculating such statistics are much harder than traditional stats, and near impossible to do if considering any arbitrary combination of volumes.
   That being said, it is clear that a storage array holds all of the information required to actually compute these numbers. It is just that the sheer amount of metadata that needs to be analyzed to produce these statistics is too large to actually analyze with acceptable resources (CPU and memory).

Our Work - Sketching Capacities: We present a novel approach to produce capacity statistics in a deduplicated storage system for any single volume or any combination of volumes. Moreover, our approach can answer complex placement questions, e.g., not only do we answer how much space would be reclaimed when moving an arbitrary set of volumes out of a system, we can answer how much space this set would take up at a different deduplicated storage system (which holds different content than the original system).

At the core of our solution is the decision to forgo the attempt to produce accurate statistics. Rather, we settle for estimations of the capacity statistics as long as we can gauge the accuracy of these estimations. We borrow techniques from the realm of streaming algorithms in which the metadata of each volume is sampled using a content-based sampling technique to produce a so-called *sketch* (or *capacity sketch*) of the volume. The key is that the sketch is much smaller than the actual metadata, yet contains enough information to evaluate the volumes capacity properties in conjunction with any other set of volume sketches. To illustrate this, consider a storage system holding 1 PB of data. In order to manage such a huge amount of data, a system has to hold a very large amount of metadata which could be on the order of 10 TB (depending on the specific design). In contrast, our sketch data for such a system takes under 300 MB, which makes our statistics calculations easily manageable. Part of this technology is integrated into the IBM FlashSystem A9000/A9000R.

Main Contributions: In the paper we provide details of the technique, its accuracy statement and a description of our implementation. In our design, sketch data is collected by the storage system and pulled out of the system to an adjacent management server where it is analyzed and presented to the administrator. Our implementation includes the following:

- Provides reclaimable capacities and attributed capacities for any volume in the system.
- Supports queries of these capacities on any arbitrary group of volumes within a deduplication domain (an option that to the best of our knowledge is not available in any system to date).
- In a multi-system environment, we answer how much physical space such a volume/group would consume if it were to be migrated to another deduplicated storage system (with different content).

The implementation is optimized for high performance, providing a real-time user-experience. After initial extraction of the sketch and ingestion into the sketch analyzer, we can answer queries in well under a second.

The high performance is also key for providing next level optimization of management functions. We present an example of a greedy solution for multi-system space reclamation optimization. The algorithm creates a migration plan from a full source system onto the other systems in a way that optimizes overall capacity utilization.

2 Background and Related Work

Deduplication is a form of compression in which duplicate chunks of data are replaced by pointers to the original repeating data chunk. This practice can greatly reduce the amount of physical space required to store a large data repository, depending on the amount of repetitiveness of the data chunks. Deduplication is typically used in concert with traditional “local" compression. Unlike deduplication, which looks for repeating data across an entire repository of data, in compression a single data chunk or block is compressed on its own (typically using a techniques such as Zip [13, 25, 35]). To measure data reduction, we use the convention by which the data reduction ratio is the size of the data after reduction divided by the size of data before reduction (so 1 means no compression and close to 0 is highly compressible). Deduplication can consider fixed size data chunks or variable size

[1] It should be noted that while this is conceptually easy, many times it collecting these statistics requires complex engineering, especially in highly distributed storage systems.

chunks. In our work we refer to systems that use fixed size chunks (we use chunks of size 8KB), but our techniques can generalize nicely to variable sized chunking as well.

The common technique for performing deduplication is via chunk fingerprinting. Namely, for each data chunk a hash of its content is calculated, creating an identifier of the data. If a strong cryptographic hash is used, then for all practical purposes this hash is considered a unique identifier of the content. Duplications are found by holding a database of chunk fingerprints and finding repeating fingerprints across the entire data set.

Related Work: Variations of content-based sampling have been deployed in the context of deduplication for various tasks. Mainly for identifying deduplication potential in data sets that have not yet been deduplicated (e.g. [17, 23, 24, 34]), for finding repetitive streams/files as part of the actual deduplication process (e.g. [10,26]) or for automatic routing of data in a distributed setting [12, 14, 18, 19]. Accuracy guarantees for data reduction estimations have been explored in [22–24, 34]. These works focused on analyzing data that has not been deduplicated for assessing their potential data reduction and sizing of the storage required to store them.

In contrast, our aim is to address gaps for reporting and management of data that *has already been deduplicated*, which prior works do not address. We use similar techniques, but the application presents different requirements and hence we deploy slightly different practices. The idea of content based sampling dates back to the classical algorithm of Flajolet and Martin [16] and was thoroughly studied in the context of streaming algorithms for estimating the number of distinct elements in a data stream, a problem which is similar to estimating the amount of distinct data chunks in a data set. We use a variation of a method introduced by Gibbons and Tirthapura [20] and by Bar Josef et al. [11]. In the deduplication context, a similar technique was used by Xie et al. [34] who use filtering according to hash values. The main technical difference between the sketch that they use and ours is that Xie et al. always keep a bounded sample of hashes, a practice that cannot provide adequate accuracy in our context. There is also a significant difference in what the sketches are eventually used for. We also extend the methods to handle the combination of compression with deduplication and provide a different approach to the accuracy analysis (See Section 4). In the storage realm, variants of sketches on the space of LBA addresses have also been used by Wires at al. [33] to estimate amount of exclusive blocks in a snapshot and by Waldspurger et al. [32] for simulating cache behaviors.

There have also been studies tackling the problem of freeing space from a deduplicated system [30] or balancing of content between several deduplicated systems [15, 27]. These suggest various heuristics to perform such optimizations, but largely avoid the question of how to actually learn the interplay between various volumes and assume that this is a given quantity and is computed as a preprocessing step. As such, our work is very suitable to work together with any of these methods.[2]

3 Sketches and their Use

Capacity Sketches: The main idea of capacity sketches is to choose samples of the metadata according to the respective data content. At its core, the sampling technique is very simple: for each data chunk, examine its fingerprint (the hash of its content) and include it in the sketch only if it contains k leading zeros for a parameter k (namely, the k most-bits of the hash are all zeros). So if, for example, $k = 13$ and the fingerprints are random, then on average a $\frac{1}{2^{13}} = \frac{1}{8192}$ fraction of the data chunks participate in the sketch and the rest are ignored. We refer to 2^k as the *sketch factor* and in our implementation we typically set the sketch factor to be 8192. This choice was made by balancing the tradeoffs between the required resources to handle the sketches vs. the accuracy which they provide (See Section 4).

Denote by $\mathcal{S}$ a data set that corresponds to a volume/group/system, denote by $Sketch_{\mathcal{S}}$ the set of hashes which were included in the sketch of the data set and denote by $Space_{\mathcal{S}}$ the physical space required for storing $\mathcal{S}$ ($Space_{\mathcal{S}}$ is the value that we aim to estimate). Denote by $Written_{\mathcal{S}}$ the amount of logical data written by the user (prior to compression and deduplication). For each hash $h \in Sketch_{\mathcal{S}}$ in the sketch we also hold the chunk's compression ratio – denoted by $CompRatio(h)$. We estimate the number of unique chunks in $\mathcal{S}$ by the amount of chunks that participate in the sketch times 2^k, namely by $2^k \cdot |Sketch_{\mathcal{S}}|$. The estimated amount of space required for storing this data set in a clean system is:

$$\widehat{Space_{\mathcal{S}}} = 2^k \cdot \sum_{h \in Sketch} CompRatio(h) \cdot ChunkSize$$

In such a case the estimated data reduction ratio (combining both deduplication and compression) is :

$$ReductionRatio_{\mathcal{S}} = \frac{\widehat{Space_{\mathcal{S}}}}{Written_{\mathcal{S}}}$$

This is the basis for a sketch-based estimation of a single set.[3] In the following sections we describe how to use sketches for estimating volume level statistics in an existing deduplicated storage system. In Section 4 we discuss the accuracy of these estimations as a function of the size of the sketch and the sketch factor.

[2] Note that these heuristics are typically based solely on the knowledge of the pair-wise deduplication relations between volumes but ignore the numbers about a combination of a larger number of volumes. This is mainly because computing such quantities is extremely taxing. Our techniques open the door to utilizing much more information than just pair-wise information.

[3] Note that the same estimation method for $\widehat{Space_{\mathcal{S}}}$ holds for variable sized chunking, only that then *ChunkSize* is also a function off the hash h.

Using Sketches for Data Inside a Deduplicated System: When discussing the statistics of volumes or groups with respect to a storage system, additional challenges arise. Unlike the stand-alone case, the statistics of a volume or group do not depend solely on the contents (or the sketch) of this single data set. Rather, they depend on the contents of all of the data in the system and may change even though the volume itself observed no changes at all. To facilitate efficient computation of the statistics for a live existing system, we maintain at all time a full system sketch (denoted $Sketch_{FULL}$) representing all of the data in the system. We also collect further parameters in the sketch. Specifically, for each fingerprint $h \in Sketch_{\mathcal{S}}$ the sketch holds:

- Reference count - Denoted $Ref(h,\mathcal{S})$ is the number of times the data chunk with fingerprint h was written in the data set by the user.
- Physical count - Denoted $Phys(h,\mathcal{S})$ is the number of physical copies stored during writes to the data set $\mathcal{S}$. In contrast to reference count which refers to the virtual space, this counter refers to how many virtual copies of this chunk where eventually written in the physical space. Note that there are a number of reasons for a chunk to have more than a single physical copy in a system. Most obvious is deduplication opportunities that were not identified. But sometimes this is done out of choice, e.g., as a means to limit the size of the reference counters, or the choice to forgo a deduplication opportunity for avoiding extensive data fragmentation.

With this additional information we can calculate the statistics of a volume or group as part of a larger system. It should be noted that a real deduplicated system may also hold quite a bit of *unhashed* data (data written at IO's smaller than a single chunk size, or misaligned with the deduplication chunk alignment). We use various techniques to account for such data, but this is out of the scope of this paper. We now describe the main estimations that we calculate as well as their motivation.

3.1 Reclaimable Capacity

As mentioned in the introduction, a key product of our method is the ability to accurately predict how much physical space would be freed from a system if a volume or a group of volumes were to be removed from it. Note that the reclaimable capacity is an inherently non-linear quantity. For example, if a system contains just two identical volumes, then the reclaimable capacity of each of the volumes separately is essentially 0, yet the reclaimable capacity of their combination amounts to the system's entire space which is very different than the sum of their respective reclaimable numbers. As such, some deduplicated storage vendors do not produce such a number at all. Others (e.g. [8]) resort to reporting how much *unique* data a volume holds.[4] This number is additive and is easier to maintain, but can be very misleading when a volume holds internal deduplication, a situation that is magnified when trying to estimate the reclaimable of a group of volumes.

[4] Unique data counts only data chunks that have reference count $= 1$.

Our strategy for estimating the reclaimable capacity consists of "subtracting" the sketch of the data set being examined from the full system sketch as follows:

Calculate Reclaimable

Input: $Sketch_{\mathcal{S}}$, $Sketch_{FULL}$
$Reclaimable = 0$
for $h \in sketch_{\mathcal{S}}$ do:
  if $(ref(h,\mathcal{S}) == ref(h,FULL))$ then
    $Reclaimable \mathrel{+}= CompRatio(h) \cdot Phys(h,FULL)$
$Reclaimable = Reclaimable \cdot SketchFactor \cdot ChunkSize$

While the algorithm above gives the general idea, there are some additional subtleties that need to be addressed. For instance, each chunk held in the system also holds some metadata associated with it. While this is typically much smaller than the data itself, it can amount to a significant portion of the space, especially for highly compressed or deduplicated data. So reclaimable space should account for metadata space that is released when a chunk is removed (whether it was a physical chunk or a reference). Another subtlety is the fact that it is hard to gauge if a physical chunk would be released when its physical count is two or more. Handling this requires additional information from the system, but for the most part tends to account for a very small portion of the physical space.

3.2 Attributed Capacity and Data Reduction Ratios

Unlike the reclaimable statistic which is very clearly defined, it is not straightforward to define the data reduction ratio of a volume, or what capacity is owned by a volume. This is because data is shared across volumes and has no clear owner. Still, there are a number of motivating reasons to define and support such numbers. The first reason is the possibility to do fair chargeback of tenant capacities for service providers (see discussion in [31], Section 6). Another reason is to allow the storage administrator to understand the data reduction properties of volumes – how much is a volume involved in deduplication? how much does it gain from compression? Such knowledge can allow better planning of storage capacities (e.g., an administrator can learn what data reduction to expect from her Databases), and better placement decisions (e.g., a volume that has no data reduction saving can be placed in a system that does not support data reduction).

To that end, we define a measure that we call the *attributed* capacity of a volume and a breakdown of its space savings to deduplication and compression. Our definition follows a fair sharing principle: Data which is shared among a number of volumes will receive its proportionate share in attributed capacity. For example, if a data chunk is only referenced twice, once in each of two volumes, then the space to hold

this chunk is split evenly in the attributed capacity of the corresponding volumes. If it has 3 references, 2 originating from volume A and one from volume B, then its space is split in a $\frac{2}{3}$ and $\frac{1}{3}$ fashion between volumes A and B respectively. Note that there is no single correct definition of attributed capacity, but rather a choice of what the vendor deems as fair sharing. Our sketches approach can accommodate more or less any definition.[5]

For the breakdown to deduplication and compression we define the following: Deduplication savings are an estimate of what the savings would have been if compression was not deployed. For compression we give a different estimate, basically answering how much additional space was saved after deduplication was performed. This does not answer the question of how much space savings we would gain if only compression was performed (without deduplication). In order to answer the latter question, one needs to sample the virtual space of data (as described in [22]), rather than sample the fingerprint space which is what our sketch does.

The following method is used for attributed space and deduplication savings:

Calculate Attributed

Input: $Sketch_{\mathcal{S}}$, $Sketch_{FULL}$
$Attributed = 0$
$DedupeOnly = 0$
for $h \in sketch_{\mathcal{S}}$ do:
$\quad Attributed \mathrel{+}= \frac{ref(h,\mathcal{S})}{ref(h,FULL)} \cdot CompRatio(h) \cdot Phys(h,FULL)$
$\quad DedupeOnly \mathrel{+}= \frac{ref(h,\mathcal{S})}{ref(h,FULL)} \cdot Phys(h,FULL)$
$Attributed = Attributed \cdot SketchFactor \cdot ChunkSize$
$DedupeOnly = DedupeOnly \cdot SketchFactor \cdot ChunkSize$

3.3 Insights on the Achieved Deduplication

An additional benefit for our methodology is the ability to collect drill down statistics regarding deduplication and compression at a very low price. For example, we collect statistics regarding the effectiveness of the deduplication in the storage system. Another set of interesting statistics is the correlation between deduplication and compression, this can be done at a volume granularity, as well as at the single chunk granularity (e.g. is there a correlation between the reference count of a chunk and its compression ratio?). A summary of such insights is sent back to us via a call-home mechanism, and will serve as a mechanism for collecting information from the field about the deduplication properties of real data in the field in order to improve the deduplication process and design.

Explaining deduplication behavior: It is not uncommon for gaps between the customer expectation of deduplication and compression effectiveness versus the reality. In many cases the gap arises from the data written to the system. The ability to correlate hashes between volumes very quickly, and identify correlations and anti-correlations can provide the explanation of why certain deduplication and compression ratios are achieved.

[5] Having said that, it makes sense to use a definition that allows for correct accumulation of attributed capacity between any set of volumes.

3.4 Cross System Capacity Estimations

In a data center environment spanning a number of storage systems, the question of space reclamation from one system is accompanied by the question of where to move the volume to? The goal here is to provide insight into the overall capacity management of the data center rather than just managing a single system; namely, can I gain capacity by moving data between systems? The use of sketches allows to answer the capacity aspect of such complex “what if" questions. Specifically, how much capacity would be freed when moving a volume from system A to another system, and how much capacity would this volume potentially consume in each of the target migration systems. This question can be answered given a sketch for a data set $\mathcal{S}$ and a full sketch of a target system using the following method:

Calculate Space in Target System

Input: $Sketch_{\mathcal{S}}$, $Sketch_{TARGET}$
$TargetSpace = 0$
for $h \in sketch_{\mathcal{S}}$ do:
$\quad$ if $h \notin Sketch_{TARGET}$ then
$\quad\quad TargetSpace \mathrel{+}= CompRatio(h)$
$TargetSpace = TargetSpace \cdot SketchFactor \cdot ChunkSize$

Data Center Space Optimizations: Such a cross system estimation presents a strong tool for performing optimizations across multiple systems. Without clarity of the capacity required to store a volume on various systems, such decisions are made in the blind. Our technique provides clarity and allows us to explore optimizations and advanced placement, rebalancing and space reclamation decisions. In Section 6.4 we present an example of such an optimization for data center level space reclamation.

4 Accuracy Guarantees

A crucial property of sketches is the ability to make concrete statements regarding its accuracy, hence allowing decision makers to make educated decisions with confidence while taking into account known error margins. To this end, we provide a mathematically proven statistical accuracy theorem. We also evaluate this guarantee empirically and see that the actual estimations indeed behave according to the mathematical statement (see Section 6.3).

As is common in statistical statements, we have two parameters: the error (or skew) denoted by ε and the confidence denoted by δ. The formal statement says that the estimation will be off by error greater than ε with probability no larger than δ. It turns out that the key parameter relating between

ε and δ is the size of the physical space being estimated – whether it is reclaimable, attributed, or the space required for storing $\mathcal{S}$ in a deduplicated and compressed system. In the following statement we simply use $Space_{\mathcal{S}}$ to denote this size.

Theorem 1. *Let $\mathcal{S}$ be a data set whose physical space (after deduplication and compression) is $Space_{\mathcal{S}}$ and let $\widehat{Space_{\mathcal{S}}}$ be a sketch-based estimation of this space using a random hash function. Then the probability of over-estimation:*

$$Pr[\widehat{Space_{\mathcal{S}}} > (1+\varepsilon)Space_{\mathcal{S}}] < \left(\frac{e^{\varepsilon}}{(1+\varepsilon)^{(1+\varepsilon)}}\right)^{\frac{Space_{\mathcal{S}}}{ChunkSize \cdot SketchFactor}}$$

and the probability of under-estimation:

$$Pr[\widehat{Space_{\mathcal{S}}} < (1-\varepsilon)Space_{\mathcal{S}}] < \left(\frac{e^{-\varepsilon}}{(1-\varepsilon)^{(1-\varepsilon)}}\right)^{\frac{Space_{\mathcal{S}}}{ChunkSize \cdot SketchFactor}}$$

The theorem follows from the classical multiplicative variant of the Chernoff Bound (e.g., in [29]). However, we needed to reprove a more generalized form of this bound in order to capture variations including compression ratios, variable sized chunking, reclaimable and attributed. Note that [34] use a different method to achieve their accuracy guarantee. Their guarantee relies on the estimation of Bernoulli variables by a normal distribution, which for smaller numbers may add some noise. Our estimation avoids this and turns out to be slightly more conservative in the guarantees it provides.

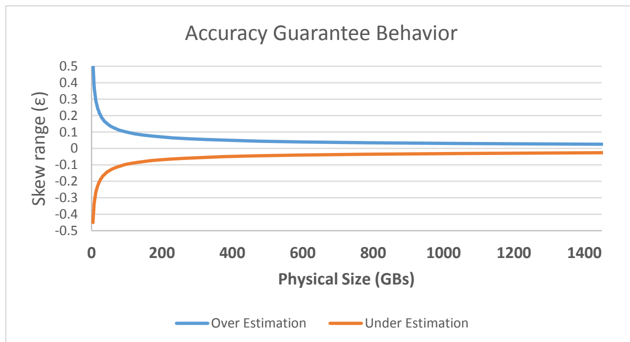


Figure 1: The behavior of the accuracy guarantee ε as a function of the physical size for a fixed choice of the confidence $\delta = \frac{1}{2000}$. The smallest values in the graph are at 4 GBs.

Figure 1 depicts the behavior of our bound for a fixed choice of confidence δ. Note that we need to consider a δ which is small enough to account for a large number of volumes and sets being tested. For example, if we evaluate 750 volumes, using $\delta = \frac{1}{100}$ is not sufficient, as we expect on average 7.5 volumes to exceed the error that corresponds to $\delta = \frac{1}{100}$. In our evaluations we typically use $\delta = \frac{1}{2000}$, but this should be adapted depending on the circumstances.

Using Theorem 1: The goal of the mathematical guarantee is to produce an estimation together with a range for which we can say with confidence that the actual value resides in. For example, we estimate that the reclaimable of a volume is 200GB +/- 14 GB. To this end, for a given fixed confidence parameter δ we create an inverse lookup table that on input $Space_{\mathcal{S}}$ returns the corresponding $\varepsilon(Space_{\mathcal{S}})$. We can then return a +/- value of $Space_{\mathcal{S}} \cdot \varepsilon(Space_{\mathcal{S}})$. One subtlety, however, is that the bound above is dictated by the actual physical space $Space_{\mathcal{S}}$ whereas we only have the estimation of this value $\widehat{Space_{\mathcal{S}}}$. Therefore, in order to get an accurate ε one has to find what is the smallest $Space_{\mathcal{S}}$ such that $\widehat{Space_{\mathcal{S}}} > Space_{\mathcal{S}} - Space_{\mathcal{S}} \cdot \varepsilon(Space_{\mathcal{S}})$. Note that this is important only for evaluating over-estimations, since it turns out that the function $Space_{\mathcal{S}} \cdot \varepsilon(Space_{\mathcal{S}})$ is monotonically increasing with $Space_{\mathcal{S}}$. We also note that the difference between this method and simply returning $\widehat{Space_{\mathcal{S}}} \cdot \varepsilon(\widehat{Space_{\mathcal{S}}})$ is only noticeable for small volumes.

The Effect of the Sketch Factor: The sketch factor appears in Theorem 1 as a divisor of the actual space. This means that, for example, moving from sketch factor 2^k to sketch factor 2^{k+1} will shift the same accuracy guarantees to volumes that are double the size. On the other hand, the amount of sketch data to handle will be cut in half. We arrived at our choice of $k = 8192$ by taking a sketch factor that is high enough to ensure good performance and low overheads, yet still give acceptable accuracy guarantees.

Handling Small Volumes: The accuracy we can achieve when estimating small volumes is limited (and more precisely, volumes of small physical space). For example, the guarantee for a volume of physical size 50GB is only $\varepsilon = 0.14$. There are a number of points to consider here:

- The virtual capacity of the volume is important in understanding if the estimation is worthwhile. For example, for an estimation in the range of 2GB we can only say with confidence that the value is in the range between 0.5GB and 4.2GB. This is not saying much if the volume's virtual size is 4GB but contains very valuable information if the volume's virtual size is 100GB. In the latter case it means that the reclaimable space of the volume is just a small fraction of the original volume (and is a bad candidate for space reclamation).
- Small logical volumes gain very little from sketches (except in very extreme cases, e.g. if the volume is very compressible). It could be argued that small volumes are not very interesting from a capacity management perspective since they have very little impact. On the other hand, grouping several small volumes together to form a larger group is highly recommended. The sketch merge functionally accommodates this and accuracy improves as the size of the merged data set increases.

5 Architecture and Implementation

We turn to describe our actual implementation and integration of the sketch-based capacity analytics for a production storage system. Our overall strategy is to pull the sketch data out of the storage system onto an adjacent management server where the sketch data is analyzed and the outcome is displayed to the storage client (See Figure 2). The choice to do the analysis outside of the storage box has a number of reasons. For one, this avoids using CPU and memory resources in the storage that could otherwise be spent on serving IO requests. But more importantly, it is the optimal location for managing cross system placement options (such as the ones discussed in Section 3.4). As such, our design has two separate components: the *sketch collection* embedded in the storage system and the *sketch analysis* running on an external server. We next describe our design and implementation of these two components.

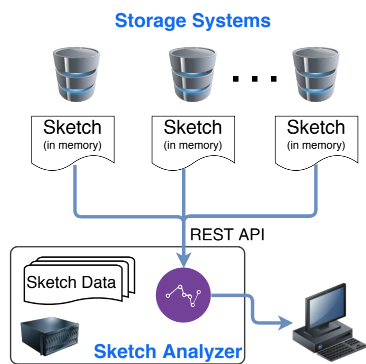


Figure 2: The general sketches support architecture.

5.1 Sketch Collection

There are multiple approaches that can be used for sketch collection. One can use a bump in the wire approach that directs all sketch information as data is written to a sketch collection mechanism. However, in such a design, support for updating the sketch should also be added for deletion or data overwrites which makes this harder to maintain. In addition, in a highly distributed storage system such as ours, it is unclear where the sketch collector should run and if it should be likewise distributed. Another approach is to do an offline metadata scan to extract the actual sketches. In this approach all metadata in the system must be read, and since the metadata is typically paged in and out of memory, such a scan can be relatively slow and may have a negative performance impact on the storage system by introducing additional reads from disk. Instead, we use a third variant which is somewhat of a medium of the two aforementioned approaches.

Our design has the following key principles:

- All sketch data is held in memory at all times - This allows to retrieve this data swiftly, but more crucially avoids adding IOs to the disk backend for sketch update and retrieval.
- Each process is in charge of the sketch data for its jurisdiction - Our storage system is highly distributed, with hundreds of processes working in parallel to serve IOs. Each process is in charge of serving IOs for slices of the entire virtual space. In our design each slice has its own sketch which is maintained by the owning process.
- The sketch portrays the state of a slice at a point in time - The sketch data is held and managed as an integral part of the metadata of a slice, and therefore there is no history of writes and deletes as part of the sketch.

These principles are achieved using the following methodology: During sketch retrieval, if the metadata of a slice happens to be in memory, then the sketch data for this slice is extracted directly from the slice metadata. Whenever the metadata of a slice is paged out of memory, its sketch data is kept “alive" in a designated memory area and retrieved from there.

For the act of sketch extraction, a central process contacts all processes and retrieves their respective sketch data. These are streamed out of the system to the adjacent server. Note that the sketch data in the storage is always in distributed form, and the aggregation of this data only takes place outside the system once the sketch data has been extracted. It should also be noted that as in many cases for distributed systems, the extracted sketch does not actually reflect a single point in time. In our case, the sketch provides a fuzzy state, e.g., when we actually obtain the sketch for the last slice, the sketch in the system for the early slices might have changed. This is an inaccuracy that we are willing to tolerate since storage systems are dynamic and we cannot expect to freeze them at a specific state. That being said, the fact that we can serve the sketches quickly from memory is a considerable advantage as it can reduce the time window in which the sketches are extracted.

Extensive performance tests were run to ensure that our sketch collection and retrieval mechanisms do not interfere with the performance of the storage system and the effects are unnoticeable even at peak performance.[6] In order to minimize the memory footprint, we hold the sketches in packed format and the size of each element in the sketch is limited to 19 bytes. This includes volume information, compression ratio, reference and physical counters and 8 Bytes for the actual hash value, truncated down from 20 bytes of a full SHA1 hash (we take 8 bytes that do not include the leading zero bits). We point out that while an 8 byte hash is not enough for avoiding collisions in a deduplicated system, it is well

[6]The retrieval process is throttled to ensure it does not interfere with the systems IO chores.

suited for achieving high accuracy estimations.[7] Overall this means that for 1PB of user data the sketch data will amount to approximately 300MB on average.

5.2 The Sketch Analyzer

The statistics provided by our sketch analysis do not reflect a real time state of the system. Rather, they reflect a fuzzy state of the storage over the sketch retrieval time duration. Additionally, we can provide the resulting statistics only after the sketch has been fully transferred and processed by the analyzer. That being said, we invest quite a bit of effort to make our sketch analysis as fast as possible, for a couple of reasons: 1) In order to be interactive and support online statistics queries on arbitrary volumes groups by the storage administrator. Our aim is to provide a real-time user experience for this, and indeed we manage to answer all queries well within one second; and 2) Using our tools for performing optimizations typically entails performing a very large number of queries, and therefore the fast processing of queries on sketches allows such optimizations to be feasible.

Recall that the sketch extracted from the storage arrives as a stream in its distributed form. It contains hardly any aggregation at all, and therefore the first phase that we need to do is to ingest it (including sorting to volumes and aggregation). The next phase is the actual analysis using the methods described in Section 3.

The Ingest Phase: The first phase of the process is therefore an ingestion phase. In a nutshell, the sketch contains a stream of hash values along with their respective compression ratios, a local reference count (within the slice), an indication if this was written as a physical copy or just a reference, and finally the name of the owning volume. For each volume we need to collect all of its relevant hashes while merging and aggregating multiple appearances of hashes. The same applies toward creating the full system sketch.

In order to accommodate this, we create two types of data structures at ingest time:

1. **The full system table:** An open addressing hash table holding all of the hashes seen in the full system (including the compression ratio, reference counts and physical copies count). We use statistics from the system to estimate the size of this table and allocate the memory for this table accordingly.
2. **Volume level structures:** We hold a temporary B-Tree for each volume in the system which aggregates the hashes that appeared in the respective volume (along with the reference count for each volume). At the end of the ingest phase we replace each B-Tree with an immutable sorted array, which is a more space efficient data structure which will support faster volume merges.

[7] Under the randomness of SHA1, a false collision of 8 bytes in the sketch data would occur on average once on every 256PB of logical written data.

Our sketch analyzer is designed as a micro-service and uses a REST API to receive the raw sketch data and to then answer statistics queries.

The implementation of the REST interface is in Python, but the core data structures are implemented in C++, using the C-types interface. The implementation in C++ is critical for achieving the performance that we require and for minimizing the memory utilization of the sketch analyzer. For space and time optimization we leveraged the following implementation details:

Space optimization: instead of holding the full hash in the B-Trees and volume arrays, they are held only once in the full-system table, and in the volume level structures we only store a pointer to the entry in the full table (the pointer takes just 5 bytes rather than 8 bytes for the sketch hash).

Sketch distribution and concurrency: Each of our data structures is divided in to 16 partitions, each handling on average $\frac{1}{16}$ of the hashes in the sketch, depending on the first four bits of the hash value. This allows for analysis concurrency as each (hash range) partition can be independently analyzed. Keep in mind that due to the randomness of the hash function, it is expected that each such partition will receive a fair share of the load.[8] In addition, the partitioning provides easy sparsification of the sketch. Depending on the query, a higher sketch factor than 8192 may be sufficient for allowing faster computations (by handling smaller sketches). Simply working with j out of the 16 partitions can easily allow us to work with a sketch factor of $\frac{16}{j} \cdot 8192$.

The Analysis Phase: The basic analysis phase consists of computing the reclaimable and attributed statistics for all volumes in a system. In addition, we implemented support for running these queries on any arbitrary group of volumes and the ability to query cross system migration costs for any group of volumes. We emphasize that we consider this phase as the most performance critical phase, since it should support interactive administrator queries that should be satisfied online to deliver a favorable user experience. In addition, the performance of grouping and merging is critical for the next level optimization algorithms as discussed in Section 3.4.

The basic functionality is straightforward and tightly follows the methods described in Section 3. For the group queries, however, an additional step is required to generate the sketch of a newly defined group. This process receives a list of volumes that form the group and merges them into a single sketch (reference counts and physical counts are summed in the merge, whereas compression ratio is averaged). To this end we implemented a classical heap-based k-way merge of sorted arrays (e.g., see [21]).[9]

[8] A different approach would be to run the analysis for many volumes/groups in parallel. However, the volumes can be very different in size which could create a strong imbalance between the processes and require more complex load sharing between the processes.

[9] Recall that at this point the volume sketches are held in sorted arrays.

Test Name	Data Written (TB)	Number of volumes	Ingest time (sec)	Analysis time (sec)
UBC-Dedup	63	768	22	0.21
Synthetic	1500	5	89	0.93
Customer System 1	980	3400	104	4.80
Customer System 2	505	540	65	2.70

Table 1: Performance of the ingest and analysis phase

6 Evaluation

6.1 Methods and Workloads

We used a number of methodologies to evaluate our sketches implementation, each with its own workloads. We describe these below:

1. **Synthetic data** - These are end-to-end tests performed in our lab in which various sets of synthetic data were written to the storage system, the sketches were extracted by the adjacent manager server and analyzed with the sketch analyzer. The tests evaluate both the performance of the mechanism as well as the accuracy of the results. The data was crafted in a way that allows us to predict the expected outcome and evaluate it. In addition we also ran tests that delete volumes from the system and compare the space that was released to the reclaimable that was predicted by the sketches.
   The data was generated using the following methodology: a number of equally sized data units were created, each with a different chosen compression ratio and with *no deduplication*. These were generated using the VDBench benchmarking suite [1]. We then wrote a number of data volumes to the storage, each consisting of a chosen set of units. Deduplication was created by reusing the same data units in different volumes, or repeating in inside the same volume. Our tests were of various sizes, ranging from small tests with data units of size 100GB each, to a large scale test in which 1.5 PB of data was written to the storage system, using data units of size 100TB each.
2. **UBC data traces** We leveraged the data trace called UBC-Dedup from the SNIA's IOTTA repository [6] (The Input/Output Traces, Tools, and Analysis repository). This are traces that were collected for the study of Meyer and Boloski [28], spanning 852 file systems from Microsoft in the form of hashes of the data and some related metadata (compression ratios are not available in this trace). After cleaning some small file systems, we ended up with hashes for 768 file systems representing 63 TBs of data.[10] The traces offers several chunking options and we used the fixed 8KB chunks. We use these file systems to simulate volumes in a storage system and evaluate our sketch analyzer both from a performance standpoint and from an accuracy standpoint. Note that these tests are not an end-to-end evaluation as real data was not involved. Rather, from the full hash traces we generated a much smaller sketch and ingested it into our sketch analyzer.
3. **Production data in the field:** The product implementation allows us to gain some insights via call-home data. While this data is very succinct and contains only general statistics, we learn from it about the performance of the sketch analyzer and can gather some insights about the data reduction properties of the written data.

[10] The traces also contain a number of versions of each file system, but we use only a single snapshot from each file system.

6.2 Performance Evaluation

Ingest & Analyze Performance: We first evaluate the performance of the ingestion and volume level statistics calculation. These provide an idea on the time it takes to acquire sketch statistics at a volume level on large production systems. The timing of the ingestion phase is less critical and it is also hampered by the fact that the sketch data is read from disk in the manager system and passed to the sketch analyzer via a REST API. The performance of the analysis phase is more crucial, as it gives us an indication on our ability to process real-time queries and answer mass queries for optimization purposes.

In Table 1 we present timing results for four large scale tests. The times are affected both by the actual sketch size as well as by the number of volumes. In addition the hardware available for the sketch analyzer also has an effect. We ran the local tests on a virtual machine running on an Intel® Xeon® Gold 6130 CPU @ 2.10GHz (cpu cores: 4) with 4×16 GB DDR3 RAM and do not know the configuration of the external runs. But in general, the take home is that we can easily perform a cycle of sketch statistics once in a couple of minutes even for very large systems (small systems can run in seconds). In our field deployment the cycle is longer since the sketch retrieval process is throttled to reduce network overheads.

Group Query Performance: We turn to evaluate our ability to support online queries on reclaimable capacities of arbitrary groups. The latency of answering such queries for large groups is dominated by the merge operation which creates the sketch of the queried group (described in Section 5.2). Figure 3 plots the performance of the merge operation and the entire query response time (with reclaimable computation) for

random groups from the UBC-Dedup workload, on various group sizes. The graph depicts the average of 50 runs, with the largest skew being under 10 ms. We are able to satisfy such queries in less than 0.2 seconds even for a very large group which contains half of the volumes in the system. For a small group, e.g. of 12 volumes, the average test managed to run in under 4 ms.

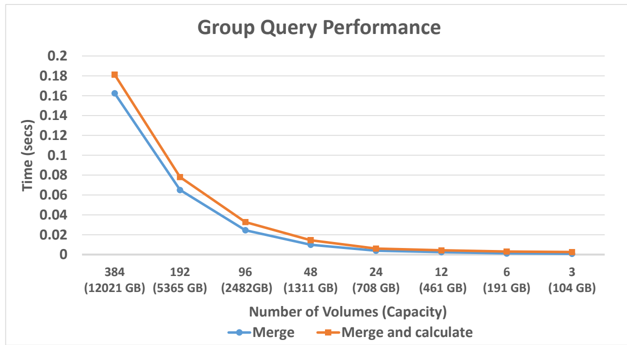


Figure 3: Performance of group query operations as a function of the group size ranging from a group of half the volumes in the system (368) through a small group of just 3 volumes.

Note that for further speedup we can run the merge operation using multiple threads, where each thread runs on separates partitions (according to hash values). For example, we tested a merge over all the 768 volumes on a virtual machine with four cores. The merge operation took 0.396 seconds using a single process, 0.186 seconds when using two processes and using 4 processes brought us down to 0.121 seconds.

6.3 Accuracy Evaluation

As mentioned in Section 6.1 we evaluated the accuracy of our work using two methods: using synthetic data and by studying the UBC-Dedup traces. The first is by writing synthetic data with expected behavior to the storage system and evaluating the expected reclaimable and attributed numbers. We complemented this by deleting volumes and measuring the amount of physical space reclaimed from the system. A crucial aspect of these tests was to evaluate the combination of compression and deduplication, since the UBC traces do not contain compression ratios. The synthetic tests therefore included writing data with a variety of deduplication and compression ratios. The skew observed in these test (not presented here) was always well within the accuracy guarantee.

The second method that we used to evaluate accuracy was using the UBC-Dedup traces. In order to evaluate the accuracy behavior of the sketch estimation method we first computed the exact physical capacities required to store each of the 768 volumes from the UBC-Dedup traces. This was done once by a lengthy offline process and recorded. We then evaluated the same physical capacities using our sketch mechanism with sketch factor 8192. In our evaluation we compare the observed error for each of the sketch estimations to the accuracy guarantee obtained in Theorem 1. For example, if the estimation is off by a factor of -3%, and the accuracy guarantee is $\varepsilon = 0.05$, the relative measured skew is $\frac{-0.03}{0.05} = -0.6$. In Figure 4 we show a histogram detailing the number of volumes in each range of relative measured skew of reclaimable estimations. The behavior comes out as a very nice bell curved distribution. Note that this behavior is not symmetric like a normal distribution, but rather is shifted to the negative skew, a behavior expected in a Binomial distribution $B(n, p)$ in which p is very small ($\frac{1}{8192}$ in our case).[11]

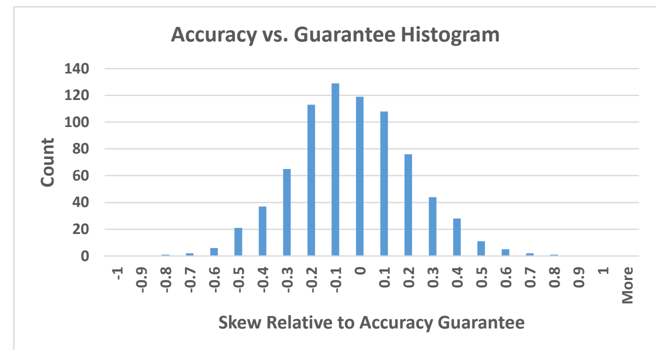


Figure 4: The errors observed over 768 volumes relative to the accuracy guarantee. The observed estimation skew was always smaller than the accuracy bound. In fact, in over 95% of the volumes the skew was less than half of the calculated accuracy bound.

To give a further indication on the behavior of sketch estimations, we picked six large volumes from the UBC-Dedup trace and examined the sketch estimation for with a growing sketch factor (starting with $2^{13} = 8192$ through 2^{17}). Figure 5 depicts the skew observed for each volume as the sketch factor grows. We observe that indeed the error tends to grows significantly as the sketch becomes sparser.

6.4 Data Center Level Optimizations

As an example of the potential of our methods for cross system optimizations, we implemented a greedy algorithm for space reclamation in a data center consisting of a number of deduplicated storage systems. The input to the algorithm is the name of the source system that is filling up and a minimal amount of space that needs to be reclaimed from it. The output is a migration plan of volumes from the source system to the other available system. The goal is to minimize the overall space usage of the entire data center by finding data similarities and exploiting them.

[11] This serves as justification for our choice not to use a binomial to normal estimation in the accuracy proof (as was done in [34]).

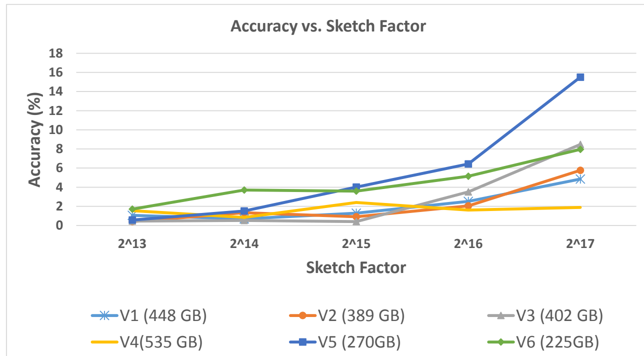


Figure 5: The errors observed for six volumes (and their physical sizes) as the sketch factor grows.

In a nutshell, at each round we enumerate over all of the volumes in the source system and evaluate what is their reclaimable space from the source system, and how much space they would take up in each of the other systems. At each round we pick the single volume for which the migration would yield the best space saving ratio[12] and update the sketches of the systems as though the migration has already happened. We then move to the next round in which the same process is repeated until the amount of reclaimed space from the source system is reached. Note that if data reduction mechanisms exist as part of the networking used for migrations (such as compression or WAN deduplication) then these consideration can easily be taken into account as part of the decisions in such a greedy algorithm.

The above algorithm does not attempt to find an overall optimum for such a process, and would generally not work well in situations in which the optimal solution involves moving several highly correlated volumes together. That being said, it exemplifies the insight and capabilities that the storage administrator has with clarity about expected volume capacities across multiple systems in the datacenter.

Evaluation: We evaluated the above algorithm using a simulated environment of four storage systems. The UBC-Dedup workload was partitioned among the four storage systems in a random fashion (each system received 192 volumes). On average, the physical space in each system amounted to 7TB.

We then ran the algorithm four times, each time with a different system serving as the source. In each test we asked to release at least 1TB of data from the source system. The tests ran between 30 to 55 seconds (depending on the system) and produced a migration plan that frees over 1TB of data from the source while taking up significantly less physical space in the other systems. The space savings achieved were between 257GB to 296GB, depending on the source system.

Figure 6 plots the progression of space reclamation and the capacity consumed at the targets for one of the experiments. We point out that as the rounds progress the ratio of space savings achieved by the migration process predictably declined. For example, at the beginning some volumes were found for which the space saving ratio from migration was as high as 10:1 and 3.7:1. As the algorithm progresses the extremely beneficial volumes have already been migrated and the saving ratio went down to around 1.15:1.

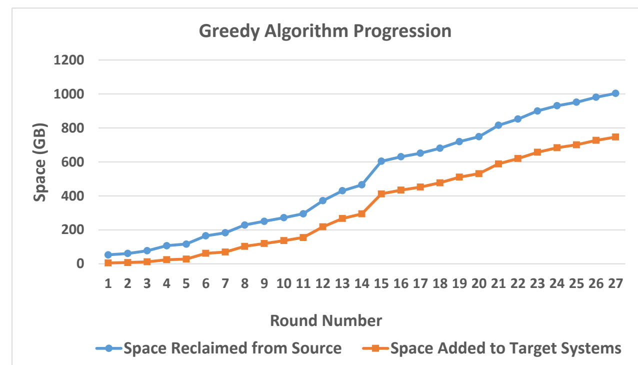


Figure 6: The progression of the reclaimed space from the source and the space the suggested volumes would take up in other systems.

6.5 Results from Early Adopters

As mentioned, our implementation is running as a beta offering for early adopters. This gives us initial statistics acquired in the field, on real customer data. We show here a glimpse of some insights that we have learnt (other than the performance numbers presented in Section 6.2): We evaluated how different the reclaimable numbers of a volume would be if they rely solely on unique data accounting rather than on sketches. The early numbers show that on average there is a 42% difference between the numbers and this can be attributed to the relative high internal volume deduplication encountered in the data sets that have been analyzed. This result strongly motivates the use of sketches for reclaimable capacity estimation.

Another example of insight that can be learnt is regarding the correlation between a data chunks' deduplicability and compression ratio. In the data sets that were scanned by the sketches mechanism we found no evidence of such correlation. Specifically, 99.9% of the data chunks had reference count between 1 and 4. For these four reference count values, we observed the exact same compression ratio of the data chunks.We are confident that as this feature is integrated and widely adopted we would gain some important insights on deduplication.

[12] We slightly penalize small volumes since we have a preference to migrate fewer volumes rather than many.

7 Conclusions and Discussion

We described a novel and efficient approach to analyzing volume capacities in storage systems with deduplication. Our mechanism provides accurate estimations for capacity measures that are not available in deduplicated storage systems to date. We have shown the accuracy of the capacity statistics computed from the sketch and demonstrated how it can be seamlessly collected from a system. From a performance standpoint our algorithms scale well and exhibit high performance even with high capacities. The small scale of the sketch and the ability to pull it out of the storage systems allows for further analytics and automation. To date, placement decision algorithm were mostly focused only on performance optimization and just making sure we don't overrun the system overall capacity. The sketch mechanism enables a new dimension of data center capacity optimization. This opens the door for performing insight analytics on storage capacities and making placement decisions at the pool or system level as well as across multiple deduplication domains and systems. Among the potential uses of our technology is the ability to reduce the overall space usage in a number of circumstances: Upon space reclamation when a system fills up (as described in Section 6.4); as part of data rebalancing between systems upon introducing of new systems; or by actively relocating volumes to reside in the same deduplication domain together with their optimal cluster of related volumes.

Acknowledgements: We are grateful to our many colleagues at the IBM Systems Israel Development Center who contributed to our efforts to bring this technology to the field.

References

[1] VDBench users guide. https://www.oracle.com/technetwork/server-storage/vdbench-1901683.pdf, 2012.

[2] HPE storeonce data protection backup appliances. https://www.hpe.com/us/en/storage/storeonce.html, 2018.

[3] IBM FlashSystem 9100. https://www.ibm.com/us-en/marketplace/flashsystem-9100, 2018.

[4] IBM FlashSystem A9000. https://www.ibm.com/il-en/marketplace/small-cloud-storage/specifications, 2018.

[5] Pure storage: purity-reduce. https://www.purestorage.com/products/purity/purity-reduce.html, 2018. (Retrieved Sept. 2018).

[6] SNIA: Iotta repository home. http://iotta.snia.org/, 2018.

[7] VMware vsan: Using deduplication and compression. https://docs.vmware.com/en/VMware-vSphere/, 2018.

[8] XIOS 6.1 data reduction (drr) reporting per a volume. https://xtremio.me/, 2018.

[9] XtremIO integrated data reduction. https://www.emc.com/collateral/solution-overview/h12453-xtremio-integrated-data-reduction-so.pdf, 2018. (Retrieved Sept. 2018).

[10] Aronovich, L., Asher, R., Bachmat, E., Bitner, H., Hirsch, M., and Klein, S. T. The design of a similarity based deduplication system. In *SYSTOR* (2009), ACM.

[11] Bar-Yossef, Z., Jayram, T. S., Kumar, R., Sivakumar, D., and Trevisan, L. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques, 6th International Workshop, RANDOM 2002* (2002), pp. 1–10.

[12] Bhagwat, D., Eshghi, K., Long, D. D. E., and Lillibridge, M. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *MASCOTS* (2009), pp. 1–9.

[13] Deutsch, P., and Gailly, J. L. Zlib compressed data format specification version 3.3. Tech. Rep. RFC 1950, Network Working Group, May 1996.

[14] Dong, W., Douglis, F., Li, K., Patterson, R. H., Reddy, S., and Shilane, P. Tradeoffs in scalable data routing for deduplication clusters. In *FAST* (2011), pp. 15–29.

[15] Douglis, F., Bhardwaj, D., Qian, H., and Shilane, P. Content-aware load balancing for distributed backup. In *Proceedings of the 25th Large Installation System Administration Conference, LISA* (2011).

[16] Flajolet, P., and Martin, G. N. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci. 31*, 2 (1985), 182–209.

[17] Forman, G., Eshghi, K., and Suermondt, J. Efficient detection of large-scale redundancy in enterprise file systems. *Operating Systems Review 43*, 1 (2009), 84–91.

[18] Frey, D., Kermarrec, A., and Kloudas, K. Probabilistic deduplication for cluster-based storage systems. In *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012* (2012), p. 17.

[19] Fu, Y., Jiang, H., and Xiao, N. A scalable inline cluster deduplication framework for big data protection.

In *Middleware 2012 - ACM/IFIP/USENIX 13th International Middleware Conference, Montreal, QC, Canada, December 3-7, 2012. Proceedings* (2012), pp. 354–373.

[20] GIBBONS, P. B., AND TIRTHAPURA, S. Estimating simple functions on the union of data streams. In *SPAA* (2001), pp. 281–291.

[21] GREENE, W. k-way merging and k-ary sorts. In *Proceedings of the 31-st Annual ACM Southeast Conference* (1993), pp. 127–135.

[22] HARNIK, D., KAT, R., SOTNIKOV, D., TRAEGER, A., AND MARGALIT, O. To zip or not to zip: Effective resource usage for real-time compression. In *USENIX FAST'13* (2013).

[23] HARNIK, D., KHAITZIN, E., AND SOTNIKOV, D. Estimating Unseen Deduplication—from Theory to Practice. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016), pp. 277–290.

[24] HARNIK, D., MARGALIT, O., NAOR, D., SOTNIKOV, D., AND VERNIK, G. Estimation of deduplication ratios in large data sets. In *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012* (2012), pp. 1–11.

[25] HUFFMAN, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers 40*, 9 (September 1952), 1098–1101.

[26] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)* (2009).

[27] LU, M., CONSTANTINESCU, C., AND SARKAR, P. Content sharing graphs for deduplication-enabled storage systems. *Algorithms 5*, 2 (2012).

[28] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)* (2011), pp. 1–13.

[29] MOTWANI, R., AND RAGHAVAN, P. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[30] NAGESH, P. C., AND KATHPAL, A. Rangoli: Space management in deduplication environments. In *Proceedings of the 6th International Systems and Storage Conference* (2013), SYSTOR '13, pp. 14:1–14:6.

[31] SHILANE, P., CHITLOOR, R., AND JONNALA, U. K. 99 deduplication problems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2016, Denver, CO, USA, June 20-21, 2016.* (2016).

[32] WALDSPURGER, C. A., PARK, N., GARTHWAITE, A., AND AHMAD, I. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, 2015), USENIX Association, pp. 95–110.

[33] WIRES, J., GANESAN, P., AND WARFIELD, A. Sketches of space: ownership accounting for shared storage. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017* (2017), pp. 535–547.

[34] XIE, F., CONDICT, M., AND SHETE, S. Estimating duplication by content-based sampling. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (2013), USENIX ATC'13, pp. 181–186.

[35] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (1977), 337–343.

Finesse: Fine-Grained Feature Locality based Fast Resemblance Detection for Post-Deduplication Delta Compression

Yucheng Zhang[†], *Wen Xia*[‡,⋆], *Dan Feng*[§,⋆], *Hong Jiang*[¶], *Yu Hua*[§], *Qiang Wang*[§]

[†]*Hubei University of Technology* [‡]*Harbin Institute of Technology, Shenzhen & Peng Cheng Laboratory*
[§]*WNLO, School of Computer, Huazhong University of Science and Technology* [¶]*University of Texas at Arlington*
[⋆]*Corresponding authors: xiawen@hit.edu.cn and dfeng@hust.edu.cn*

Abstract

In storage systems, delta compression is often used as a complementary data reduction technique for data deduplication because it is able to eliminate redundancy among the non-duplicate but highly similar chunks. Currently, what we call *'N-transform Super-Feature' (N-transform SF)* is the most popular and widely used approach to computing data similarity for detecting delta compression candidates. But our observations suggest that the *N-transform SF* is compute-intensive: it needs to *linearly transform each Rabin fingerprint of the data chunks N times to obtain N features*, and can be simplified by exploiting the fine-grained feature locality existing among highly similar chunks to *eliminate time-consuming linear transformations*. Therefore, we propose *Finesse*, a fine-grained feature-locality-based fast resemblance detection approach that divides each chunk into several fixed-sized subchunks, computes features from these subchunks individually, and then groups the features into super-features. Experimental results show that, compared with the state-of-the-art *N-transform SF* approach, *Finesse* accelerates the similarity computation for resemblance detection by 3.2×∼3.5× and increases the final throughput of a deduplicated and delta compressed prototype system by 41%∼85%, while achieving comparable compression ratios.

1 Introduction

Data deduplication, a popular data reduction technique, usually identifies duplicate data at the chunk level (e.g., 8KB size) by using secure fingerprints (e.g., SHA1) to uniquely and globally represent data chunks in storage systems [34, 44]. Hence, deduplication-based storage systems only store one physical instance referred to by any other duplicates, which helps improve storage space efficiency [18, 25, 44] or network bandwidth efficiency [26, 29].

Recently, delta compression has also gained increasing attention due to its ability to eliminate data redundancy among non-duplicate but highly similar chunks, which can be used

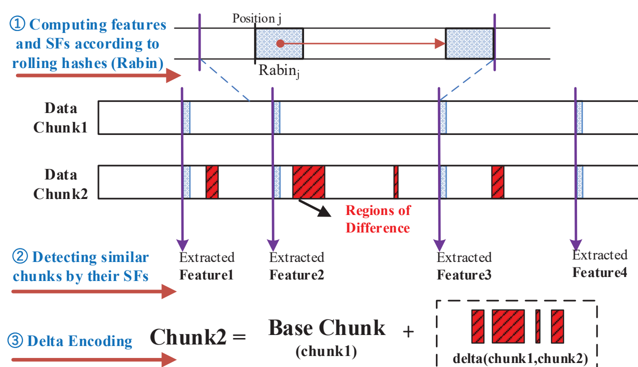


Figure 1: An example of delta compression on two similar chunks with the three typical steps: ① computing similarity, ② indexing, and ③ delta encoding.

post-deduplication as a complementary technique to further eliminate redundancy. For example, if chunk A_2 is similar to chunk A_1 (the base chunk), the delta compression approach only stores or transfers the differences (*delta*) and the mapping relation between A_2 and A_1, removing the redundant data to improve storage space efficiency [21, 30, 36, 37, 41] or network bandwidth efficiency [8, 29, 42, 43]. Several studies [29, 30, 37, 38] suggest that delta compression is able to achieve about 2× additional compression ratio beyond deduplication and local compression in backup storage workloads.

For delta compression in deduplication-based storage systems, resemblance detection is the first key step in its workflow, which identifies delta compression candidates. This is because a higher similarity degree in the detected chunks implies more space savings from delta compression. Currently, the most commonly used chunk-level resemblance detection approach computes the 'super-features' (***SF*** for short) [5, 17, 29] based on the Rabin fingerprints [28] of data contents, to detect highly similar chunks. Figure 1 gives an example of the general workflow for this *SF*-based delta compression approach: ① computing the similarity of chunks, namely, computing features and grouping features

into *SFs* (detailed in Section 3), ② detecting similar chunks according to their *SFs* (any two chunks having a *SF* in common are considered highly similar [6]), ③ delta encoding the two similar chunks, i.e., calculating their differences, also called 'delta'. For decompression, the input chunk is recovered by decoding the 'delta' with the base chunk.

To achieve high delta compression efficiency, some recent works on delta compression [17, 19, 29] recommend grouping four or more features into one *SF* to reduce false positives in resemblance detection, and using three or more *SFs* to detect more highly similar chunks for delta compression. But according to observations in our delta compressed prototype system, computing the similarity of data chunks, namely, generating their *SFs*, is quite time-consuming. Specifically, to ensure high similarity detection efficiency, Rabin fingerprints are calculated byte-by-byte on data chunks (similar to Content-Defined Chunking [26, 40, 43]), and are each then linearly transformed *N* times to calculate *N*-dimensional hash value sets. Finally the *N* maximal values, one from each of the N dimensions, are selected as features. Thus, the traditional *SF* approach needs to linearly transform each Rabin fingerprint of data chunks *N* times, which we refer to as ***'N-transform SF'*** to distinguish it from our approach in the remainder of the paper.

Consistent with the backup stream locality observed by many studies on deduplication [13, 16, 18, 22, 33, 35, 44], we observe that there also exists fine-grained locality among similar chunks. This locality refers to the fact that the corresponding subregions (subchunks) of chunks and their features also appear in the same order among the similar chunks with a very high probability, which is referred to as *feature locality* in this paper. Based on this key observation, we argue that a collection of features, exactly one extracted from each subchunk of a chunk, can also be used for representing the similarity of a chunk for generating *SFs*, which is much less compute-intensive than the *N-transform SF* since it eliminates the time-consuming linear transformations.

In this paper, we propose *Finesse*, a fast resemblance-detection approach that exploits the fine-grained feature locality of similar chunks. Specifically, *Finesse* simplifies computing the similarity by first dividing each chunk into several subchunks and then quickly computing features from each subchunk, finally grouping these features into *SFs*. Experimental results based on six datasets show that, compared with the baseline *N-transform SF* approach, *Finesse* accelerates the similarity computation by 3.2×∼3.5× and increases the throughput of a delta compression prototype system by 41%∼85%, while achieving comparable and even higher compression ratios.

2 Background and Related Work

Data reduction has gained increasing attention and popularity in storage and file-transfer systems due to the explosive growth of digital data. Compared with local compression (e.g., LZ [34]), data deduplication is able to identify and eliminate redundancy globally at at a much larger granularity (i.e., chunk- or file-level) in large-scale storage systems. Thus it is widely studied and used in large-scale backup storage [18, 29, 33, 44], primary storage [10, 24, 31], and HPC storage [23].

Meanwhile, delta compression, another data reduction technique that removes redundancy among non-duplicate but highly similar chunks, is able to help maximize the compression ratio when combined with deduplication and local compression in backup storage [30], storage replication [29], database storage [41], etc. Shilane et al. [29, 30] suggest that delta compression can achieve an additional 2× compression ratio beyond data deduplication in their production backup storage systems. Similar results are also observed in other scenarios, such as, database storage [41, 42] and migratory compression [19].

While greatly improving storage efficiency, delta compression also introduces extra compute and I/O overheads. SIDC [29] suggests that the issue of on-disk large-sized similarity indexing faced by delta compression can be addressed by exploiting (caching) backup stream locality, in a way similar to data deduplication systems [44]. Ddelta [38] and Edelta [39] have been proposed to accelerate the delta encoding process by using the idea of CDC-based deduplication and exploiting fine-grained locality of the backup data streams.

3 Super-Feature based Approach

Resemblance detection is the first step needed for delta compression to compute the similarity of data chunks and find compression candidates. As mentioned earlier, the ***'N-transform SF'*** approach is currently the most popular method for chunk-level resemblance detection. It was first proposed by Broder [6] and is based on "Broder's theorem" [5], which evaluates the resemblance between two sets, as detailed below:

Theorem 1 *Consider two sets A and B, with H(A) and H(B) being the corresponding sets of the hashes of the elements of A and B respectively, where H is chosen uniformly and randomly from a min-wise independent family of permutations [2, 7]. An element in the set is mapped to an integer. Let min(S) denote the smallest element of the set of integers S. Then:*

$$\Pr[\min(H(A))) = \min(H(B))] = \frac{|A \cap B|}{|A \cup B|}.$$

Broder's theorem states that the probability of the two sets A and B having the same minimum hash element is the same as their Jaccard similarity coefficient [14]. Based on this theorem, Broder proposed a resemblance detection approach called super-features [6, 29] that extracts a fixed number of features from a chunk. Specifically, this *SF*-based approach [29] (referred to as *N-transform SF* in this paper)

Algorithm 1 Extracting features in *N-transform SF*.

Require: chunk content, Str; length of the chunk, L; randomly value pair for linear transformation, m_i and a_i;
Ensure: N features, Feature[N];

```
1: function FEATURE-EXTRACT-N-TRANSFORM_SF(Str, L)
2:     Feature[0,···,N−1] ← 0;
3:     for m = 0 to L−1 do
4:         FP ← RabinFunction(Str, m);
5:         for i = 0 to N−1 do
6:             Transform[i] ← (m_i*FP + a_i) mod 2^32;
7:             if Feature[i] ≤ Transform[i] then
8:                 Feature[i] ← Transform[i];
9:             end if
10:        end for
11:    end for
12: end function
```

computes data similarity by extracting features from Rabin fingerprints (a rolling hash algorithm [28]) and then grouping these features into *SFs* to detect resemblance for data reduction. For example, $\mathrm{Feature_i}$ of a chunk (length = L), is uniquely generated with a randomly pre-defined value pair m_i & a_i (i.e., linear transformation) and L Rabin fingerprints (as used in Content-Defined Chunking [26, 40, 43] with a sliding window size of 48 bytes as follows:

$$\mathrm{Feature_i} = \mathrm{Max}_{j=1}^{L}\{(m_i \cdot \mathrm{Rabin_j} + a_i) \bmod 2^{32}\} \quad (1)$$

Where $\mathrm{Rabin_j}$ is the Rabin fingerprint of the sliding window located at position j) Thus chunks that have one or more such features (maximal values) in common are likely to be very similar, but small changes to the data are unlikely to perturb the maximal values [5, 29]. Algorithm 1 provides a detailed pseudo-code implementation of extracting features by *N-transform SF*. Then a super-feature of this chunk, $\mathrm{SF_x}$, can be calculated by several such features as follows:

$$\mathrm{SF_x} = \mathrm{Rabin}(\mathrm{Feature}_{x\cdot k}, ..., \mathrm{Feature}_{x\cdot k+k-1}) \quad (2)$$

For example, to generate three *SFs* with k=4 features each, we must first generate N=12 features, namely, features 0...3 for $\mathrm{SF_0}$, features 4...7 for $\mathrm{SF_1}$, etc. For similar chunks that differ only in a tiny fraction of bytes, most of their features will be identical and thus so are their *SFs* [6]. More specifically, this *N-transform SF* approach is able to maximally detect the highly similar chunks for two reasons. ① The matching of one *SF* means that almost all the features grouped in this *SF* are identical and thus grouping features into *SFs* reduces false positives for resemblance detection. ② Multiple SFs are computed to increase the probability of detecting highly similar chunks. Meanwhile, this *N-transform SF* approach needs to linearly transform Rabin fingerprints of the data chunks *N* times, which is time-consuming and slows the whole post-deduplication delta compression process.

It is worth noting that there are also some other *coarse-grained resemblance detection* approaches [4, 9, 11, 15, 27, 41] for matching similar files or large data blocks (e.g., size of 16MB), which extract features from non-overlapped strings (or chunks) and thus may suffer from high false positives. In this paper, we focus on improving the most popular *N-transform SF* approach for the chunk-level resemblance detection in post-deduplication delta compression scenario [30].

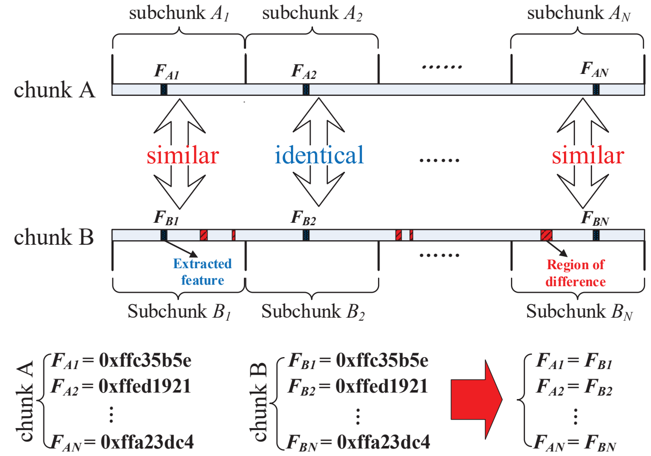


Figure 2: An example of the existence of fine-grained locality among two similar chunks. Here each chunk is divided into *N* fixed-sized subchunks. The corresponding subchunks in chunk B are largely similar (one-by-one) to subchunks in chunk A, and thus their features are largely identical.

4 *Finesse* Design and Implementation

4.1 Observations

As analyzed above, the root cause of the relatively high computation overhead of the *N-transform SF* approach is its linearly transforming the whole chunk's Rabin fingerprints *N* times (i.e., compute multiple rounds of transformations on each fingerprint) to extract *N* features. According to our observation of delta compression on backup workloads, the features extracted from the subchunks inside individual chunks can also be used for resemblance detection, which means that we eliminate the linear transformations and thus simplify the feature computation.

Computing features from subchunks is motivated by our observation that the fine-grained stream locality widely exists in the detected similar chunks. Figure 2 provides an example of this locality: the subregions (subchunks) of individual chunks also appear in the same order among their highly similar chunks with a very high probability, meaning that these subchunks are also very similar to each other.

Table 1 studies this locality on six deduplicated backup datasets (the detailed experimental environment and workload characteristics can be found in Section 5), which demonstrates that most of the corresponding subchunk pairs

Table 1: A study of the repeatability of subchunks and their features (i.e., the fine-grained locality) in the identified similar chunks in six deduplicated backup datasets. Here the identified chunks are all divided into 12 equal-sized subchunks and then we verify the locality shown in Figure 2.

Datasets	WEB	TAR	RDB	SYN	VMA	VMB
Avg. # of subchunks (identical)	8.27	9.19	6.86	5.78	5.99	6.34
Avg. # of subchunks (own the same features)	10.82	10.97	10.23	10.10	10.04	10.64

Here *identical* is judged by checking SHA-1 fingerprints of subchunks.

Algorithm 2 Extracting features in *Finesse*.

Require: chunk content, Str; length of the chunk, L;
Ensure: N features, Feature[N];

```
1:  function FEATURE-EXTRACT-FINESSE(Str, L)
2:      subChunkSize ← L/N;
3:      Feature[0, ··· , N−1] ← 0;
4:      for m = 0 to N−1 do
5:          for i = 0 to subChunkSize−1 do
6:              FP ← RabinFunction(Str, m*subChunkSize+i);
7:              if Feature[m] ≤ FP then
8:                  Feature[m] ← FP;
9:              end if
10:         end for
11:     end for
12: end function
```

in the detected similar chunks have the same features, accounting for 87.22% on average, although many of them are non-duplicate, accounting for 41.07% on average. Therefore, grouping some of these features into *SFs* by exploiting this fine-grained locality of similar chunks may also potentially enable maximal detection of highly similar chunks.

More importantly, this fine-grained locality-based resemblance detection approach has the potential to greatly reduce the execution time of computing features while achieving comparable resemblance detection efficiency relative to the *N-transform SF* approach, which is comprehensively evaluated and demonstrated in in Section 5.

4.2 Implementation

In this subsection, we discuss some implementation issues of *Finesse*, including the feature extraction and grouping strategies, overhead analysis, and other design considerations.

Feature Extraction. To exploit the fine-grained feature locality of similar chunks for extracting features, *Finesse* first divides a chunk into several fixed-sized subchunks, and then computes features on each subchunk based on the Rabin fingerprints of the data contents, in the same way as the traditional *N-transform SF* approach, which is detailed in Algorithm 2. Note that a chunk can be divided into variable-sized subchunks, similar to Content-Defined Chunking [26, 40, 43], but the feature grouping process will become very complicated since the subchunks' sizes and the number of features will become unknown. In addition, our preliminary results suggest that extracting features on fixed-sized subchunks is able to achieve nearly the same compression ratio as *N-transform SF*, and thus we use the fixed-sized subchunks for feature extraction.

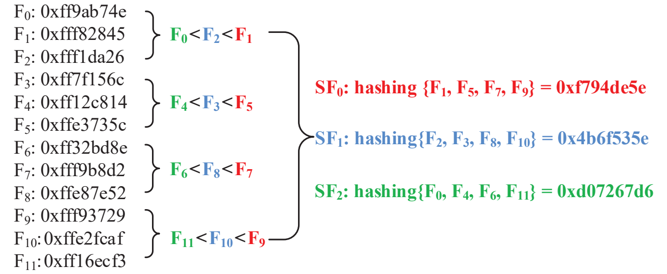


Figure 3: A concrete example of the grouping strategy in *Finesse* with actual values for the features and *SFs*.

Feature Grouping. The grouping strategy in *Finesse* is different from traditional *N-transform SF* since the way features are extracted is changed in *Finesse*. Specifically, *Finesse* first divides the subchunks and their corresponding features into several contiguous sets of the same size. Then the biggest features (with the largest hash values from each of the sets) are grouped to constitute the first *SF*, the second-biggest features of the sets are grouped to form the second *SF*, and so on and so forth. This grouping strategy in *Finesse* ensures that the grouped features in each *SF* are selected uniformly and consistently all over the chunks, which achieves grouping efficiency similar to *N-transform SF*.

To better illustrate this grouping strategy, we provide a detailed example with three *SFs* and four features per *SF* in *Finesse* as shown in Figure 3. We first divide the chunk into twelve subchunks to extract 12 features $F_0...F_{11}$. These features are further divided into four sets and sorted by their values, $\{F_0<F_2<F_1\}...\{F_{11}<F_{10}<F_9\}$. Finally, SF_0 is composed of the maximal values from the above four sets, namely, $\{F_1, F_5, F_7, F_9\}$, SF_2 of the 2nd biggest values $\{F_2, F_3, F_8, F_{10}\}$, and SF_2 of the 3rd biggest values $\{F_0, F_4, F_6, F_{11}\}$. Therefore, compared with feature extraction, feature grouping is fast since it only processes a small amount of features instead of the whole data chunk.

Note that we tried other grouping strategies for *Finesse* but the performance differences were small or ever worse. And our final evaluation result suggests *Finesse* using this grouping strategy achieves nearly the same delta-compression ratio as the classic *N-transform SF*.

Computational Overhead. As discussed above, the computational overhead of grouping features in *Finesse* is insignificant compared with computing features. Thus, we only analyze the computational overhead on computing features. Specifically, to generate *N* features from one chunk, for each Rabin fingerprint on the data chunk contents:

- *N-transform SF* needs at least $3 \times N$ operations, includ-

Table 2: Workload characteristics of the tested datasets.

Name	Size	DR	Workload descriptions
WEB	367 GB	4.21	135 days' snapshots of the website: *news.sina.com*.
TAR	112 GB	1.70	258 versions of Linux kernel source code [1]. Each version is packaged as a tar file.
RDB	540 GB	12.25	100 backups of the redis key-value store database.
SYN	330 GB	13.07	176 synthetic backups by simulating file create/delete/modify operations [32].
VMA	117 GB	1.61	78 virtual machine images of different OS release versions, including Fedora, CentOS, Debian, etc [3].
VMB	321 GB	10.45	20 backups of an Ubuntu 12.04 VM image in use by a research group.

Deduplication Ratio (DR) is measured by $\frac{\text{total data size before deduplication}}{\text{total data size after deduplication}}$.

ing N multiply, N add, and N conditional branch operations, to select N maximal values (i.e., features) after linear transformation as discussed in Section 3.

- *Finesse* only needs one operation, i.e., one 'conditional branch', to select one maximal value (one feature) in each subchunk.

Therefore, *Finesse* greatly reduces the computation overhead for feature extraction and thus accelerates the whole resemblance-detection process.

Limitations. Note that *Finesse* has one limitation in that it does not detect "similar" chunks with very different sizes. This is because *Finesse* divides a chunk into several equal-sized subchunks and the features will be totally different if the two "similar" chunks are of very different sizes. But in the delta-compression scenario, detecting chunks with similar sizes is reasonable since "similar" chunks with very different sizes (detected by the *N-transform SF approach*) may result in a low delta-compression ratio [17]. For example, two non-similar chunks that only have a small region in common may have many features and *SFs* in common and thus be considered to be similar chunks by the *N-transform SF approach*.

5 Performance Evaluation

5.1 Evaluation Setup

Experimental Platform. We implement delta compression in an open-source deduplication prototype system called Destor [12, 13] on the Ubuntu 12.04.2 operating system running on a quad-core Intel i7-4770 processor at 3.4 GHz and two 1TB 7200RPM hard disks. Another Intel E5-2620 processor at 2.4 GHz is also used for performance comparison.

Data Reduction Configurations. In our prototype system, deduplication is configured with Rabin-based chunking with the expected chunk size of 8KB as used in LBFS [26] and an in-memory SHA1 fingerprint table for duplicate detection.

For the post-deduplication delta compression, the non-duplicate chunks are processed in three steps: *resemblance detection, base chunk reading, and delta encoding*. The *resemblance detection* step for both *Finesse* and *N-transform SF* is configured to compute 3 *SFs* and 4 features per *SF* for matching highly similar chunks as suggested by SIDC [29] and MC [19] (to trade off the space savings and the computation & indexing overheads). In addition, a chunk may have multiple similar chunks, and our prototype system selects the first matched chunk as its base, which is also known as "FirstFit" [17]. For the *base chunk reading* step, delta compression needs to read for each matched similar chunk its base chunk from the disk for delta encoding. Here we use a base chunk cache with LRU and a size of 400MB to reduce base chunk I/Os. For *delta encoding*, we employ the classic Xdelta [20] to calculate the delta of the similar chunks for space saving.

Performance Metrics. We evaluate resemblance detection performance of *Finesse* using two metrics, *Delta Compression Ratio* (DCR) and *Delta Compression Efficiency* (DCE). DCR is measured by $\frac{\text{total size before delta compression}}{\text{total size after delta compression}}$, reflecting the total space saved by resemblance detection and then delta compression. DCE is used to estimate the similarity degree between the similar chunks detected by *Finesse*, i.e., $\frac{\text{the chunk data size after delta compression}}{\text{the chunk data size before delta compression}}$. It is worth noting that DCR focuses on the overall space savings while DCE emphasizes the detected resembling chunks themselves. Thus higher DCE means lower probability of false positives for detecting similar chunks.

In addition, *Similarity Computing Speed* is measured by the processing speed at which the input data are calculated to obtain *SFs* for resemblance detection. *System Throughput* is measured by the throughput with which the input data are deduplicated and then delta compressed. We run each experiment five times to get the stable and the average results of the deduplication throughput.

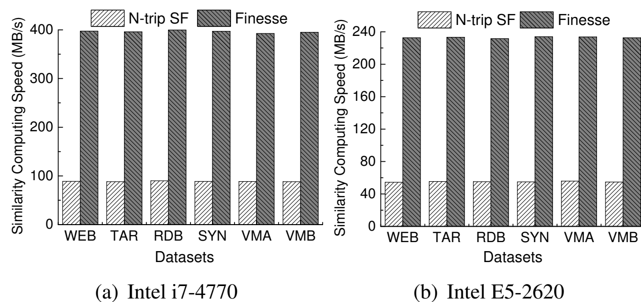
Evaluated Datasets. Six datasets are used for evaluation as shown in Table 2. These datasets represent various typical workloads, including website snapshots, tarred source code files, database snapshots, and virtual machine images.

5.2 Evaluation of *Finesse* vs. *N-transform SF*

Resemblance Detection Efficiency. Table 3 provides the delta compression results of all the similar chunks detected (i.e., they have a super-feature in common) by *Finesse* and *N-transform SF* respectively. Generally, evaluation results in Table 3 suggest that *Finesse* achieves comparable compression ratio (with difference of -3.21%∼+7.36%) to the N-transform SF approach in the metrics of DCR and DCE. In addition, the resemblance detection performance of *Finesse* is sensitive to the datasets due to the different ways in which the files of each workload are evolved (i.e., modified) during backups. Thus the six workloads have different levels of the fine-grained locality as studied in Table 1 (see Section 4.1). For example, *Finesse* achieves higher DCR on datasets TAR

Table 3: Comparison of resemblance detection efficiency of *N-transform SF* and *Finesse* on the six datasets.

Dataset	Approaches	DCR	DCE
WEB	*N-transform SF*	7.60	0.8749
	Finesse	7.52 (–1.05%)	0.8795 (+0.53%)
TAR	*N-transform SF*	15.00	0.9516
	Finesse	15.34 (+2.27%)	0.9846 (+3.47%)
RDB	*N-transform SF*	3.67	0.9129
	Finesse	3.94 (+7.36%)	0.9448 (+3.49%)
SYN	*N-transform SF*	1.75	0.9326
	Finesse	1.70 (–2.86%)	0.9640 (+3.37%)
VMA	*N-transform SF*	1.56	0.9088
	Finesse	1.51 (–3.21%)	0.9161 (+0.80%)
VMB	*N-transform SF*	1.30	0.9093
	Finesse	1.28 (–1.54%)	0.9193 (+1.10%)



(a) Intel i7-4770 (b) Intel E5-2620

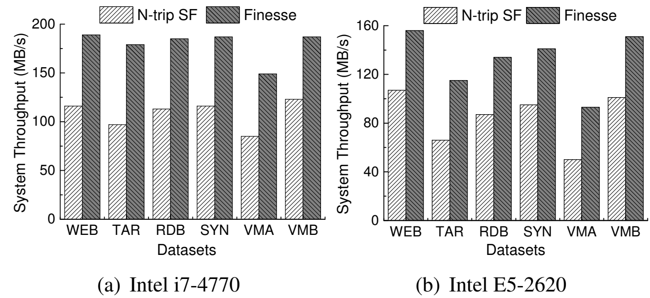
Figure 4: Similarity computing speed.

and RDB, and lower DCR on datasets SYN and VMB.

Meanwhile, *Finesse* achieves higher DCE than *N-transform SF* on all the six datasets. There are two reasons. ① The *N-transform SF* approach may obtain all the features from one subregion of the chunk, which can lead to possible false positive resemblance detection and thus lower DCE. In contrast, *Finesse*'s *SF* grouping strategy ensures that the features grouped for each *SF* are coming from multiple subchunks of a chunk. ② *N-transform SF* may detect "similar" chunks with the very different sizes, which can result in poor delta compression efficiency as discussed in Section 4.2.

Speed of Computing SFs. While *Finesse* achieves comparable compression ratios to that of *N-transform SF*, it greatly accelerates the similarity computation as shown in Figure 4. *Finesse* improves this speed by an average of 3.5× and 3.2× respectively on the i7-4770 and E5-2620 CPUs. This is because it requires much fewer operations on computing features as discussed in Section 4. Note that the SF computing speed is not sensitive to the datasets because the time on computing features is decided by the size of the data chunks (i.e., scan all the bytes to calculate features and SFs).

System Throughput. To understand the impact of the resemblance detection approaches on the total throughput of the composite data reduction system combining deduplication and delta compression, we construct and evaluate



(a) Intel i7-4770 (b) Intel E5-2620

Figure 5: Throughputs of the *Finesse* based and *N-transform SF* based delta compression prototype systems.

the throughputs of two such systems with *Finesse* and *N-transform SF* as their delta compression components respectively. In our prototype system of both *Finesse* and *N-transform SF*, we pipeline the deduplication subtasks (i.e., chunking, fingerprinting, and indexing) and delta compression subtasks (i.e., resemblance detection, reading base chunk, and delta encoding) for high system throughput.

Figure 5 shows the evaluation results comparing these two systems. The system based on the *Finesse* approach outperforms the one based on *N-transform SF* by 41%-85% in total system throughput. This is because in the delta compression phase after deduplication, *Finesse* is running 3× faster than *N-transform SF* for resemblance detection.

6 Conclusion

In this paper, we propose *Finesse*, a much faster resemblance detection approach than the state-of-the-art *N-transform SF* approach. The key idea behind *Finesse* is to exploit the fine-grained feature locality of highly similar chunks by dividing data chunk into multiple subchunks and extract features from each subchunk, thus reducing the computation overhead of resemblance detection. Our experimental results based on six datasets demonstrate the superior performance of *Finesse* in terms of delta compression ratio, delta compression efficiency, speed of computing SFs, and throughput of the composite data reduction prototype system combining deduplication and delta compression.

Acknowledgments

We are grateful to our shepherd Geoff Kuenning and the anonymous reviewers for their insightful comments and feedback on this work. This research was partly supported by NSFC No.61821003, No.61502190, No.U1705261, No.61832007, No.61772222, No.61772212, No.61772180 and No.61672010; The Scientific Research Fund of Hubei Provincial Department of Education B2017042; US NSF under Grants CCF-1704504 and CCF-1629625.

References

[1] Linux archives. `https://www.kernel.org`.

[2] Minhash. `https://en.wikipedia.org/wiki/MinHash`.

[3] VMs archives. `http://www.thoughtpolice.co.uk`.

[4] ARONOVICH, L., ASHER, R., BACHMAT, E., BITNER, H., HIRSCH, M., AND KLEIN, S. T. The design of a similarity based deduplication system. In *the 2th Annual International Systems and Storage Conference (SYSTOR'09)* (Haifa, Israel, 2009), ACM Association, pp. 1–14.

[5] BRODER, A. Z. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)* (Washington, DC, USA, 1997), IEEE, pp. 21–29.

[6] BRODER, A. Z. Identifying and filtering near-duplicate documents. In *Combinatorial Pattern Matching* (Montreal, Canada, 2000), Springer, pp. 1–10.

[7] BRODER, A. Z., CHARIKAR, M., FRIEZE, A. M., AND MITZENMACHER, M. Min-wise independent permutations. *Journal of Computer and System Sciences 60*, 3 (2000), 630–659.

[8] CUI, Y., LAI, Z., WANG, X., DAI, N., AND MIAO, C. Quicksync: Improving synchronization efficiency for mobile cloud storage services. In *International Conference on Mobile Computing and NETWORKING (MobiCom'15)* (Paris, France, 2015), ACM Association, pp. 592–603.

[9] DOUGLIS, F., AND IYENGAR, A. Application-specific delta-encoding via resemblance detection. In *USENIX Annual Technical Conference, General Track* (San Antonio, TX, USA, 2003), USENIX Association, pp. 113–126.

[10] EL-SHIMI, A., KALACH, R., KUMAR, A., AND ET AL. Primary data deduplication-large scale study and system design. In *the 2012 conference on USENIX Annual Technical Conference* (Boston, MA, USA, 2012), USENIX Association, pp. 1–12.

[11] FORMAN, G., ESHGHI, K., AND CHIOCCHETTI, S. Finding similar files in large document repositories. In *the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'05)* (Chicago, Illinois, USA, 2005), ACM Association, pp. 394–400.

[12] FU, M. Destor: An experimental platform for chunk-level data deduplication. https://github.com/fomy/destor, 2014.

[13] FU, M., FENG, D., HUA, Y., HE, X., CHEN, Z., XIA, W., ZHANG, Y., AND TAN, Y. Design tradeoffs for data deduplication performance in backup workloads. In *the 13th USENIX Conference on File and Storage Technologies (FAST'15)* (Santa Clara, CA, USA, 2015), vol. 9, USENIX Association, pp. 331–345.

[14] JACCARD, P. Etude de la distribution florale dans une portion des alpes et du jura. *Bulletin De La Societe Vaudoise Des Sciences Naturelles 37*, 142 (1901), 547–579.

[15] JAIN, N., DAHLIN, M., AND TEWARI, R. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In *the USENIX Conference on File and Storage Technologies (FAST'05)* (San Francisco, CA, USA, 2005), USENIX Association, pp. 281–294.

[16] KAISER, J., MEISTER, D., AND BRINKMANN, A. Deriving and comparing deduplication techniques using a model-based classification. In *the 10th European Conference on Computer Systems (EuroSys'15)* (Bordeaux, France, 2015), ACM Association, pp. 1–13.

[17] KULKARNI, P., DOUGLIS, F., LAVOIE, J. D., AND TRACEY, J. M. Redundancy elimination within large collections of files. In *the 2004 USENIX Annual Technical Conference (ATC'04)* (Boston, MA, USA, 2004), USENIX Association, pp. 1–14.

[18] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., AND ET AL. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *the 7th USENIX Conference on File and Storage Technologies (FAST'09)* (San Jose, CA, 2009), vol. 9, USENIX Association, pp. 111–123.

[19] LIN, X., LU, G., DOUGLIS, F., SHILANE, P., AND WALLACE, G. Migratory compression: Coarse-grained data reordering to improve compressibility. In *the 12th USENIX Conference on File and Storage Technologies (FAST'14)* (Santa Clara, CA, USA, 2014), USENIX Association, pp. 257–271.

[20] MACDONALD, J. *File system support for delta compression*. PhD thesis, Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[21] MEISTER, D., JÜRGEN, AND BRINKMANN. Multi-level comparison of data deduplication in a backup scenario. In *the 2th Annual International Systems and Storage Conference (SYSTOR'09)* (Haifa, Israel, 2009), ACM Association, pp. 1–12.

[22] MEISTER, D., KAISER, J., AND BRINKMANN, A. Block locality caching for data deduplication. In *the 6th International Systems and Storage Conference (Systor'13)* (Haifa, Israel, 2013), ACM Association, pp. 1–12.

[23] MEISTER, D., KAISER, J., BRINKMANN, A., AND ET AL. A Study on Data Deduplication in HPC Storage Systems. In *the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)* (Salt Lake City, Utah, USA, 2012), IEEE Computer Society Press, pp. 1–11.

[24] MEYER, D., AND BOLOSKY, W. A study of practical deduplication. In *the 9th USENIX Conference on File and Storage Technologies (FAST'11)* (San Jose, CA, USA, 2011), USENIX Association, pp. 229–241.

[25] MIN, J., YOON, D., AND WON, Y. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers 60*, 6 (2011), 824–840.

[26] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A Low-Bandwidth Network File System. In *the ACM Symposium on Operating Systems Principles (SOSP'01)* (Banff, Canada, 2001), ACM Association, pp. 1–14.

[27] PUCHA, H., ANDERSEN, D. G., AND KAMINSKY, M. Exploiting similarity for multi-source downloads using file handprints. In *the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI'07)* (Cambridge, MA, 2007), USENIX Association, pp. 15–28.

[28] RABIN, M. O. *Fingerprinting by Random Polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[29] SHILANE, P., HUANG, M., WALLACE, G., AND ET AL. WAN optimized replication of backup datasets using stream-informed delta compression. In *the 10th USENIX Conference on File and Storage Technologies (FAST'12)* (San Jose, CA, USA, 2012), USENIX Association, pp. 49–63.

[30] SHILANE, P., WALLACE, G., HUANG, M., AND HSU, W. Delta Compressed and Deduplicated Storage Using Stream-Informed Locality. In *the 4th USENIX conference on Hot Topics in Storage and File Systems (HotStorage'12)* (Boston, MA, USA, 2012), USENIX Association, pp. 201–214.

[31] TARASOV, V., JAIN, D., KUENNING, G., MANDAL, S., PALANISAMI, K., SHILANE, P., TREHAN, S., AND ZADOK, E. Dmdedup: Device mapper target for data deduplication. In *Ottawa Linux Symposium (OLS'14)* (2014), pp. 1–10.

[32] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating realistic datasets for deduplication analysis. In *Proceedings of the 2012 conference on USENIX Annual technical conference* (2012), USENIX Association, p. 24C34.

[33] WALLACE, G., DOUGLIS, F., QIAN, H., AND ET AL. Characteristics of backup workloads in production systems. In *the 10th USENIX Conference on File and Storage Technologies (FAST'12)* (San Jose, CA, 2012), USENIX Association, pp. 33–48.

[34] XIA, W., JIANG, H., FENG, D., DOUGLIS, F., SHILANE, P., HUA, Y., FU, M., ZHANG, Y., AND ZHOU, Y. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE 104*, 9 (2016), 1681–1710.

[35] XIA, W., JIANG, H., FENG, D., AND HUA, Y. SiLo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *the 2011 conference on USENIX Annual Technical Conference (ATC'11)* (Portland, OR, 2011), USENIX Association, pp. 285–298.

[36] XIA, W., JIANG, H., FENG, D., AND TIAN, L. Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets. In *Data Compression Conference (DCC), 2014* (2014), IEEE, pp. 203–212.

[37] XIA, W., JIANG, H., FENG, D., AND TIAN, L. DARE: A deduplication-aware resemblance detection and elimination scheme for data reduction with low overheads. *IEEE Transactions on Computers 65*, 6 (2016), 1692–1705.

[38] XIA, W., JIANG, H., FENG, D., TIAN, L., FU, M., AND ZHOU, Y. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation 79* (2014), 258–272.

[39] XIA, W., LI, C., JIANG, H., FENG, D., HUA, Y., QIN, L., AND ZHANG, Y. Edelta: A word-enlarging based fast delta compression approach. In *the 7th USENIX conference on Hot Topics in Storage and File Systems* (Santa Clara, CA, 2015), USENIX Association, pp. 1–5.

[40] XIA, W., ZHOU, Y., JIANG, H., FENG, D., HUA, Y., HU, Y., LIU, Q., AND ZHANG, Y. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 101–114.

[41] XU, L., PAVLO, A., SENGUPTA, S., AND GANGER, G. R. Online deduplication for databases. In *ACM International Conference on Management of Data (SIGMOD'17)* (Chicago, IL, USA, 2017), ACM Association, pp. 1355–1368.

[42] XU, L., PAVLO, A., SENGUPTA, S., LI, J., AND GANGER, G. R. Reducing replication bandwidth for distributed document databases. In *the 6th ACM Symposium on Cloud Computing (SoCC'15)* (Big Island, Hawaii, USA, 2015), ACM Association, pp. 222–235.

[43] ZHANG, Y., JIANG, H., FENG, D., XIA, W., FU, M., HUANG, F., AND ZHOU, Y. AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *Proceedings of the IEEE INFOCOM* (2015), IEEE, pp. 1337–1345.

[44] ZHU, B., LI, K., AND PATTERSON, R. H. Avoiding the disk bottleneck in the data domain deduplication file system. In *the 6th USENIX Conference on File and Storage Technologies (FAST'08)* (San Jose, CA, USA, 2008), vol. 8, USENIX Association, pp. 269–282.

Sliding Look-Back Window Assisted Data Chunk Rewriting for Improving Deduplication Restore Performance

Zhichao Cao[†] Shiyong Liu[‡] Fenggang Wu[†] Guohua Wang[§]
Bingzhe Li[†] David H.C. Du[†]

[†]University of Minnesota, Twin Cities [‡]Ocean University of China
[§]South China University of Technology

Abstract

Data deduplication is an effective way of improving storage space utilization. The data generated by deduplication is persistently stored in data chunks or data containers (a container consisting of a few hundreds or thousands of data chunks). The data restore process is rather slow due to data fragmentation and read amplification. To speed up the restore process, data chunk rewrite (a rewrite is to store a duplicate data chunk) schemes have been proposed to effectively improve data chunk locality and reduce the number of container reads for restoring the original data. However, rewrites will decrease the deduplication ratio since more storage space is used to store the duplicate data chunks.

To remedy this, we focus on reducing the data fragmentation and read amplification of container-based deduplication systems. We first propose a flexible container referenced count based rewrite scheme, which can make a better tradeoff between the deduplication ratio and the number of required container reads than that of capping which is an existing rewrite scheme. To further improve the rewrite candidate selection accuracy, we propose a sliding look-back window based design, which can make more accurate rewrite decisions by considering the caching effect, data chunk localities, and data chunk closeness in the current and future windows. According to our evaluation, our proposed approach can always achieve a higher restore performance than that of capping especially when the reduction of deduplication ratio is small.

1 Introduction

With the fast development of new eco-systems such as social media, cloud computing, artificial intelligence (AI), and Internet of Things (IoT), the volume of data created increases exponentially. However, the storage density and capacity increase in main storage devices like disk drives (HDD) and solid-state drives (SSD) cannot match the explosion in the speed of data creation [1, 2]. Data deduplication is an efficient way of improving the storage space utilization such that the cost of storing data can be reduced. Data deduplication is a widely used technique to reduce the amount of data to be transferred or stored in today's computing and communication infrastructure. It has been applied to primary and secondary storage systems, embedded systems, and other systems that host and transfer a huge amount of data.

Data deduplication partitions the *byte stream* formed by the original data into data chunks. The chunk size can be either fixed or variable (e.g., 4KB size in average). For efficiency, deduplicating systems usually process a *segment* of data at one time. A segment is typically the size of several thousand data chunks (e.g., 20MB). Each chunk is represented by a *chunk ID* which is a hashed fingerprint of the chunk content. By representing the original data stream with a sequence of the chunk IDs as well as the metadata to find the data chunks (called a *recipe*) and storing only the unique data chunks (not the duplicates of existing data chunks) into the storage system, the amount of data to be stored can be greatly reduced. The result of deduplication can be measured by the *deduplication ratio* which is the total amount of data in the original byte stream divided by the total size of stored unique data chunks.

Since the size of a data chunk is rather small, storing individual data chunks directly cannot efficiently utilize the storage bandwidth especially for storage systems with low random read/write performance. For HDD based storage systems used by backup or archive applications, the deduplication process typically accumulates a number of data chunks (say 1000 data chunks) in a *container* before writing them out together [3]. For the same reason, when reading a data chunk to restore the original data, the whole container is read from storage since we are expecting that many data chunks in the same container will be accessed soon. In this paper, we focus on the container-based deduplication systems.

The data deduplication restore process accesses and returns the data chunks based on the order in the recipe to recreate the original byte stream. Since the unique data chunks are writing to storage based on the order of their first appearance in the byte stream, a duplicate data chunk may require reading a container that was stored a long time ago. In this case, the restore process may not require most of the data chunks in such a container in the near future. This causes data fragmentation (i.e., data chunks are scattered) and read amplification (i.e., the size of data being read is larger than

the size of data being restored), which leads to low restore performance. Therefore, reducing the number of container reads is a major task for restore performance improvement.

To improve the restore performance, several techniques have been proposed including caching schemes (e.g., container-based caching [4, 5, 6, 7], chunk-based caching [8, 9, 10], and forward assembly [8, 10]) and duplicate data chunk rewrite schemes [4, 5, 6, 7, 8]. Since the unique data chunks are stored in the same order as they first appeared, unique data chunks in the same container are usually at nearby locations that these chunks are first identified in the original byte stream. If a large number of data chunks from the same container are used in a small range in the original byte stream, caching schemes can be very useful in reducing the number of container reads.

When a deduplication system runs for a long time, the data chunks in a segment of the original byte stream can be highly fragmented. That is, the duplicate data chunks in this segment are stored in a large number of containers that are new or have already existed (old containers). When restoring this segment, each container read from storage (read-in container) holds only a few data chunks that belong to this segment. Therefore, a large number of data chunks are read in from storage, possibly evicting cached data chunks before they are used again in the restore process. In this scenario, the number of container reads cannot be effectively reduced by applying caching schemes only.

In this situation, each container read only contributes a small number of data chunks for restore. This type of container read is inefficient and expensive. To address this issue, rewriting some of the duplicate data chunks in a highly fragmented segment and storing them together with the nearby newly identified unique data chunks in the same container can effectively reduce the number of container reads required for restoring these chunks. However, it is hard to decide which duplicate data chunks to rewrite during the deduplication process due to the limited information that can be used by a rewrite scheme.

Previous studies introduced several data chunk rewriting policies. Among these studies, Lillibridge *et al.* [8] proposed a simple but effective data chunk rewriting selection policy, named *capping*. Capping first partitions the byte stream into fixed size segments (e.g., 20MB). Then, in each segment, the referenced old containers are sorted in a descending order based on the number of data chunks in each container that appeared in the segment. Data chunks in the old containers which are ranked out of a pre-defined threshold (i.e., *capping level*) are stored again (to be *rewritten*). Therefore, when restoring the data chunks in this segment, the total number of container reads is limited by the capping level plus the number of new containers created when deduplicating this segment.

Capping operates on a single fixed-size segment of input data at a time. It applies the same cap to each segment and analyzes each segment in isolation, without considering the contents of the previous or following segments. Also, the deduplication ratio cannot be guaranteed via capping. In this paper, we explore two data chunk rewrite schemes to improve restore performance. First, based on capping, we propose a flexible container referenced count based scheme to adjust the cap for each segment according to the fragmentation of the duplicate data chunks in that segment. Second, we propose a new rewrite scheme called Sliding Look-Back Window Rewrite that makes better rewrite decisions by considering a larger amount of input data around each duplicate chunk. This avoids inefficiencies when related duplicate chunks are divided by the arbitrary boundaries between segments. The new rewrite policy for the sliding look-back window scheme combines the flexible container referenced count based scheme with the consideration of caching effects in the restore process.

To fairly and comprehensively evaluate our rewriting designs, we implemented a system with both deduplication and restore engines for normal deduplication, the capping scheme, the two schemes we are proposing, Forward Assembly (FAA) [8], and a chunk-based caching scheme called ALACC [10]. We compared and evaluated the performance of different combinations of deduplication and restore engines. We use *speed factor* (MB/container-read), which is defined as the mean size data being restored (MB) per container read [8], to indicate the amount of data that can be restored by one container read on average. Speed factor is platform independent and a higher speed factor usually indicates a higher restore performance. Using several real-world deduplication traces, with the same deduplication ratio and the same restore engine, our proposed sliding look-back window based design always achieves the best restore performance. Our design can improve the speed factor up to 97% compared with normal deduplication and it can improve the speed factor up to 41% compared with capping.

The rest of the paper is presented as follows. Section 2 reviews the background of data deduplication and the related work of caching and rewrite schemes. Section 3 describes a flexible container referenced count based rewrite scheme which improves on capping. To further reduce the number of container reads, a sliding look-back window based rewrite scheme is proposed and presented in Section 4. Based on the sliding look-back window, the rewrite candidate selecting policy is discussed in Section 5. We present the evaluation results and analysis in Section 6. Finally, we conclude our work and discuss future work in Section 7.

2 Background and Related Work

In this section, we first briefly describe the deduplication and restore processes. Then, the related studies of improving the restore performance are presented and discussed.

2.1 Deduplication and Restore Process

Data deduplication is widely used in secondary storage systems, such as archiving and backup systems, to improve storage space utilization [3, 11, 12, 13, 14, 15, 16, 17]. Data deduplication is also deployed in primary storage systems to make better trade-offs between cost and performance [18, 19, 20, 21, 22, 23, 24, 25]. After the original data is deduplicated, only the unique data chunks and the recipe are stored. When the original data is requested, the recipe is used to read the corresponding data chunks for assembling the data. From the beginning of the recipe, the restore engine uses the data chunk metadata to access the corresponding data chunks one by one and assembles the data chunks in the memory buffer (assembling buffer). Once the engine accumulates a buffer worth of data chunks, it returns the restored data.

To store the unique data chunks, some deduplication systems such as HYDRAstor [17], iDedup [18], Dmdedup [22], and ZFS [26] directly store individual data chunks to the persistent storage. They do not incur read amplification during restore, but they suffer from data chunk fragmentation and the slow performance of random reads. Other deduplication systems, such as the backup products from Veritas [27] and Data Domain [3], pack a number of data chunks (compression may be applied) in one I/O unit (called a container in this paper). Data chunks in the same container are written out and read in together to benefit from the high sequential I/O performance and good chunk localities. In this paper, we focus on the container-based deduplication systems.

In the worst case of restore, we may need N container reads to assemble N data chunks. A straightforward way to reduce the number of container reads is to cache some of the containers or data chunks. Since some data chunks will be used soon after they are read into memory, these cached chunks can be directly copied from the cache to the assembling buffer which can reduce the number of container reads. In some scenarios, even with caching schemes like chunk-based caching and forward assembly [8], the number of container reads cannot be further reduced. This is especially true for restoring a duplicate data chunk since this data chunk was stored in a container created earlier and most of the data chunks in this container may not be needed by the restore process.

Another way to reduce the number of container reads is to store (rewrite) some of the duplicate data chunks together with the unique chunks in the same container during the deduplication process. The decision to rewrite a duplicate data chunk has to be made during the deduplication process instead of being done at restore process like caching. After rewriting, the duplicate chunks and unique chunks will be read together from the same container, thus avoiding the need to read these duplicate chunks from other old containers. However, this approach reduces the effectiveness of data deduplication (i.e., reduces the deduplication ratio). Rewrite can effectively reduce the number of container reads in the cases that caching schemes do not work well, especially when each container read only contributes a few data chunks to restoring the original data.

2.2 Related Work on Restore Performance Improvement

We will first review the different caching policies such as container-based caching, chunk-based caching, and forward assembly. Then, we will focus on the studies of storing duplicate chunks to improve restore performance.

Different caching policies are studied in [4, 5, 6, 8, 9, 10, 28]. To improve the cache hit ratio, Kaczmarczyk *et al.* [4], Nam *et al.* [5, 6], and Park *et al.* [28] used container-based caching, which cache containers in memory. Container-based caching schemes can achieve Belady's optimal replacement policy [7]. To achieve a higher cache hit ratio, some studies cache data chunks directly [8, 9]. If we compare container-based with chunk-based caching, the former has lower cache management overhead (e.g., fewer memory copies), while the latter has a higher cache hit ratio. Therefore, caching chunks is preferred in the restore process, especially when the cache space is limited. Lillibridge *et al.* proposed the forward assembly scheme, which reads ahead in the recipe and pre-fetches some chunks into a Forward Assembly Area (FAA) [8]. This scheme ensures that no container will be read more than once when restoring the data chunks of the current FAA. The management overhead of FAA is lower than that of chunk-based caching, but the restore performance of these two schemes is closely related to the workload characteristics. Therefore, we previously proposed a new caching scheme which combines FAA and chunk-based caching called Adaptive Look-Ahead Chunk Caching (ALACC). ALACC can potentially adapt to various workloads and achieve better restore performance [10].

Nam *et al.* [5, 6] first introduced the Chunk Fragmentation Level (CFL) to estimate degraded read performance of deduplication storage. CFL is defined as a ratio of the optimal number of containers with respect to the actual number of containers required to store all the unique and duplicate chunks of a backup data stream. When the current CFL becomes worse than a predefined threshold, data chunks will be rewritten. Kaczmarczyk *et al.* [4, 29] utilized stream context and disk context of a duplicate block to rewrite highly-fragmented duplicates. A data chunk whose stream context in the current backup is significantly different from its disk context will be rewritten. Fu *et al.* [7, 30] proposed a History-Aware Rewriting algorithm (HAR) which identifies and rewrites sparse containers according to the historical information of the previous backup. A new rewrite scheme called container capping was proposed by Lillibridge *et al.* [8], which uses a fixed-size segment to identify the data chunks to be rewritten. Since each container read involves a

large fixed number of data chunks, Tan *et al.* [31] proposed a Fine-Grained defragmentation approach (FGDefrag) that uses variable-size data groups to more accurately identify and effectively remove fragmented data. FGDefrag rewrites the fragmental chunks and the new unique chunks of each segment into a single group.

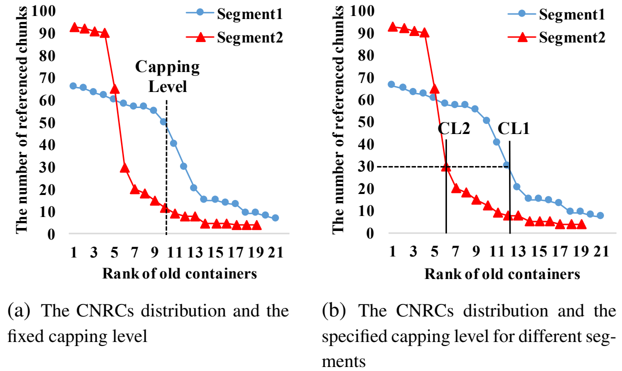
In these data chunk rewrite schemes, after some of the data chunks are rewritten, one data chunk can appear in several different containers. How to choose one container to be referenced is challenging. Wu *et al.* [32, 33] proposed a cost-efficient rewriting scheme (SMR). SMR first formulates the defragmentation as an optimization problem of selecting suitable containers and then builds a sub-modular maximization model to address this problem by selecting containers with more distinct referenced data chunks.

Caching schemes can be effective if the data chunks of the read-in container will be used again in the near future during the restore. If duplicate data chunks are spread across many containers, a large number of container reads will be required. Caching schemes cannot reduce these compulsory misses and reads. Thus, rewriting some of the duplicate data chunks that can effectively reduce the number of container reads is an alternative solution. However, it is difficult to make decisions on which duplicate chunks need to be rewritten with the minimum reduction of the deduplication ratio. This is the focus of our study.

As mentioned before, container capping is a simple and effective rewrite scheme [8]. In this scheme, the original data stream is partitioned into fixed size segments, and each segment has the size of N containers (e.g., $N \cdot 4MB$). The goal of this scheme is to limit the number of container reads required to restore a segment of data to $T+C$. C is the number of new containers generated when deduplicating the segment, and T is the *capping level*. Thus, the capping level limits the number of containers that will be read to load duplicate chunks. In capping, the first step is to count the number of data chunks (including duplicates) in the segment that belongs to an old container, called *CoNtainer Referenced Count* (CNRC). Then, these old containers containing at least one duplicate data chunk in the segment are sorted with their CNRCs in descending order. If the container is ranked lower than T, the duplicate data chunks in this container are written together with unique data chunks into the currently active container. In this way, capping ensures the higher bound of container reads in each segment. However, capping also has limitations, which motivate us to design new rewrite schemes for higher restore performance. The details are presented in the following.

3 Flexible Container Referenced Count based Design

In this section, we first discuss the limitations of the capping scheme. Then, we propose a flexible container referenced count based scheme which improves the performance of capping and achieves a good tradeoff between the number of container reads and the deduplication ratio.



(a) The CNRCs distribution and the fixed capping level

(b) The CNRCs distribution and the specified capping level for different segments

Figure 1: An example of the relationship between the CNRCs distribution and the capping level

3.1 Limitations of Capping

Capping uses a fixed threshold to control the capping level for all the segments such that a higher bound on the number of container reads can be obtained. However, the deduplication ratio is not considered as an optimization objective in the capping scheme. We have found that with some changes to the capping scheme, a better tradeoff between the number of container reads and the data deduplication ratio can be achieved.

If we consider the number of old containers involved in a segment and the distribution of CNRCs of these old containers, we find that the number of old containers and the distribution of CNRCs can be very different for different segments. If we wish to bound the reduction of deduplication ratio, some segments can keep fewer container reads than the targeted capping level while other segments may have to go beyond the targeted capping level. For example, in a segment, after sorting the old containers according to their CNRCs in descending order, we can plot out the CNRC distribution of these containers as shown in Figure 1(a). The X-axis is the rank of containers and Y-axis is the number of referenced data chunks of an involved container. Different segments have different distributions. Consider Segment 1 and Segment 2 in Figure 1(a). A fixed capping level of 10 is shown. All duplicate data chunks in the containers to the right of the capping level have to be rewritten.

In this example, the number of container reads of these two segments is capped by 20 plus the number of newly generated containers. For Segment 1, containers ranked from 10 to 12 have a relatively large number of referenced data chunks. Rewriting duplicates of data chunks in these containers will cause more reduction of deduplication ratio. The ideal capping level for Segment 1 should be either 12 or 13 according to the distribution. For Segment 2, since data

chunks are more concentrated in the higher ranking containers, rewriting the duplicate data chunks in the containers ranked beyond container 6 or 7 instead of container 10 will reduce more container reads while the increased amount of duplicate data chunks being rewritten is limited. Therefore, as shown in Figure 1(b), if we use 12 as the new capping level for Segment 1 (CL1 in the figure) and 6 as the new capping level for Segment 2 (CL2 in the figure), the number of container reads will be 1 fewer and the number of duplicate data chunks being rewritten will be even lower. Therefore, applying varied capping levels for different segments can further reduce container reads and achieve even fewer data chunk rewrites. However, how to decide the "capping levels" for different segments is challenging.

3.2 Flexible Container Referenced Count Scheme

To address the aforementioned limitations of using a fixed capping level, we proposed the Flexible Container Referenced Count based scheme (FCRC). It is an improvement of capping. Instead of using a fixed capping level as the selection threshold, FCRC uses a value of CNRC as the new threshold. It rewrites duplicate data chunks from old containers that have CNRCs lower than the threshold. In this way, different segments will have different actual capping levels. The actual capping levels are decided by the threshold and the distribution of CNRCs of these segments. The CNRC threshold T_{cnrc} can be estimated by a targeted capping level. That is, the total number of duplicate data chunks (including all copies of a duplicate) in a segment divided by the targeted capping level. Statistically, if each read-in container in a segment can contribute more than T_{cnrc} number of duplicate data chunks, the total number of old container reads to restore this segment will be bounded by the targeted capping level. Thus, rewriting duplicate data chunks in a container with CNRC lower than T_{cnrc} can achieve a similar number of container reads as using the same targeted capping level in the capping scheme.

Using a fixed value of T_{cnrc} as a threshold can make a tradeoff between the number of container reads and the deduplication ratio, but it cannot guarantee either an upper bound on the number of container reads, like capping, or a lower bound on the deduplication ratio. T_{cnrc} can only decide which data chunks belonging to the old containers with low CNRC are rewritten. The actual number of duplicate data chunks being rewritten in each segment is unknown. Also, the estimation of T_{cnrc} does not guarantee the minimal or maximal number of container reads. To solve this issue, we use a targeted deduplication ratio reduction limit $x\%$ and the targeted number of container reads Cap in one segment to generate these two bounds for T_{cnrc}. In the following, we will discuss the algorithm to calculate the two bounds and how we decide the T_{cnrc} of one segment. The old containers referenced in one segment are sorted by CNRCs in descending order before we start the following algorithm.

Bound for Deduplication Ratio Reduction We first calculate the targeted number of data chunks that can be rewritten in total (N_{rw_total}) according to the deduplication ratio reduction limit $x\%$ (i.e., after rewrite, deduplication ratio is at most x% lower than that of no the rewrite case). Let us consider a backup system as an example. Suppose in the current backup version, the total number of data chunks is N_{dc} and the deduplication ratio reduction limit is x%. For one backup version, we use the number of unique data chunks generated in the previous deduplicated version as an estimate of that value for the current version, which is N_{unique}. If we do not rewrite duplicate data chunks, the deduplication ratio is $DR = \frac{N_{dc}}{N_{unique}}$. To have at most x% of deduplication ratio reduction, we need to rewrite at most N_{rw_total} data chunks in total after deduplication. We can calculate the new deduplication ratio after rewrites by $DR \cdot (1 - x\%) = \frac{N_{dc}}{N_{unique} + N_{rw_total}}$. Finally, we have $N_{rw_total} = \frac{N_{unique} \cdot x\%}{1 - x\%}$. By dividing N_{rw_total} by the total number of segments in this backup version, we can calculate the average data chunks that can be rewritten in each segment, which is H_{rw}. H_{rw} is used for all the segments in one backup. For other applications, we can run deduplication for a short period of time. Then, we estimate H_{rw} using the current total number of data chunks being generated as N_{dc} and the current total number of unique chunks as N_{unique}.

Since the actual rewrite number can be smaller than H_{rw} in some segments, we can accumulate and distribute the saving as credits to the rest of the segments such that some of the segments can rewrite more than H_{rw} duplicate data chunks. In this way, the number of container reads can potentially be reduced, but the overall deduplication reduction limit x% is still satisfied. Note that, we cannot allow rewriting more data chunks than planned and hope segments in the future can pay for the deficit. So the accumulated credit is always non-negative.

For the i_{th} segment, suppose the accumulated actual rewrites before i_{th} segment is N_{rw}^{i-1}. We can rewrite at most $H_{rw} \cdot i - N_{rw}^{i-1}$ duplicate data chunks in the i_{th} segment. If $N_{rw}^{i-1} = H_{rw} \cdot (i-1)$ (the accumulated credit is 0), we still use H_{rw} as the rewrite limit for the i_{th} segment. In this way, we get the referenced count bound RC_{rw}^{i} of the i_{th} segment by adding the CNRCs of the old containers from low to high until the sum reaches the rewrite limit.

Bound for Container Reads Suppose the bound on the number of container reads of the i_{th} segment is RC_{reads}^{i}. The maximum number of old containers being referenced in one segment is Cap, which is the same concept as capping level in capping scheme. Suppose the accumulated number of old containers referenced before the i_{th} segment is N_{reads}^{i-1}. For the i_{th} Segment, we can tolerate referencing at most $Cap \cdot i - N_{reads}^{i-1}$ containers. Counting the old containers ranking from high to low, when the container number is $Cap \cdot i - N_{reads}^{i-1}$, the CNRC of this container is RC_{reads}^{i}. It is possible that $Cap \cdot i - N_{reads}^{i-1}$ is higher than the number of referenced old

containers in this segment. In this case, $RC^i_{reads} = 0$ and the credit is accumulated for the future.

T_{cnrc} **Calculation** If $RC^i_{rw} < RC^i_{reads}$ in the i_{th} segment, we are unable to satisfy the number of container reads and the targeted deduplication reduction limit at the same time. We choose to satisfy the deduplication ratio reduction limit first. Therefore, the threshold T_{cnrc} that we use in this segment is RC^i_{rw}. The bound on the number of container reads will be violated and the credits of the number of container reads of this segment will be negative. If $RC^i_{rw} \geq RC^i_{reads}$, the threshold T_{cnrc} can be adjusted between RC^i_{rw} and RC^i_{reads}. If T_{cnrc} of the previous segment is in between, we use the same T_{cnrc}. Otherwise, we use $\frac{RC^i_{rw}+RC^i_{reads}}{2}$ as the new T_{cnrc} which can accumulate the credit for both the number of data chunks rewritten and the number of container reads.

The performance comparisons between the FCRC scheme and the capping scheme are shown in Section 6. In general, by using a flexible referenced count based scheme, the container reads are effectively reduced compared to the capping scheme with a fixed value of capping level when the same deduplication reduction ratio is achieved.

4 Sliding Look-Back Window

Although our proposed FCRC scheme can address the tradeoff between the number of container reads and the deduplication ratio, using a fixed size segment partition causes another problem. In this section, we will first discuss the problem. Then we present a new rewrite framework called Sliding Look-Back Window (LBW) in detail.

4.1 Issues of Fixed Size Segment Partition

In both the capping and FCRC schemes, the decision to rewrite duplicate data chunks near the segment partition boundaries may have issues. Let us look at the example shown in Figure 2. There are two consecutive segments $S1$ and $S2$. The majority of data chunks of container $C1$ appear at the front of $S1$ and the majority of data chunks of $C2$ are at the front of $S2$. Due to the segment partition point, a few duplicate chunks of container $C1$ are also at the front of $S2$ and a few duplicate chunks of container $C2$ are at the end of $S1$. According to the capping scheme, the number of data chunks of $C1$ in $S1$ and that of $C2$ in $S2$ are ranked higher than the capping level. These chunks will not be rewritten. However, the ranking of container $C1$ in $S2$ and container $C2$ in $S1$ are out of the capping level. Since we do not know the past information about $C1$ in $S1$ when deduplicating $S2$ and the future information about $C2$ in $S2$ when deduplicating $S1$, these chunks (i.e., a few data chunks from $C2$ in $S1$ and a few data chunks from $C1$ in $S2$) will be rewritten. When we restore $S1$, $C1$ and $C2$ will be read in the cache. When restoring $S2$, $C2$ is already in the cache and the additional container read will not be triggered. Therefore, rewriting the

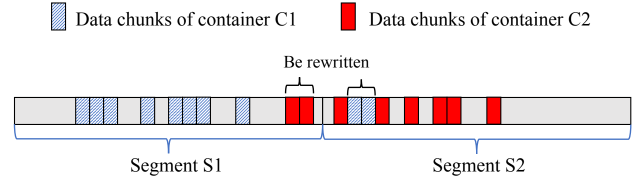


Figure 2: An example to show the fixed size segment partition issue in capping

data chunks of $C2$ appearing in $S1$ and a few data chunks of $C1$ in $S2$ is wasted.

As demonstrated in the aforementioned example, in both the capping and FCRC schemes, the container and data chunk information based on a fixed size segment do not include information about the past and future segments. On one hand, data chunks close to the front of a segment are evaluated together with subsequent chunks in the same segment, but they do not benefit from information about data chunk references and rewrite decisions made in the previous segment. On the other hand, data chunks close to the end of the segment are evaluated with earlier chunks in the segment, but there is no information about the next segment that can be used. The data chunks close to the end of the segment have a higher probability of being rewritten if most of the data chunks of their containers appear in the subsequent few segments. As a result, the rewrite decisions made with the statistics only from the current segment are less accurate.

4.2 Sliding Look-Back Window Assisted Rewrite

To prevent various cases at the boundaries of the segments and to more precisely evaluate each data chunk in a segment, we propose a fixed size "sliding" window design. In the deduplication process, the window covers a range of data chunks that have been recently generated by the deduplication engine and slides forward to cover the newly generated data chunks. A newly generated data chunk will start at the front of the sliding window. At this moment, each data chunk can be evaluated with a window-size of "past information" to make an initial rewrite decision, or a rewrite decision for this data chunk can be made later before it moves out the window. As the deduplication process continues, the window moves forward. Before a previously generated data chunk moves out of the sliding window, we can evaluate it with a window-size of "future information" to make the final rewrite decision. In this design, all data chunks are fairly evaluated with one window-size of past information and one window-size of future information which solves the issues of fixed segment partition. The details of the proposed sliding look-back window assisted rewrite scheme are presented in the following.

The overall architecture is shown in Figure 3. To move the window efficiently, the sliding window consists of N containers (4 containers of three chunks each in this example)

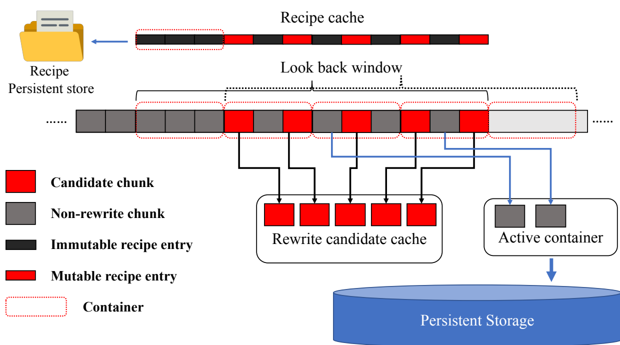


Figure 3: The overview of the sliding look-back window assisted rewrite design

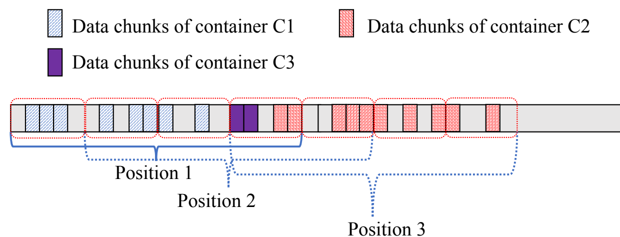


Figure 4: An example of the sliding look-window assisted rewrite scheme

and it is moved forward in container granularity. When one container size new data chunks are accumulated, they are added to the front of the window together. At the same time, one container size data chunks at the end of the window are moved out. The window moves forward to process the newly generated data chunks while these data chunks are evaluated together with other data chunks in the back of the window to make the initial rewrite decisions. This is why we call it a *Look-Back Window* (LBW). The LBW acts as a recipe cache that maintains the metadata entries of data chunks in the order covered by the LBW in the byte stream. The metadata entry for each data chunk consists of the same information found in a recipe referencing the data chunk: chunk metadata, offset in the byte stream, container ID/address, and the offset in the container. This information is used to select the data chunks to be rewritten.

To implement such a look back mechanism, some data needs to be temporarily cached in memory. In our design, we cache three types of data in memory as shown in Figure 3. First, as mentioned before, the LBW maintains a recipe cache. Second, similar to the most container based deduplication engines, we maintain an active container as chunk buffer in memory to store the unique and rewritten data chunks. The buffer is of one container size. Third, we maintain a rewrite candidate cache in memory to temporally cache the candidate data chunks. The size of the rewrite candidate cache is user-configurable. The same chunk may appear in different positions in the window, but only one copy is cached in the rewrite candidate cache. As the window moves, data chunks in the rewrite candidate cache will be gradually evicted according to the rewrite selection policy and these chunks become non-rewrite chunks. Finally, before the last container is moved out, the remaining candidate chunks in that container are rewritten to the active container. The metadata entry of a candidate chunk is mutable while the metadata entry of a non-rewrite chunk is immutable. The details of the rewrite selection policy will be presented in Section 5.

We perform the following steps to move the LBW forward. First, as each new data chunk is received, it is identified as unique or duplicate and added to the first container of the LBW. If a data chunk is unique, its data will be put into the active container and its corresponding metadata entry is stored in the recipe cache. If the data chunk is a duplicate, its metadata entry is added to the recipe cache and temporarily references the old container. Then, we make an initial rewrite decision for this data chunk. If a data chunk is a duplicate satisfying the non-rewrite condition (described later), it will not be rewritten and will be marked as a *non-rewrite chunk*. In this case, the data chunk will not be added to the rewrite candidate cache. If the rewrite decision cannot be made at this moment, this data chunk will be added to the rewrite candidate cache as one *candidate chunk* and we will wait for more information. When the most recent container in the LBW fills up with data chunks, the oldest container in the LBW (at the end of the LBW) is moved out and its corresponding metadata entries are written to the recipe on storage. Before the oldest container is moved out, the rewrite decisions of all candidate chunks in that container have to be made. Then, the LBW will move forward to cover a new container size of data chunks.

The effect of delaying rewrite decisions using a look-back window is as follows. When a data chunk is identified as a candidate chunk, we will have the information of all data chunks in the past $N-1$ containers including old container referencing information and the data chunk rewrite decisions of these data chunks. If we cannot make a decision for this candidate chunk immediately due to the limited information, we can wait to make a decision before this chunk is moved out of the window. At that time, we will know the data chunks subsequent to this data chunk in the future $N-1$ containers. With both past and future information in the LBW, a more accurate decision can be made.

Let us consider the simple example as shown in Figure 4. Suppose we use a fixed reference count value of 4 as the threshold. That is, once an old container has more than 4 duplicate data chunks in the window, the duplicate data chunks in that container will not be rewritten (non-rewrite condition). When the LBW is at Position 1, the number of data chunks from Containers 1, 2, and 3 are 8, 2, and 2, re-

spectively. Based on the non-rewrite condition, the chunks from Container 1 will not be rewritten. The newly identified data chunks from Containers 2 and 3 cannot satisfy the non-rewrite condition. We add these data chunks to the rewrite candidate cache. When the LBW is moved to Position 2, the number of data chunks from Container 2 is already 5, which satisfy the non-rewrite condition. The data chunks from Container 2 will not be rewritten. Therefore, the data chunks of Container 2 in the rewrite candidate cache are dropped. For the data chunks from Container 3, we still need to delay to make the rewrite decision. When the LBW is moved to Position 3, the data chunks from Container 3 are already at the end of the LBW. At this moment, the number of data chunks from Container 3 is still lower than the threshold. Thus, the two data chunks from Container 3 are rewritten to the active container. At the same time, the corresponding metadata entries in the recipe cache are updated with the active container information. As the oldest container is moved out of the LBW, the corresponding metadata entries are written to storage.

5 Rewrite Selection Policy for LBW

In the LBW scheme, since there are no segment partitions, we cannot directly sort the old containers by their CNRCs, which vary as the LBW moves. Therefore, either the capping level or the threshold T_{cnrc} of FCRC cannot be directly used in LBW. Based on the flexible container referenced count concept proposed in FCRC, we design a new rewrite selection policy for LBW. In the new policy, we further consider the cache-effective range and the container read efficiency of the restore process. These two criteria help us to adjust the threshold and make more accurate rewrite decisions. In this section, we first discuss the two criteria and how we can use them during the deduplication process to help make the rewrite decisions of duplicate data chunks. Then, the details of the rewrite policy are described.

5.1 Two Criteria of Restore Process

In most data chunk rewrite studies, caching-effectiveness of the restore is not considered. Some data chunks being rewritten may actually be in the cache when these chunks are restored, causing unnecessary rewrite overhead. On the other hand, how many data chunks can be restored by one container read and how long these chunks will stay in the cache can also influence the rewrite decision. Therefore, we bridge the deduplication and restore processes with the two criteria and design a new rewrite policy for LBW.

Cache-Effective Range In most data deduplication systems, caching is used to improve the restore performance. Different caching policies (e.g., chunk-based caching, container-based caching, or FAA) have different eviction priorities. However, for a very short period of time, the data chunks of a read-in container will not be evicted. In other words, if a container is read into the memory, the data chunks of that container will be cached for at least the next S container size data chunks of restore, we consider this the cache-effective range. After restoring S container size data chunks, these cached data chunks will be gradually evicted. For FAA, S is the size of FAA in terms of the number of containers. For chunk-based caching and container-based caching, S is closely related to the size of the cache and varies as the workload changes. Although it may be hard to precisely know when a data chunk will be evicted, we can still estimate S as a lower bound on the cache-effective range of each read-in container.

Importantly, the size of the LBW is related to the cache-effective range S. On one hand, if the size of the LBW is much smaller than S, some rewritten data chunks may have a probability to be in the cache. This will cause an unnecessary reduction of the deduplication ratio. On the other hand, if LBW is much larger than S, a data chunk that was not rewritten due to the same data chunk existing in LBW may not be cached during the restore process (i.e., should have been rewritten). Also, a large LBW requires maintaining a larger amount of metadata in memory and caching a larger number of candidate chunks, which may not be acceptable. Therefore, maintaining an LBW with the size compatible with the cache-effective range is a good tradeoff.

Container Read Efficiency A container's read efficiency can be measured by the number of data chunks used in a short duration after it is read-in and the concentration level of these chunks in a given restore range. Please note that at different moments of the restore, the same old container may have a different read efficiency. If one container read can restore more data chunks, that container read is more efficient than rewriting those data chunks. With the same number of data chunks being restored from one old container, if those data chunks are more concentrated in a small range of the byte stream, fewer data chunks will be cached and the cached chunks can be evicted earlier. In this case, the cache space can be more efficiently used and more container reads can be potentially eliminated.

To quantitatively define a container read efficiency, we use two measurements of an old container in the LBW: *container referenced count* and *referenced chunk closeness*. The container referenced count (CNRC) was used in FCRC to decide the rewrite candidates in a segment. Similarly, we define the container referenced count of an old container as the number of times the data chunks in that container appear in the LBW (duplications are counted) at a point in time, which is $CNRC_{lbw}$. $CNRC_{lbw}$ of an old container can change with each movement of the LBW as data chunks are added in and moved out. In fact, the nature of capping is to rank the $CNRC_{lbw}$ of the old containers appearing in a segment and to rewrite the data chunks that belong to the old containers with low CNRC. The FCRC scheme rewrites the containers with CNRC lower than the CNRC threshold.

We define the referenced chunk closeness L_c as: the average distance (measured by the number of data chunks) between a data chunk that will potentially trigger a container read and the rest of the data chunks of that container in the same LBW, divided by the size of LBW (the total number of chunks in the LBW), where $L_c < 1$. Note that if one data chunk appears in multiple positions, only the first one is counted. Smaller L_c means the data chunks are closer.

5.2 Rewrite Selection Policy for LBW

The rewrite selection policy is designed based on the information covered by the LBW. Suppose the LBW size is S_{LBW} which is measured by the number of containers. When the LBW moves forward for one container size, a container size of data chunks (added container) will be added to the front of the LBW and one container size of data chunks (evicted container) will be removed from the end of the LBW. There are five steps to make the rewrite decision: 1) process the added container, classify the data chunks into three categories: unique chunks, non-rewrite chunks (duplicate data chunks that will not be rewritten), and candidate chunks (duplicate data chunks that may be rewritten); 2) update the metadata entries of data chunks in the added container and add the identified candidate chunks to the rewrite candidate cache; 3) recalculate the $CNRC_{lbw}$ of old containers that contain the data chunks in the rewrite candidate cache and reclassify these data chunks according to the updated $CNRC_{lbw}$ and the threshold T_{dc}. At this step, some of the data chunks in the candidate cache are identified as non-rewrite chunks and are evicted from the cache; 4) rewrite the remaining candidate chunks in the evicted container to the active container and update their metadata entries in LBW. The updated metadata entries of data chunks in the evicted container are written to the recipe persistently; 5) every S_{LBW} containers movement of the LBW, adjust the CNRC threshold T_{dc}, which is used to filter out the non-rewrite chunks. The details of each step are explained in the following.

In **Step 1**, by searching the indexing table, the unique data chunks and duplicate data chunks are identified. The unique data chunks are written to the active container. Next, for each duplicate data chunk, we search backward in the LBW. If some data chunks from the same old container appear ahead of that duplicate data chunk and these chunks are non-rewrite chunks, the duplicate data chunk is marked as a non-rewrite chunk. Otherwise, the duplicate chunk becomes a candidate chunk and it is added to the candidate cache. If a candidate chunk is the first chunk of its container appearing in the most current S containers, it means this container has not been used or referenced at least S containers range. This chunk will potentially trigger one container read and thus we call it the *leading chunk* of this container. As the LBW moves, we can determine the range and distance of data chunks from the same container appearing after the leading chunk. Based on future information, we can decide whether these data chunks should be rewritten or not.

In **Step 2**, the metadata entries of data chunks in the added container are updated with the metadata information of the referenced chunks in the active container or old container. Note that the same duplicate data chunk can appear in multiple old containers due to past rewrites. One of the old containers should be referenced in the metadata entry. There are some studies on how to optimize the selection of old containers. For an example, an approximately optimal solution is proposed by [32, 33]. Here, when the indexing table returns the list of old containers that store a data chunk, we use a greedy algorithm to select the container that has the most chunks in the current LBW. Other algorithms can also be easily applied to our proposed policy.

In **Step 3**, we follow the definition of $CNRC_{lbw}$ described in Section 5.1 to calculate the latest $CNRC_{lbw}$ of each old container referenced by the candidate chunks. The candidate chunks from the old containers whose $CNRC_{lbw}$ are higher than the threshold T_{dc} (the calculation of T_{dc} will be discussed in Step 5) are removed from the rewrite candidate cache. They become non-rewrite chunks and their metadata entries in the recipe cache still reference the old containers.

In **Step 4**, the data chunks in the evicted container that are still in the candidate cache are rewritten to the active container before we move the evicted container out of the LBW. Their metadata entries in the recipe cache are updated to reference the active container. If a leading chunk of one old container is rewritten, which means the $CNRC_{lbw}$ of its container is still lower than T_{dc}, other data chunks from the same container appearing in the current LBW will be rewritten too. Once a data chunk is rewritten, it is evicted from the candidate cache.

In **Step 5**, T_{dc} is adjusted to make better tradeoffs between deduplication ratio reduction and the number of container reads. Every S_{lbw} size movement is one LBW moving cycle and the T_{dc} is adjusted at the end of each cycle. Similar to the FCRC scheme in Section 3.2, we calculate the two bounds of T_{dc} in the i_{th} moving cycle: RC^i_{rw} and RC^i_{reads}. The referenced chunk closeness of current and previous moving cycle are L^i_c and L^{i-1}_c and they are used to further adjust the threshold according to the data chunk closeness. The algorithm details to calculate T_{dc} are described in **Algorithm 1**. Finally, we get the threshold T^{i+1}_{dc} for the $i+1_{th}$ moving cycle.

6 Performance Evaluation

To comprehensively evaluate our design, we implemented four deduplication engines including normal deduplication with no rewrite (Normal), capping scheme (Capping), flexible container referenced count based scheme (FCRC), and Sliding Look-Back Window scheme (LBW). For the restore engine, we implemented two state-of-the-art caching designs, Forward Assembly Area (FAA) [8] and Adaptive

Algorithm 1 T_{dc} Adjusting Algorithm

Input: $T_{dc}^i, L_c^i, L_c^{i-1}, RC_{rw}^i, RC_{reads}^i$
Output: T_{dc}^{i+1}

```
if RC^i_rw < RC^i_reads then
    T^{i+1}_dc ← RC^i_rw
else
    if RC^i_reads < T^i_dc AND T^i_dc < RC^i_rw then
        T^i_start ← T^i_dc
    else
        T^i_start ← (RC^i_reads + RC^i_rw)/2
    end if
    if L^i_c < L^{i-1}_c then
        T^{i+1}_dc ← T^i_start − 1
    else
        T^{i+1}_dc ← T^i_start + 1
    end if
end if
```

Look-Ahead window Chunk based Caching (ALACC) [10]. Since any restore experiments will include one deduplication engine and one restore engine, there are 8 combinations. Speed factor and deduplication ratio are used as the evaluation metrics, which are determined by the workload, deduplication and restore engine designs. The two metrics are platform independent. Since the container I/O time dominates the whole restore time, especially when low performance storage (e.g., HDD or tape) are used to store containers, a higher speed factor (fewer container reads) represents a higher restore performance. The high performance storage scenario (e.g., SSD) is not considered in this paper.

6.1 Experimental Setup and Data Sets

The prototype is deployed on a Dell PowerEdge R430 server with a Seagate ST1000NM0033-9ZM173 SATA hard disk of 1TB capacity as the storage. The server has a 2.40GHz Intel Xeon with 24 cores and 64GB of memory. In our experiments, the container size is set to 4MB. To fairly compare the performance of these four deduplication engines, we try to use the same amount of memory in each engine. That is, for Capping and FCRC, the size of each segment is fixed at 5 containers. For LBW, the total size of the recipe cache and the rewrite candidate cache size is also 5 containers. For FAA, the look-ahead window size is 8 containers and the forward assembly area size is also 8 containers. ALACC has a 4 container size FAA, a 4 container size chunk cache, and an 8 container size look-ahead window.

In the experiments, we use six deduplication traces from the File System and Storage Lab (FSL) [34, 35, 36] as shown in Table 1. The traces were collected by the File Systems and Storage Lab (CS Department, Stony Brook University) and its collaborators. These traces cover two types of file system snapshots: MacOS and Homes. The former was collected on a Mac OS X Snow Leopard server running in an academic computer lab which contains SMTP, MySQL, HTTP, FTP, Wiki, etc., the latter contains snapshots of students' home directories from a shared network file system of the File System and Storage Lab. MAC1, MAC2, and MAC3 are three different backup traces from MacOS. FSL1, FSL2, and FSL3 are from FSL /home directory snapshots from 2014 to 2015.

Table 1: Characteristics of datasets

Dataset	MAC1	MAC2	MAC3	FSL1	FSL2	FSL3
TS(TB)[1]	2.04	1.97	1.97	2.78	2.93	0.63
US(GB)[2]	121.8	134.5	142.3	183.7	202	71
ASR(%)[3]	45.99	46.50	46.80	42.49	39.96	33.60
SV(%)[4]	99.28	96.60	95.48	97.87	98.64	93.00
IV[5]	5	20	60	5	20	60

[1] TS stands for the total data size of the trace.
[2] US stands for the unique data size of the trace.
[3] ASR stands for the average self-referenced chunk ratio in each version.
[4] SV stands for the average similarity between each version.
[5] IV stands for the time interval (days-based) between each version.

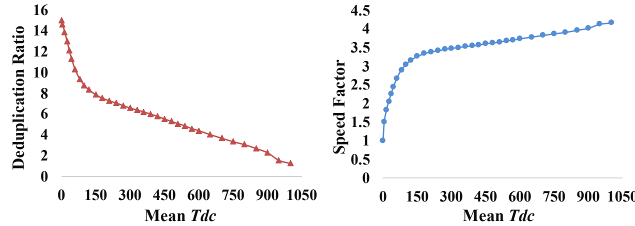
Each trace has 10 full backup snapshots and we choose the version with the 4KB average chunk size. As shown in SV (average similarity between versions) of Table 1, in each trace at least 93% of the data chunks appears in the previous versions. This indicates that most data chunks are duplicate data chunks and they are fragmented. The time interval (IV) between versions of each trace is also shown in Table 1. We select different IV for each trace such that the fragmentation and version similarity are varied in different traces. Each trace record contains the chunk hash value (17 bytes), chunk size, and compression ratio. The implemented deduplication engines use the chunk size and hash value to simulate the deduplication process without the chunking and chunk hash generating steps. Before replaying a trace, all data in the cache is cleared. For each experiment, we run the traces 3 times and present the average value.

6.2 Influence of LBW Size and Mean Value of T_{dc}

In the LBW design, different configurations of LBW size and different mean values of CNRC threshold T_{dc} will influence the deduplication ratio and the speed factor. We use the MAC2 trace as an example and design a set of experiments to investigate these relationships. In the restore process, we use FAA as the restore caching scheme and set the FAA size to 8 containers. To investigate the influence of LBW size, we fix the mean T_{dc} to 120 by setting the two bounds while the size of the LBW changes from 16 to 2 containers. The results are shown in Table 2. As the LBW size decreases, the deduplication ratio decreases slightly. It indicates that, when the threshold configuration is the same, if we reduce the LBW size, less information is maintained in the window and more data chunks are selected to be rewritten. The deduplication ratio decreases even faster when the LBW size is smaller than the FAA size, which is caused by a large num-

Table 2: The change of the deduplication ratio (DR) and the speed factor (SF) with different LBW size and a fixed mean T_{dc} (mean T_{dc} =120)

LBW size	16	14	12	10	8	6	4	2
DR	8.53	8.53	8.53	8.50	8.45	8.37	8.27	8.07
SF	2.80	2.84	2.87	2.92	3.01	3.14	3.16	3.18



(a) The deduplication ratio variation (b) The speed factor variation

Figure 5: The change of deduplication ratio and speed factor with different mean T_{dc} and a fixed LBW size (LBW=5)

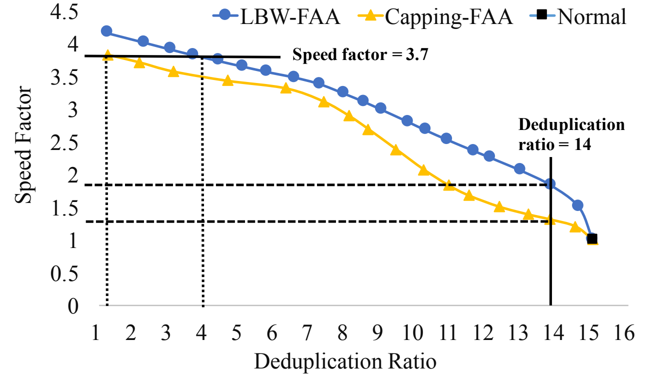


Figure 6: The speed factor of Capping and LBW when different deduplication ratios are achieved for the same trace. The Normal scheme is a black point in the figure

ber of unnecessary rewrites due to a small LBW size. In contrast, when the LBW size is close to or even smaller than the FAA size as shown in Table 2, there is a relatively large increase in the speed factor. Therefore, to achieve a good tradeoff between the deduplication ratio and the speed factor, the LBW size should be compatible with the FAA size, which is the cache-effective range in this case.

In another experiment, we fix the LBW size to 8 containers, FAA size to 8 containers, and vary T_{dc} from 0 to 1000. With a higher T_{dc}, more old containers will be rewritten since it is harder for each old container to have more than T_{dc} duplicate data chunks in the LBW. Therefore, as shown in Figure 5(a), the deduplication ratio decreases as T_{dc} increases. Also, the speed factor increases since more data chunks are rewritten as shown in Figure 5(b). When T_{dc} is zero, there is no rewrite and the deduplication regresses to the Normal deduplication case. On the other hand, when T_{dc}= 1000 (one container stores 1000 data chunks on average), almost every duplicate chunk is re-written. Thus, no deduplication is performed (i.e., deduplication ratio is 1) and the speed factor is 4. Note that, the decreasing of the deduplication ratio as well as the increasing of the speed factor are at a faster pace when T_{dc} is less than 100. When T_{dc} is larger than 100, the two curves vary linearly at a slower pace. It shows that we can achieve a good tradeoff between deduplication ratio reduction and restore performance improvement when T_{dc} is set smaller than 100 for this trace. The "knee" point can be different in other workloads. It is closely related to the CNRC distribution of old containers.

6.3 Deduplication Ratio & Speed Factor Comparison

In this subsection, we compare the performance of Capping with LBW in terms of deduplication ratio and speed factor when deduplicating the last version of MAC2. We choose FAA as the restore engine for Capping and LBW. In both cases, we set the FAA size to 8 containers. For Capping, the segment size is set to 5 containers and we run with the capping level varying from 0 to 70. For LBW, the memory for the rewrite candidate cache and the recipe cache is also 5 containers and the LBW size is fixed to 8 containers. We vary T_{dc} from 0 to 1000 by adjusting the targeted deduplication ratio and the targeted number of container reads. To fairly compare Capping and LBW, we need to compare the speed factor of the two designs when they have the same deduplication ratio.

The results are shown in Figure 6. We can clearly conclude that when the deduplication ratio is the same, the speed factor of LBW is always higher than that of Capping. For example, when the speed factor is 3.7, Capping has a deduplication ratio of 1.2, which indicates a very high rewrite number (close to no deduplication). With the same speed factor, LBW has a deduplication ratio of 4, which means only 25% of data chunks are stored. With the deduplication ratio of 14, Capping has a speed factor of 1.27 and LBW has a speed factor of 1.75 which is about 38% higher than that of Capping. If no rewrite is performed (Normal deduplication process), the deduplication ratio is 15 while the speed factor is close to 1. When the deduplication ratio increases from 8 to 14, which indicates fewer data chunks are rewritten and is close to the no rewrite case (Normal), the speed factor of LBW ranges from 20% to 40% higher than that of Capping. In general, LBW can achieve a better tradeoff between deduplication ratio reduction and speed factor improvement especially when the deduplication ratio is high.

6.4 Restore Performance Comparison

To obtain the performance of Normal, Capping, FCRC, and LBW as deduplication engines, we pair them with two restore engines (FAA and ALACC). Since a small reduction of deduplication ratio (i.e., fewer data chunk rewrites)

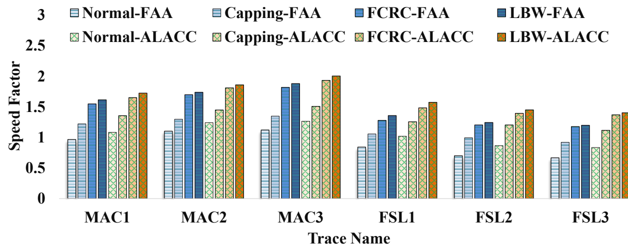
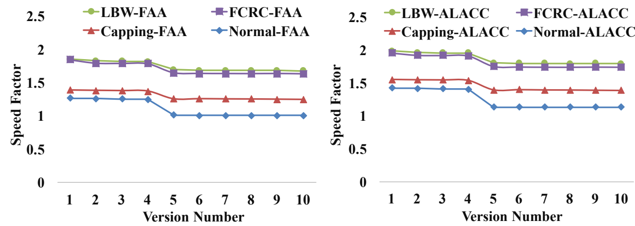


Figure 7: The mean speed factor of Normal, Capping, FCRC, and LBW for the 10 versions of six traces when restoring with FAA and ALACC



(a) The speed factor of using FAA as the restore caching scheme

(b) The speed factor of using ALACC as the restore caching scheme

Figure 8: The speed factor comparison of 8 combinations of deduplication-restore experiments when deduplicating and restoring the trace MAC2

is preferred in production, we configure the parameters of Capping, FCRC, and LBW in the following experiments to achieve the deduplication ratio reduction of 7%.

Each trace has 10 backup versions and the mean speed factors achieved by all 8 combination schemes are shown in Figure 7. If data chunk rewrite is not applied (Normal design) in the deduplication process, even with FAA or ALACC as the restore caching schemes, the speed factor is always lower than 1.3. In three FSL traces, the speed factor is even lower than 1, which means reading out a 4MB size container cannot even restore 1MB of data. It indicates that the data chunks are fragmented and the two caching schemes cannot effectively reduce a large number of container reads. By rewriting duplicate data chunks, the speed factor is improved. For all 6 traces, we find that with the same deduplication ratio, LBW always achieves the highest speed factor (best restore performance) when the same restore engine is used. The speed factor of FCRC is lower than that of LBW, but higher than that of Capping. Due to a lower average self-reference count ratio (ASR) shown in Table 1, the duplicate data chunks of FSL traces are more fragmented than that of MAC traces. Thus, the overall speed factor of FSL traces is lower than those of MAC traces. In general, when using the same restore caching scheme, the speed factor of LBW is up to 97% higher than that of Normal, 41% higher than that of Capping, and 7% higher than that of FCRC.

We also compared the speed factors of different backup versions in trace MAC2. The evaluation results of using FAA and ALACC as the restore caching schemes are shown in Figure 8(a) and Figure 8(b), respectively. Among the 10 versions, the duplicate data chunks are more fragmented after version 4. Therefore, there is a clear speed factor drop at version 5. As shown in both figures, by using FCRC, the speed factor is improved a lot compared with the Normal and Capping schemes. LBW further improves the speed factor when compared with FCRC by addressing the rewrite accuracy issues caused by a fixed segment partition and considering the cache-effective range as well as container read efficiency. So the speed factor of LBW is always the highest in all 10 versions. In general, with the same deduplication ratio, LBW can effectively reduce the number of container reads such that its restore performance is always the best.

7 Conclusion and Future Work

Rewriting duplicate data chunks during the deduplication process is an important and effective way to reduce container reads, which cannot be achieved by caching schemes during restore. In this paper, we discuss the limitations of capping design and propose the FCRC scheme based on capping to improve rewrite selection accuracy. To further reduce the inaccurate rewrite decisions caused by the fixed segment partition used by capping and FCRC, we propose the sliding look-back window based rewrite approach. By combining the look-back mechanism and the caching effects of the restore process, our approach makes better tradeoffs between the reduction of deduplication ratio and container reads. In our experiments, the speed factor of LBW is always better than that of capping and FCRC when the deduplication ratio is the same. In our future work, the adaptive LBW size and more intelligent rewrite policies will be investigated. Also, the garbage collection of the deleted data chunks will be investigated together with the rewrite design.

Acknowledgments

We thank all the members in CRIS group for providing the useful comments to improve our design. We would like to thank our shepherd, Keith Smith, for his useful comments, suggestions, and help in the paper revision. This work was partially supported by NSF awards 1421913, 1439622, 1525617, and 1812537.

References

[1] Robert E Fontana, Gary M Decad, and SR Hetzler. The impact of areal density and millions of square inches (MSI) of produced memory on petabyte shipments of TAPE, NAND flash, and HDD storage class memories.

In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST 13)*, pages 1–8. IEEE, 2013.

[2] https://www.backblaze.com/blog/hdd-vs-ssd-in-data-centers/.

[3] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, pages 1–14, 2008.

[4] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference, Haifa, Israel (SYSTOR 12)*, pages 1–12, 2012.

[5] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David HC Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *2011 IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 581–586. IEEE, 2011.

[6] Young Jin Nam, Dongchul Park, and David HC Du. Assuring demanded read performance of data deduplication storage with backup datasets. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 12)*, pages 201–208. IEEE, 2012.

[7] 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 *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 181–192, 2014.

[8] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 183–198, 2013.

[9] Bo Mao, Hong Jiang, Suzhen Wu, Yinjin Fu, and Lei Tian. SAR: SSD assisted restore optimization for deduplication-based storage systems in the cloud. In *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage (NAS 12)*, pages 328–337. IEEE, 2012.

[10] Zhichao Cao, Hao Wen, Fenggang Wu, and David HC Du. ALACC: accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 309–323. USENIX Association, 2018.

[11] Wei Zhang, Tao Yang, Gautham Narayanasamy, and Hong Tang. Low-cost data deduplication for virtual machine backup in cloud storage. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13)*, 2013.

[12] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, pages 271–294, 2011.

[13] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 331–344, 2015.

[14] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2011.

[15] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *2009 IEEE 17th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 09)*, pages 1–9. IEEE, 2009.

[16] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*, pages 89–101, 2002.

[17] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: A scalable secondary storage. In *7th USENIX Conference on File and Storage Technologies (FAST 09)*, pages 197–210, 2009.

[18] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. iDedup: latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, pages 1–14, 2012.

[19] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. Cachededup: in-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, 2016.

[20] Biplob K Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[21] Yoshihiro Tsuchiya and Takashi Watanabe. DBLK: Deduplication for primary block storage. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST 11)*, pages 1–5. IEEE, 2011.

[22] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device mapper target for data deduplication. In *2014 Ottawa Linux Symposium*, 2014.

[23] 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 *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 315–322, 2016.

[24] Zhuan Chen and Kai Shen. OrderMergeDedup: Efficient, failure-consistent deduplication on flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 291–299, 2016.

[25] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data deduplication-large scale study and system design. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 285–296, 2012.

[26] J. bonwick. ZFS deduplication, November 2009. https://blogs.oracle.com/bonwick/zfs-deduplication-v2.

[27] Veritas netbackup cloud administrator's guide. https://www.veritas.com/content/support/en_us/doc/ka6j000000004pkaaq.

[28] Dongchul Park, Ziqi Fan, Young Jin Nam, and David HC Du. A lookahead read cache: Improving read performance for deduplication backup storage. *Journal of Computer Science and Technology*, 32(1):26–40, 2017.

[29] Michal Kaczmarczyk and Cezary Dubnicki. Reducing fragmentation impact with forward knowledge in backup systems with deduplication. In *Proceedings of the 8th Annual International Systems and Storage Conference, Haifa, Israel (SYSTOR 15)*, pages 1–12, 2015.

[30] 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 Transactions on Parallel and Distributed Systems*, 27(3):855–868, 2016.

[31] Yujuan Tan, Baiping Wang, Jian Wen, Zhichao Yan, Hong Jiang, and Witawas Srisa-an. Improving restore performance in deduplication-based backup systems via a fine-grained defragmentation approach. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[32] Jie Wu, Yu Hua, Pengfei Zuo, and Yuanyuan Sun. A cost-efficient rewriting scheme to improve restore performance in deduplication systems. In *2017 IEEE 33th Symposium on Mass Storage Systems and Technologies (MSST 17)*, 2017.

[33] Jie Wu, Yu Hua, Pengfei Zuo, and Yuanyuan Sun. Improving restore performance in deduplication systems via a cost-efficient rewriting scheme. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[34] http://tracer.filesystems.org/.

[35] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 261–272, 2012.

[36] Zhen Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, et al. A long-term user-centric analysis of deduplication patterns. In *2016 IEEE 32nd Symposium on Mass Storage Systems and Technologies (MSST 16)*, pages 1–7. IEEE, 2016.

DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching

Zaoxing Liu[⋆], *Zhihao Bai*[⋆], *Zhenming Liu*[†], *Xiaozhou Li*[◁],
Changhoon Kim[‡], *Vladimir Braverman*[⋆], *Xin Jin*[⋆], *Ion Stoica*[◇]
[⋆]*Johns Hopkins University* [†]*College of William and Mary* [◁]*Celer Network* [‡]*Barefoot Networks* [◇]*UC Berkeley*

Abstract

Load balancing is critical for distributed storage to meet strict service-level objectives (SLOs). It has been shown that a fast cache can guarantee load balancing for a clustered storage system. However, when the system scales out to multiple clusters, the fast cache itself would become the bottleneck. Traditional mechanisms like cache partition and cache replication either result in load imbalance between cache nodes or have high overhead for cache coherence.

We present DistCache, a new distributed caching mechanism that provides provable load balancing for large-scale storage systems. DistCache co-designs cache allocation with cache topology and query routing. The key idea is to partition the hot objects with independent hash functions between cache nodes in different layers, and to adaptively route queries with the power-of-two-choices. We prove that DistCache enables the cache throughput to increase linearly with the number of cache nodes, by unifying techniques from expander graphs, network flows, and queuing theory. DistCache is a general solution that can be applied to many storage systems. We demonstrate the benefits of DistCache by providing the design, implementation, and evaluation of the use case for emerging switch-based caching.

1 Introduction

Modern planetary-scale Internet services (e.g., search, social networking and e-commerce) are powered by large-scale storage systems that span hundreds to thousands of servers across tens to hundreds of racks [1–4]. To ensure satisfactory user experience, the storage systems are expected to meet strict service-level objectives (SLOs), regardless of the workload distribution. A key challenge for scaling out is to achieve load balancing. Because real-world workloads are usually highly-skewed [5–8], some nodes receive more queries than others, causing hot spots and load imbalance. The system is bottlenecked by the overloaded nodes, resulting in low throughput and long tail latencies.

Caching is a common mechanism to achieve load balancing [9–11]. An attractive property of caching is that caching $O(n \log n)$ hottest objects is enough to balance n storage nodes, regardless of the query distribution [9]. The cache size only relates to the number of storage nodes, despite the number of objects stored in the system. Such property leads to recent advancements like SwitchKV [10] and NetCache [11] for balancing clustered key-value stores.

Unfortunately, the small cache solution cannot scale out to multiple clusters. Using one cache node per cluster only provides *intra-cluster* load balancing, but not *inter-cluster* load balancing. For a large-scale storage system across many clusters, the load between clusters (where each cluster can be treated as one "big server") would be imbalanced. Using another cache node, however, is not sufficient, because the caching mechanism requires the cache to process *all* queries to the $O(n \log n)$ hottest objects [9]. In other words, the cache throughput needs to be no smaller than the *aggregate* throughput of the storage nodes.

As such, it requires another caching layer with multiple cache nodes for inter-cluster load balancing. The challenge is on cache allocation. Naively replicating hot objects to all cache nodes incurs high overhead for cache coherence. On the other hand, simply partitioning hot objects between the cache nodes would cause the load to be imbalanced between the cache nodes. The system throughput would still be bottlenecked by one cache node under highly-skewed workloads. Thus, the key is to carefully partition and replicate hot objects, in order to avoid load imbalance between the cache nodes, and to reduce the overhead for cache coherence.

We present DistCache, a new distributed caching mechanism that provides provable load balancing for large-scale storage systems. DistCache enables a "one big cache" abstraction, i.e., an *ensemble* of fast cache nodes acts as a single ultra-fast cache. DistCache co-designs cache allocation with multi-layer cache topology and query routing. The key idea is to use independent hash functions to partition hot objects between the cache nodes in different layers, and to apply the power-of-two-choices [12] to adaptively route queries.

Using independent hash functions for cache partitioning ensures that if a cache node is overloaded in one layer, then

the set of hot objects in this node would be distributed to multiple cache nodes in another layer with high probability. This intuition is backed up by a rigorous analysis that leverages expander graphs and network flows, i.e., we prove that there exists a solution to split queries between different layers so that no cache node would be overloaded in any layer. Further, since a hot object is only replicated in each layer once, it incurs minimal overhead for cache coherence.

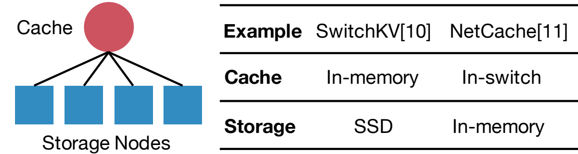
Using the power-of-two-choices for query routing provides an efficient, distributed, online solution to split the queries between the layers. The queries are routed to the cache nodes in a distributed way based on cache loads, without central coordination and without knowing what is the optimal solution for query splitting upfront. We leverage queuing theory to show it is asymptotically optimal. The major difference between our problem and the balls-and-bins problem in the original power-of-two-choices algorithm [12] is that our problem hashes objects into cache nodes, and queries to the same object *reuse* the same hash functions to choose hash nodes, instead of using a *new random source* to sample two nodes for each query. We show that the power-of-two-choices makes a "life-or-death" improvement in our problem, instead of a "shaving off a log n" improvement.

DistCache is a general caching mechanism that can be applied to many storage systems, e.g., in-memory caching for SSD-based storage like SwitchKV [10] and switch-based caching for in-memory storage like NetCache [11]. We provide a concrete system design to scale out NetCache to demonstrate the power of DistCache. We design both the control and data planes to realize DistCache for the emerging switch-based caching. The controller is highly scalable as it is off the critical path. It is only responsible for computing the cache partitions and is not involved in handling queries. Each cache switch has a local agent that manages the hot objects of its own partition.

The data plane design exploits the capability of programmable switches, and makes innovative use of *in-network telemetry* beyond traditional network monitoring to realize *application-level functionalities*—disseminating the loads of cache switches by piggybacking in packet headers, in order to aid the power-of-two-choices. We apply a two-phase update protocol to ensure cache coherence.

In summary, we make the following contributions.

- We design and analyze DistCache, a new distributed caching mechanism that provides provable load balancing for large-scale storage systems (§3).
- We apply DistCache to a use case of emerging switch-based caching, and design a concrete system to scale out an in-memory storage rack to multiple racks (§4).
- We implement a prototype with Barefoot Tofino switches and commodity servers, and integrate it with Redis (§5). Experimental results show that DistCache scales out linearly with the number of racks, and the cache coherence protocol incurs minimal overhead (§6).



Example	SwitchKV[10]	NetCache[11]
Cache	In-memory	In-switch
Storage	SSD	In-memory

Figure 1: Background on caching. If the cache node can absorb *all* queries to the hottest $O(n \log n)$ objects, the load on the storage nodes is guaranteed to be balanced [9].

2 Background and Motivation

2.1 Small, Fast Cache for Load Balancing

As a building block of Internet applications, it is critical for storage systems to meet strict SLOs. Ideally, given the per-node throughput T, a storage system with n nodes should guarantee a total throughput of $n \cdot T$. However, real-world workloads are usually high-skewed, making it challenging to guarantee performance [5–8]. For example, a measurement study on the Memcached deployment shows that about 60-90% of queries go to the hottest 10% objects [5].

Caching is a common mechanism to achieve load balancing for distributed storage, as illustrated in Figure 1. Previous work has proven that if the cache node can absorb *all* queries to the hottest $O(n \log n)$ objects, then the load on n storage servers is guaranteed to be balanced, despite query distribution and the total number of objects [9]. However, it also requires that the cache throughput needs to be at least $n \cdot T$ to not become the system bottleneck. Based on this theoretical foundation, SwitchKV [10] uses an in-memory cache to balance SSD-based storage nodes, and NetCache [11] uses a switch-based cache to balance in-memory storage nodes. Empirically, these systems have shown that caching a few thousand objects is enough for balancing a hundred storage nodes, even for highly-skewed workloads like Zipfian-0.9 and Zipfian-0.99 [10, 11].

2.2 Scaling out Distributed Storage

The requirement on the cache performance limits the system scale. Suppose the throughput of a cache node is $\widetilde{T} = c \cdot T$. The system can scale to at most a cluster of c storage nodes. For example, given that the typical throughput of a switch is 10-100 times of that of a server, NetCache [11] can only guarantee load balancing for 10-100 storage servers. As such, existing solutions like SwitchKV [10] and NetCache [11] are constrained to one storage cluster, which is typically one or two racks of servers.

For a cloud-scale distributed storage system that spans many clusters, the load between the clusters can become imbalanced, as shown in Figure 2(a). Naively, we can put another cache node in front of all clusters to balance the load between clusters. At first glance, this seems a nice solution, since we can first use a cache node in each cluster for *intra-cluster* load balancing, and then use an upper-layer cache node for *inter-cluster* load balancing. However, now each cluster becomes a "big server", of which the throughput is already $\widetilde{T}$. Using

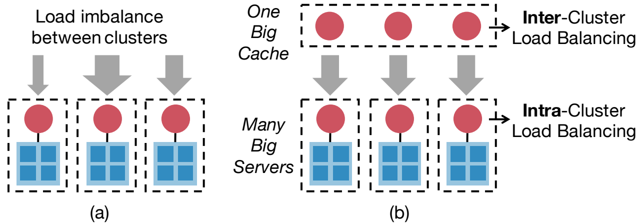


Figure 2: Motivation. (a) A cache node only guarantees load balancing for its own cluster, but the load between clusters can be unbalanced. (b) Use one cache node in each cluster for *intra-cluster* load balancing, and another layer of cache nodes for *inter-cluster* load balancing. The challenge is on cache allocation.

only one cache node cannot meet the cache throughput requirement, which is $m\widetilde{T}$ for m clusters. While using multiple upper-layer cache nodes like Figure 2(b) can potentially meet this requirement, it brings the question of how to allocate hot objects to the upper-layer cache nodes. We examine two traditional cache allocation mechanisms.

Cache partition. A straightforward solution is to partition the object space between the upper-layer cache nodes. Each cache node only caches the hot objects of its own partition. This works well for uniform workloads, as the cache throughput can grow linearly with the number of cache nodes. But remember that under uniform workloads, the load on the storage nodes is already balanced, obviating the need for caching in the first place. The whole purpose of caching is to guarantee load balancing for skewed workloads. Unfortunately, cache partition would cause load imbalance between the upper-layer cache nodes, because multiple hot objects can be partitioned to the same upper-layer cache node, making one cache node become the system bottleneck.

Cache replication. Cache replication replicates the hot objects to all the upper-layer cache nodes, and the queries can be uniformly sent to them. As such, cache replication can ensure that the load between the cache nodes is balanced, and the cache throughput can grow linearly with the number of cache nodes. However, cache replication introduces high overhead for cache coherence. When there is a write query to a cached object, the system needs to update both the primary copy at the storage node and the cached copies at the cache nodes, which often requires an expensive two-phase update protocol for cache coherence. As compared to cache partition which only caches a hot object in one upper-layer cache node, cache replication needs to update all the upper-layer cache nodes for cache coherence.

Challenge. Cache partition has low overhead for cache coherence, but cannot increase the cache throughput linearly with the number of cache nodes; cache replication achieves the opposite. Therefore, the main challenge is to carefully partition and replicate the hot objects, in order to (i) avoid load imbalance between upper-layer cache nodes, and to (ii) reduce the overhead for cache coherence.

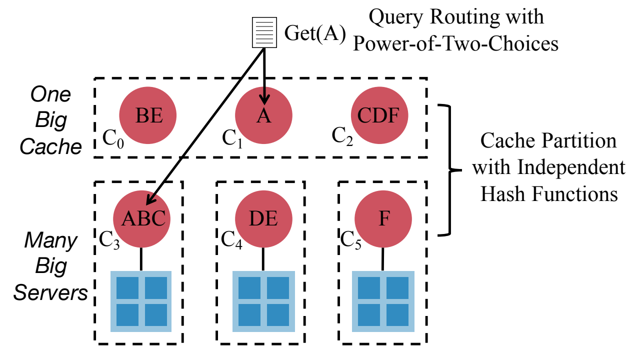


Figure 3: Key idea. (i) Use independent hash functions to partition hot objects in different layers. (ii) Use the power-of-two-choices to route queries, e.g., route $Get(A)$ to either cache node C_1 or cache node C_3 based on cache load.

3 DistCache Caching Mechanism Design

3.1 Key Idea

We design DistCache, a new distributed caching mechanism to address the challenge described in §2.2. As illustrated by Figure 3, our key idea is to use independent hash functions for cache allocation and the power-of-two-choices for query routing, in order to balance the load between cache nodes. Our mechanism only caches an object at most once in a layer, incurring minimal overhead for cache coherence. We first describe the mechanism and the intuitions, and then show why it works in §3.2.

Cache allocation with independent hash functions. Our mechanism partitions the object space with independent hash functions in different layers. The lower-layer cache nodes primarily guarantee intra-cluster load balancing, each of which only caches hot objects for its own cluster, and thus each cluster appears as one "big server". The upper-layer cache nodes are primarily for inter-cluster load balancing, and use a different hash function for partitioning. The intuition is that if one cache node in a layer is overloaded by receiving too many queries to its cached objects, because the hash functions of the two layers are independent, the set of hot objects would be distributed to multiple cache nodes in another layer with high probability. Figure 3 shows an example. While cache node C_3 in the lower layer is overloaded with three hot objects (A, B and C), the three objects are distributed to three cache nodes (C_0, C_1 and C_2) in the upper layer. The upper-layer cache nodes only need to absorb queries for objects (e.g., A and B) that cause the imbalance *between* the clusters, and do not need to process queries for objects (e.g., D and F) that already *spread out* in the lower-layer cache nodes.

Query routing with the power-of-two-choices. The cache allocation strategy only tells that there exists a way to handle queries without overloading any cache nodes, but it does not tell how the queries should be *split* between the layers.

Conceivably, we could use a controller to collect global measurement statistics to infer the query distribution. Then the controller can compute an optimal solution and enforce it at the senders. Such an approach has high system complexity, and the responsiveness to dynamic workloads depends on the agility of the control loop.

Our mechanism uses an *efficient, distributed, online* solution based on the power-of-two-choices [12] to route queries. Specifically, the sender of a query only needs to look at the loads of the cache nodes that cache the queried object, and sends the query to the less-loaded node. For example, the query $Get(A)$ in Figure 3 is routed to either C_1 or C_3 based on their loads. The key advantage of our solution is that: it is distributed, so that it does not require a centralized controller or any coordination between senders; it is online, so that it does not require a controller to measure the query distribution and compute the solution, and the senders do not need to know the solution upfront; it is efficient, so that its performance is close to the optimal solution computed by a controller with perfect global information (as shown in §3.2). Queries to hit a lower-layer cache node can either pass through an arbitrary upper-layer node, or totally bypass the upper-layer cache nodes, depending on the actual use case, which we describe in §3.4.

Cache size and multi-layer hierarchical caching. Suppose there are m clusters and each cluster has l servers. First, we let each lower-layer cache node cache $O(l \log l)$ objects for its own cluster for *intra-cluster* load balancing, so that a total of $O(ml \log l)$ objects are cached in the lower layer and each cluster appears like one "big server". Then for *inter-cluster* load balancing, the upper-layer cache nodes only need to cache a total of $O(m \log m)$ objects. This is different from a single ultra-fast cache at a front-end that handles all ml servers directly. In that case, $O(ml \log(ml))$ objects need to be cached based on the result in [9]. However, in DistCache, we have an extra upper-layer (with the same total throughput as ml servers) to "refine" the query distribution that goes to the lower-layer, which reduces the effective cache size in the lower layer to $O(ml \log l)$. Thus, this is not a contradiction with the result in [9]. While these $O(m \log m)$ inter-cluster hot objects also need to be cached in the lower layer to enable the power-of-two-choices, most of them are also hot inside the clusters and thus have already been contained in the $O(ml \log l)$ intra-cluster hot objects.

Our mechanism can be applied recursively for multi-layer hierarchical caching. Specifically, applying the mechanism to layer i can balance the load for a set of "big servers" in layer i-1. Query routing uses the *power-of-k-choices* for k *layers*. Note that using more layers actually increases the total number of cache nodes, since each layer needs to provide a total throughput at least equal to that of all storage nodes. The benefit of doing so is on reducing the cache size. When the number of clusters is no more than a few hundred, a cache node has enough memory with two layers.

3.2 Analysis

Prior work [9] has shown that caching $O(n \log n)$ hottest objects in a single cache node can balance the load for n storage nodes for any query distribution. In our work, we replace the single cache node with multiple cache nodes in two layers to support a larger scale. Therefore, based on our argument on the cache size in §3.1, we need to prove that the two-layer cache can absorb all queries to the hottest $O(m \log m)$ objects under any query distribution for all m clusters. We first define a mathematical model to formalize this problem.

System model. There are k hot objects $\{o_0, o_1, \ldots, o_{k-1}\}$ with query distribution $P = \{p_0, p_1, \ldots, p_{k-1}\}$, where p_i denotes the fraction of queries for object o_i, and $\sum_{i=0}^{k-1} p_i = 1$. The total query rate is R, and the query rate for object o_i is $r_i = p_i \cdot R$. There are in total $2m$ cache nodes that are organized to two groups $A = \{a_0, a_1, ..., a_{m-1}\}$ and $B = \{b_0, b_1, ..., b_{m-1}\}$, which represent the upper and lower layers, respectively. The throughput of each cache node is $\widetilde{T}$.

The objects are mapped to the cache nodes with two independent hash functions $h_0(x)$ and $h_1(x)$. Object o_i is cached in a_{j_0} in group A and b_{j_1} in group B, where $j_0 = h_0(i)$ and $j_1 = h_1(i)$. A query to o_i can be served by either a_{j_0} or b_{j_1}.

Goal. Our goal is to evaluate the total query rate R the cache nodes can support, in terms of m and $\widetilde{T}$, regardless of query distribution P, as well as the relationship between k and m. Ideally, we would like $R \approx \alpha m \widetilde{T}$ where α is a small constant (e.g., 1), so that the operator can easily provision the cache nodes to meet the cache throughput requirement (i.e., no smaller than the total throughput of storage nodes).

If we can set k to be $O(m \log m)$, it means that the cache nodes can absorb all queries to the hottest $O(m \log m)$ objects, despite query distribution. Combining this result with the cache size argument in §3.1, we can prove that the distributed caching mechanism can provide performance guarantees for large-scale storage systems across multiple clusters.

A perfect matching problem in a bipartite graph. The key observation of our analysis is that the problem can be converted to finding a perfect matching in a bipartite graph. Intuitively, if a perfect matching exists, the requests to k hot objects can be completely absorbed from the two layers of cache nodes. Specifically, we construct a bipartite graph $G = (U, V, E)$, where U is the set of vertices on the left, V is the set of vertices on the right, and E is the set of edges. Let U represent the set of objects, i.e., $U = \{o_0, o_1, ..., o_{k-1}\}$. Let V represent the set of cache nodes, i.e., $V = A \cup B = \{a_0, a_1, ..., a_{m-1}, b_0, b_1, ..., b_{m-1}\}$. Let E represent the hash functions mapping from the objects to the cache nodes, i.e., $E = \{e_{o_i, a_{j_0}} | h_0(i) = j_0\} \cup \{e_{o_i, b_{j_1}} | h_1(i) = j_1\}$. Given a query distribution P and a total query rate R, we define a perfect matching in G to represent that the workload can be supported by the cache nodes.

Definition 1. *Let $\Gamma(v)$ be the set of neighbors of vertex v in G.*

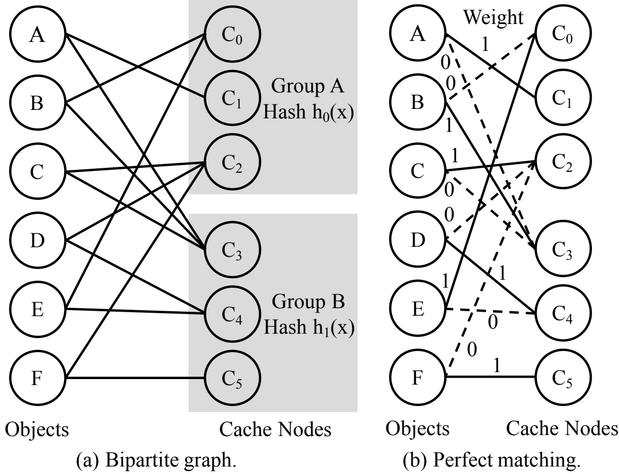


Figure 4: Example for analysis. (a) A bipartite graph constructed for the scenario in Figure 3. (b) A perfect matching for query routing when all objects have a query rate of 1, and all cache nodes have a throughput of 1.

A weight assignment $W = \{w_{i,j} \in [0, \widetilde{T}] | e_{i,j} \in E\}$ *is a perfect matching of* G *if*

1. $\forall o_i \in U: \quad \sum_{v \in \Gamma(o_i)} w_{o_i,v} = p_i \cdot R$, *and*
2. $\forall v \in V: \quad \sum_{u \in \Gamma(v)} w_{u,v} \leq \widetilde{T}$.

In this definition, $w_{i,j}$ denotes the portion of the queries to object i served by cache node j. Condition 1 ensures that for any object o_i, its query rate $p_i \cdot R$ is fully served. Condition 2 ensures that for any cache node v, its load is no more than $\widetilde{T}$, i.e., no single cache node is overloaded.

When a perfect matching exists, it is feasible to serve all the queries by the cache nodes. We use the example in Figure 4 to illustrate this. Figure 4(a) shows the bipartite graph constructed for the scenario in Figure 3, which contains six hot objects (A-F) and six cache nodes in two layers (C_0-C_5). The edges are built based on two hash functions $h_0(x)$ and $h_1(x)$. Figure 4(b) shows a perfect matching for the case that all objects have the same query rate $r_i = 1$ and all cache nodes have the same throughput $\widetilde{T} = 1$. The number besides an edge denotes the weight of an edge, i.e., the rate of the object served by the cache node. For instance, all queries to A are served by C_1. This is a simple example to illustrate the problem. In general, the query rates of the objects do not have to be the same, and the queries to one object may be served by multiple cache nodes.

Step 1: existence of a perfect matching. We first show the existence of a perfect matching for any given total rate R and *any* query distribution P. We have the following lemma to demonstrate how big the total rate R can be in terms of $\widetilde{T}$, for any P. For the full proof of Lemma 1, we refer the readers to §A.2 in the technical report.

Lemma 1. *Let* α *be a suitably small constant. If* $k \leq m^{\beta}$ *for some constant* β *(i.e., k and m are polynomial-related) and* $\max_i(p_i) \cdot R \leq \widetilde{T}/2$, *then for any* $\varepsilon > 0$, *there exists a perfect matching for* $R = (1-\varepsilon)\alpha \cdot m\widetilde{T}$ *and any P, with high probability for sufficiently large m.*

Proof sketch of Lemma 1: We utilize the results and techniques developed from expander graphs and network flows. (i) We first show that G has the so-called *expansion property* with high probability. Intuitively, the property states that the neighborhood of any subset of nodes in U *expands*, i.e., for any $S \subseteq U$, $|\Gamma(S)| \geq |S|$. It has been observed that such properties exist in a wide range of random graphs [13]. While our G behaves similar to random bipartite graphs, we need the expansion property to hold for S in any size, which is stricter than the standard definition (which assumes S is not too large) and thus requires more delicate probabilistic techniques. (ii) We then show that if a graph has the expansion property, then it has a perfect matching. This step can be viewed as a generalization of Hall's theorem [14] in our setting. Hall's theorem states that a balanced bipartite graph has a perfect (non-fractional) matching if and only if for any subset S of the left nodes, $|\Gamma(S)| \geq |S|$. and perfect matching can be fractional. This step can be proved by the max-flow-min-cut theorem, i.e., expansion implies large cut, and then implies large matching.

Step 2: finding a perfect matching. Demonstrating the existence of a perfect matching is insufficient since it just ensures the queries can be absorbed but does not give the actual weight assignment W, i.e., how the cache nodes should serve queries for each P to achieve R. This means that the system would require an algorithm to compute W and an mechanism to enforce W. As discussed in §3.1, instead of doing so, we use the power-of-two-choices to "emulate" the perfect matching, without the need to know what the perfect matching is. The quality of the mechanism is backed by Lemma 2, which we prove using queuing theory. The detailed proof can be found in §A.3 of the technical report [15].

Lemma 2. *If a perfect matching exists for G, then the power-of-two-choices process is stationary.*

Stationary means that the load on the cache nodes would converge, and the system is "sustainable" in the sense that the system will never "blow up" (i.e., build up queues in a cache node and eventually drop queries) with query rate R.

Proof sketch of Lemma 2: Showing this lemma requires us to use a powerful building block in query theory presented in [16, 17]. Consider $2m$ exponential random variables with rate $\widetilde{T}_i > 0$. Each non-empty set of cache nodes $S \subseteq [2m]$, has an associated Poisson arrival process with rate $\lambda_S \geq 0$ that joins the shortest queue in S with ties broken randomly. For each non-empty subset $Q \subseteq [2m]$, define the traffic intensity on Q as

$$\rho_Q = \frac{\sum_{S \subseteq Q} \lambda_S}{\mu_Q},$$

where $\mu_Q = \sum_{i \in Q} \widetilde{T}_i$. Note that the total rate at which objects served by Q can be greater than the numerator of (3.2) since other requests may be allowed to be served by some or all of

the cache nodes in Q. Let $\rho_{\max} = \max_{Q\subseteq[2m]}\{\rho_Q\}$. Given the result in [16, 17], if we can show $\rho_{\max} < 1$, then the Markov process is positive recurrent and has a stationary distribution. In fact, our cache querying can be described as an arrival process (§A.3 of [15]). Finally, we show that $\rho_{\max}$ is less than 1 and thus the process is stationary.

Step 3: main theorem. Based on Lemma 1 and Lemma 2, we can prove that our distributed caching mechanism is able to provide a performance guarantee, despite query distribution.

Theorem 1. *Let α be a suitable constant. If $k \leq m^{\beta}$ for some constant β (i.e., k and m are polynomial-related) and $\max_i(p_i) \cdot R \leq \widetilde{T}/2$, then for any $\varepsilon > 0$, the system is stationary for $R = (1-\varepsilon)\alpha \cdot m\widetilde{T}$ and any P, with high probability for sufficiently large m.*

Interpretation of the main theorem: As long as the query rate of a single hot object o_i is no larger than $\widetilde{T}/2$ (e.g., half of the entire throughput in a cluster rack), DistCache can support a query rate of $\approx m\widetilde{T}$ for any query distributions to the k hot objects (where k can be fairly large in terms of m) by using the power-of-two-choices protocol to route the queries to the cached objects. The key takeaways are presented in the following section.

3.3 Remarks

Our problem isn't a balls-in-bins problem using the original power-of-two-choices. The major difference is that our problem hashes objects into cache nodes, and queries to the same object by *reusing* the same hash functions, instead of using a *new random source* to sample two nodes for each query. In fact, without using the power-of-two-choices, the system is in *non-stationary*. This means that the power-of-two-choices makes a "life-or-death" improvement in our problem, instead of a "shaving off a log n" improvement. While we refer to the technical report [15] for detailed discussions, we have a few important remarks.

- **Nonuniform number of cache nodes in two layers.** For simplicity we use the same number of m cache nodes per layer in the system. However, we can generalize the analysis to accommodate the cases of different numbers of caches nodes in two layers, as long as $\min(m_0, m_1)$ is sufficiently large, where m_0 and m_1 are the number of upper-layer and lower-layer cache nodes respectively. While it requires m to be sufficiently large, it is not a strict requirement, because the load imbalance issue is only significant when m is large.
- **Nonuniform throughput of cache nodes in two groups.** Although our analysis assumes the throughput of a cache node is $\widetilde{T}$, we can generalize it to accommodate the cases of nonuniform throughput by treating a cache node with a large throughput as multiple smaller cache nodes with a small throughput.
- **Cache size.** As long as the number of objects and the number of cache nodes are polynomially-related ($k \leq m^{\beta}$), the system is able to provide the performance guarantee. It is more relaxed than $O(m \log m)$. Therefore, by setting $k = O(m \log m)$, the cache nodes are able to absorb all queries to the hottest $O(m \log m)$ objects, making the load on the m clusters balanced.
- **Maximum query rate for one object.** The theorem requires that the maximum query rate for one object is no bigger than half the throughput of one cache node. This is not a severe restriction for the system, because a cache node is orders of magnitude faster than a storage node.
- **Performance guarantee.** The system can guarantee a total throughput of $R = (1-\varepsilon)\alpha \cdot mT$, which scales linearly with m and T. In practice, α is close to 1.

3.4 Use Cases

DistCache is a general solution that can be applied to scale out various storage systems (e.g., key-value stores and file systems) using different storage mediums (e.g., HDD, SDD and DRAM). We describe two use cases.

Distributed in-memory caching. Based on the performance gap between DRAMs and SSDs, a fast in-memory cache node can be used balance an SSD-based storage cluster, such as SwitchKV [10]. DistCache can scale out SwitchKV by using another layer of in-memory cache nodes to balance multiple SwitchKV clusters. While it is true that multiple in-memory cache nodes can be balanced using a faster switch-based cache node, applying DistCache obviates the need to introduce a new component (i.e., a switch-based cache) to the system. Since the queries are routed to the cache and storage nodes by the network, queries to the lower-layer cache nodes can totally bypass the upper-layer cache nodes.

Distributed switch-based caching. Many low-latency storage systems for interactive web services use more expensive in-memory designs. An in-memory storage rack can be balanced by a switch-based cache like NetCache [11], which directly caches the hot objects in the data plane of the ToR switch. DistCache can scale out NetCache to multiple racks by caching hot objects in a higher layer of the network topology, e.g., the spine layer in a two-layer leaf-spine network. As discussed in the remarks (§3.3), DistCache accommodates the cases that the number of spine switches is smaller and each spine switch is faster. As for query routing, while queries to hit the leaf cache switches need to inevitably go through the spine switches, these queries can be arbitrarily routed through any spine switches, so that the load on the spine switches can be balanced.

Note that while existing solutions (e.g., NetCache [11]) directly embeds caching in the switches which may raise concerns on deployment, another option for easier deployment is to use the cache switches as *stand-alone specialized appliances* that are separated from the switches in the datacenter network. DistCache can be applied to scale out these specialized switch-based caching appliances as well.

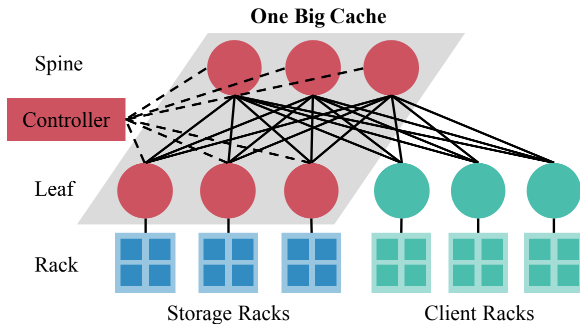


Figure 5: Architecture for distributed switch-based caching.

4 DistCache for Switch-Based Caching

To demonstrate the benefits of DistCache, we provide a concrete system design for the emerging switch-based caching. A similar design can be applied to other use cases as well.

4.1 System Architecture

Emerging switch-based caching, such as NetCache [11] is limited to one storage rack. We apply DistCache to switch-based caching to provide load balancing for cloud-scale key-value stores that span many racks. Figure 5 shows the architecture for a two-layer leaf-spine datacenter network.

Cache Controller. The controller computes the cache partitions, and notifies the cache switches. It updates the cache allocation under system reconfigurations, e.g., adding new racks and cache switches, and system failures; and thus updating the allocation is an infrequent task. We assume the controller is reliable by replicating on multiple servers with a consensus protocol such as Paxos [18]. The controller is not involved in handling storage queries in the data plane.

Cache switches. The cache switches provide two critical functionalities for DistCache: (1) caching hot key-value objects; (2) distributing switch load information for query routing. First, a local agent in the switch OS receives its cache partition from the controller, and manages the hot objects for its partition in the data plane. Second, the cache switches implement a lightweight in-network telemetry mechanism to distribute their load information by piggybacking in packet headers. The functionalities for DistCache are invoked by a reserved L4 port, so that DistCache does not affect other network functionalities. We use existing L2/L3 network protocols to route packets, and do not modify other network functionalities already in the switch.

ToR switches at client racks. The ToR switches at client racks provide query routing. It uses the power-of-two-choices to decide which cache switch to send a query to, and uses existing L2/L3 network protocols to route the query.

Storage servers. The storage servers host the key-value store. DistCache runs a shim layer in each storage server to integrate the in-network cache with existing key-value store software like Redis [19] and Memcached [20]. The shim layer also implements a cache coherence protocol to guarantee the consistency between the servers and cache switches.

Clients. DistCache provides a client library for applications to access the key-value store. The library provides an interface similar to existing key-value stores. It maps function calls from applications to DistCache query packets, and gathers DistCache reply packets to generate function returns.

4.2 Query Handling

A key advantage of DistCache is that it provides a distributed *on-path* cache to serve queries at line rate. Read queries on cached objects (i.e., cache hit) are directly replied by the cache switches, *without the need to visit storage servers*; read queries on uncached objects (i.e., cache miss) and write queries are forwarded to storage servers, *without any routing detour*. Further, while the cache is distributed, our query routing mechanism based on the power-of-two-choices ensures that the load between the cache switches is balanced.

Query routing at client ToR switches. Clients send queries via the client library, which simply translates function calls to query packets. The complexity of query routing is done at the ToR switches of the client racks. The ToR switches use the switch on-chip memory to store the loads of the cache switches. For each read query, they compare the loads of the switches that contain the queried object in their partitions, and pick the less-loaded cache switch for the query. After the cache switch is chosen, they use the existing routing mechanism to send the query to the cache switch. The routing mechanism can pick a routing path that balances the traffic in the network, which is orthogonal to this paper. Our prototype uses a mechanism similar to CONGA [21] and HULA [22] to choose the least loaded path to the cache switch.

For a cache hit, the cache switch copies the value from its on-chip memory to the packet, and returns the packet to the client. For a cache miss, the cache switch forwards the packet to the corresponding storage server that stores the queried object. Then the server processes the query and replies to the client. Figure 6 shows an example. A client in rack R_3 sends a query to read object A. Suppose A is cached in switch S_1 and S_3, and is stored in a server in rack R_0. The ToR switch S_6 uses the power-of-two-choices to decide whether to choose S_1 or S_3. Upon a cache hit, the cache switch (either S_1 or S_3) directly replies to the client (Figure 6(a)). Upon a cache miss, the query is sent to the server. But no matter whether the leaf cache (Figure 6(b)) or the spine cache (Figure 6(c)) is chosen, there is no routing detour for the query to reach R_0 after a cache miss.

Write queries are directly forwarded to the storage servers that contain the objects. The servers implement a cache coherence protocol for data consistency as described in §4.3.

Query processing at cache switches. Cache switches use the on-chip memory to cache objects in their own partitions. In programmable switches such as Barefoot Tofino [23], the on-

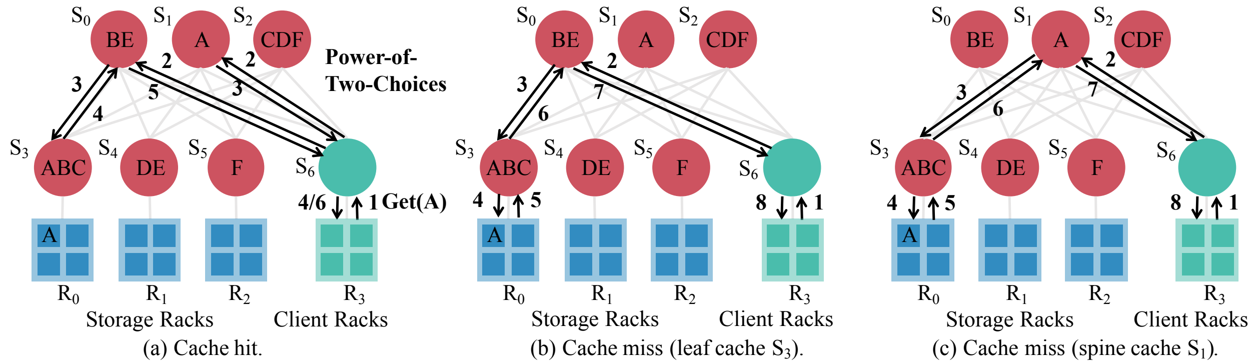


Figure 6: Query handling for $Get(A)$. S_6 uses the power-of-two-choices to decide whether to send $Get(A)$ to S_1 or S_3. (a) Upon a cache hit, the switch directly replies the query, without visiting the storage server. (b, c) Upon a cache miss, the query is forwarded to the storage server without routing detour.

chip memory is organized as register arrays spanning multiple stages in the packet processing pipeline. The packets can read and update the register arrays at line rate. We uses the same mechanism as NetCache [11] to implement a key-value cache that can support variable-length values, and a heavy-hitter (HH) detector that the switch local agent uses to decide what top k hottest objects in its partition to cache.

In-network telemetry for cache load distribution. We use a light-weight in-network telemetry mechanism to distribute the cache load information for query routing. The mechanism piggybacks the switch load (i.e., the total number of packets in the last second) in the packet headers of reply packets, and thus incurs minimal overhead. Specifically, when a reply packet of a query passes a cache switch, the cache switch adds its load to the packet header. Then when the reply packet reaches the ToR switch of the client rack, the ToR switch retrieves the load in the packet header to update the load stored in its on-chip memory. To handle the case that the cache load may become stale without enough traffic for piggybacking, we can add a simple aging mechanism that would gradually decrease a load to zero if the load is not updated for a long time. Note that aging is commonly supported by modern switch ASICs, but it is not supported by P4 yet, and thus is not implemented in our prototype.

4.3 Cache Coherence and Cache Update

Cache coherence. Cache coherence ensures data consistency between storage servers and cache switches when write queries update the values of the objects. The challenge is that an object may be cached in multiple cache switches, and need to be updated atomically. Directly updating the copies of an object in the cache switches may result in data inconsistency. This is because the cache switches are updated asynchronously, and during the update process, there would be a mix of old and new values at different switches, causing read queries to get different values from different switches.

We leverage the classic two-phase update protocol [24] to ensure strong consistency, where the first phase invalidates all copies and the second phase updates all copies. To apply the protocol to our scenario, after receiving a write query, the storage server generates a packet to invalidate the copies in the cache switches. The packet traverses a path that includes all the switches that cache the object. The return of the invalidation packet indicates that all the copies are invalidated. Otherwise, the server resends the invalidation packet after a timeout. Figure 7(a) shows an example that the copies of object A are invalidated by an invalidation packet via path R_0-S_3-S_1-S_3-R_0. After the first phase, the server can update its primary copy, and send an acknowledgment to the client, instead of waiting for the second phase, as illustrated by Figure 7(b). This optimization is safe, since all copies are invalid. Finally, in the second phase, the server sends an update packet to update the values in the cache switches, as illustrated by Figure 7(c).

Cache update. The cache update is performed in a decentralized way without the involvement of the controller. We use a similar mechanism as NetCache [11]. Specifically, the local agent in each switch uses the HH detector in the data plane to detect hot objects in its own partition, and decides cache insertions and evictions. Cache evictions can be directly done by the agent; cache insertions require the agent to contact the storage servers. Slightly different from NetCache, DistCache uses a cleaner, more efficient mechanism to unify cache insertions and cache coherence. Specifically, the agent first inserts the new object into the cache, but marks it as invalid. Then the agent notifies the server; the server updates the cached object in the data plane using phase 2 of cache coherence, and serializes this operation with other write queries. As for comparison, in NetCache, the agent copies the value from the server to the switch via the switch control plane (which is slower than the data plane), and during the copying, the write queries to the object are blocked on the server.

4.4 Failure Handling

Controller failure. The controller is replicated on multiple servers for reliability (§4.1). Since the controller is only re-

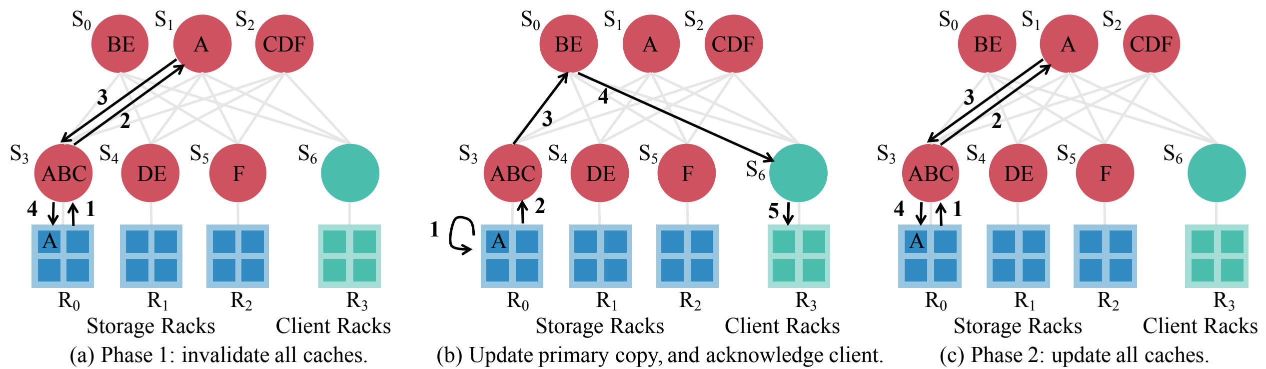


Figure 7: Cache coherence is achieved by a two-phase update protocol in DistCache. The example shows the process to handle an update to object A stored in rack R_0 with the two-phase update protocol.

sponsible for cache allocation, even if all servers of the controller fail, the data plane is still operational and hence processes queries. The servers can be simply rebooted.

Link failure. A link failure is handled by existing network protocols, and does not affect the system, as long as the network is connected and the routing is updated. If the network is partitioned after a link failure, the operator would choose between consistency and availability, as stated by the CAP theorem. If consistency were chosen, all writes should be blocked; if availability were chosen, queries can still be processed, but cache coherence cannot be guaranteed.

ToR switch failure. The servers in the rack would lose access to the network. The switch needs to be rebooted or replaced. If the switch is in a storage rack, the new switch starts with an empty cache and uses the cache update process to populate its cache. If the switch is in a client rack, the new switch initializes the loads of all cache switches to be zero, and uses the in-network telemetry mechanism to update them with reply packets.

Other Switch failure. If the switch is not a cache switch, the failure is directly handled by existing network protocols. If the switch is a cache switch, the system loses throughput provided by this switch. If it can be quickly restored (e.g., by rebooting), the system simply waits for the switch to come back online. Otherwise, the system remaps the cache partition of the failed switch to other switches, so that the hot objects in the failed switch can still be cached, alleviating the impact on the system throughput. The remapping leverages consistent hashing [25] and virtual nodes [26] to spread the load. Finally, if the network is partitioned due to a switch failure, the operator would choose consistency or availability, similar to that of a link failure.

5 Implementation

We have implemented a prototype of DistCache to realize distributed switch-based caching, including cache switches, client ToR switches, a controller, storage servers and clients.

Cache switch. The data plane of the cache switches is written in the P4 language [27], which is a domain-specific language to program the packet forwarding pipelines of data plane devices. P4 can be used to program the switches that are based on Protocol Independent Switch Architecture (PISA). In this architecture, we can define the packet formats and packet processing behaviors by a series of match-action tables. These tables are allocated to different processing stages in a forwarding pipeline, based on hardware resources. Our implementation is compiled to Barefoot Tofino ASIC [23] with Barefoot P4 Studio software suite [28]. In the Barefoot Tofino switch, we implement a key-value cache module uses 16-byte keys, and contains 64K 16-byte slots per stage for 8 stages, providing values at the granularity of 16 bytes and up to 128 bytes without packet recirculation or mirroring. The Heavy Hitter detector module contains a Count-Min sketch [29], which has 4 register arrays and 64K 16-bit slots per array, and a Bloom filter, which has 3 register arrays and 256K 1-bit slots per array. The telemetry module uses one 32-bit register slot to store the switch load. We reset the counters in the HH detector and telemetry modules in every second. The local agent in the switch OS is written in Python. It receives cache partitions from the controller, and manages the switch ASIC via the switch driver using a Thrift API generated by the P4 compiler. The routing module uses standard L3 routing which forwards packets based on destination IP address.

Client ToR switch. The data plane of client ToR switches is also written in P4 [27] and is compiled to Barefoot Tofino ASIC [23]. Its query routing module contains a register array with 256 32-bit slots to store the load of cache switches. The routing module uses standard L3 routing, and picks the least loaded path similar to CONGA [21] and HULA [22].

Controller, storage server, and client. The controller is written in Python. It computes cache partitions and notifies the result to switch agents through Thrift API. The shim layer at each storage server implements the cache coherence protocol, and uses the hiredis library [30] to hook up with Redis [19]. The client library provides a simple key-value interface. We use the client library to generate queries with different distributions and different write ratios.

6 Evaluation

6.1 Methodology

Testbed. Our testbed consists of two 6.5Tbps Barefoot Tofino switches and two server machines. Each server machine is equipped with a 16 core-CPU (Intel Xeon E5-2630), 128 GB total memory (four Samsung 32GB DDR4-2133 memory), and an Intel XL710 40G NIC.

The goal is to apply DistCache to switch-based caching to provide load balancing for cloud-scale in-memory key-value stores. Because of the limited hardware resources we have, we are unable to evaluate DistCache at full scale with tens of switches and hundreds of servers. Nevertheless, we make the most of our testbed to evaluate DistCache by dividing switches and servers into multiple logical partitions and running real switch data plane and server software, as shown in Figure 8. Specifically, a physical switch emulates several virtual switches by using multiple queues and uses counters to rate limit each queue. We use one Barefoot Tofino switch to emulate the spine switches, and the other to emulate the leaf switches. Similarly, a physical server emulates several virtual servers by using multiple queues. We use one server to emulate the storage servers, and the other to emulate the clients. We would like to emphasize that the testbed runs the real switch data plane and runs the Redis key-value store [19] to process real key-value queries.

Performance metric. By using multiple processes and using the pipelining feature of Redis, our Redis server can achieve a throughput of 1 MQPS. We use Redis to demonstrate that DistCache can integrate with production-quality open-source software that is widely deployed in real-world systems. We allocate the 1 MQPS throughput to the emulated storage servers equally with rate limiting. Since a switch is able to process a few BQPS, the bottleneck of the testbed is on the Redis servers. Therefore, we use rate limiting to match the throughput of each emulated switch to the aggregated throughput of the emulated storage servers in a rack. We normalize the system throughput to the throughput of one emulated key-value server as the performance metric.

Workloads. We use both uniform and skewed workloads in the evaluation. The uniform workload generates queries to each object with the same probability. The skewed workload follows Zipf distribution with a skewness parameter (e.g., 0.9, 0.95, 0.99). Such skewed workload is commonly used to benchmark key-value stores [10, 31], and is backed by measurements from production systems [5, 6]. The clients use approximation techniques [10, 32] to quickly generate queries according to a Zipf distribution. We store a total of 100 million objects in the key-value store. We use Zipf-0.99 as the default query distribution to show that DistCache performs well even under extreme scenarios. We vary the skewness and the write ratio (i.e., the percentage of write queries) in the experiments to evaluate the performance of DistCache under different scenarios.

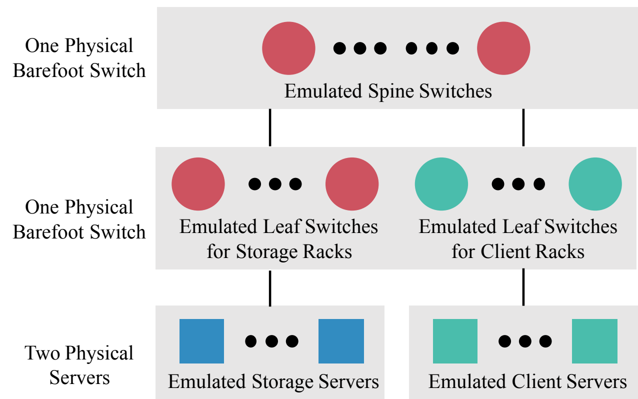


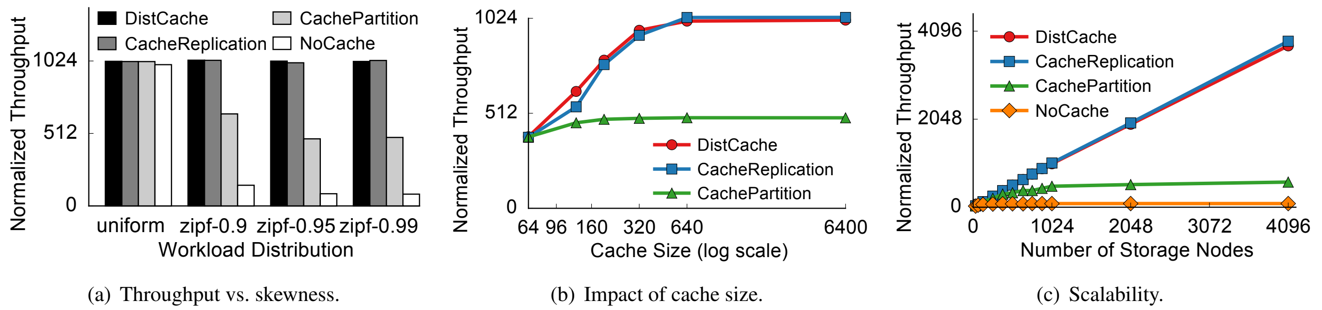
Figure 8: Evaluation setup. The testbed emulates a datacenter with a two-layer leaf-spine network by dividing switches and servers into multiple logical partitions.

Comparison. To demonstrate the benefits of DistCache, we compare the following mechanisms in the experiments: DistCache, CacheReplication, CachePartition, and NoCache. As described in §2.2, CacheReplication is to replicate the hot objects to all the upper layer cache nodes, and CachePartition partitions the hot objects between nodes. In NoCache, we do not cache any objects in both layers. Note that CachePartition performs the same as only using NetCache for each rack (i.e., only caching in the ToR switches).

6.2 Performance for Read-Only Workloads

We first evaluate the system performance of DistCache. By default, we use 32 spine switches and 32 storage racks. Each rack contains 32 servers. We populate each cache switch with 100 hot objects, so that 64 cache switches provide a cache size of 6400 objects. We use read-only workloads in this experiment, and show the impact of write queries in §6.3. We vary workload skew, cache size and system scale, and compare the throughputs of the four mechanisms under different scenarios.

Impact of workload skew. Figure 9(a) shows the throughput of the four mechanisms under different workload skews. Under the uniform workload, the four mechanisms have the same throughput, since the load between the servers is balanced and all the servers achieve their maximum throughputs. However, when the workload is skewed, the throughput of NoCache significantly decreases, because of load imbalance. The more skewed the workload is, the lower throughput NoCache achieves. CachePartition performs better than NoCache, by caching hot objects in the switches. But its throughput is still limited because of load imbalance between cache switches. CacheReplication provides the optimal throughput under read-only workloads as it replicates hot objects in all spine switches. DistCache provides comparable throughput to CacheReplication by using the distributed caching mechanism. And we will show in §6.3 that DistCache performs better than CacheReplication under writes because of low overhead for cache coherence.



(a) Throughput vs. skewness.

(b) Impact of cache size.

(c) Scalability.

Figure 9: System performance for read-only workloads.

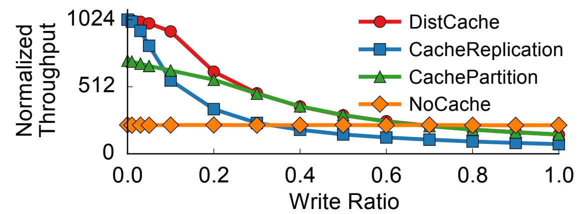
Impact of cache size. Figure 9(b) shows the throughput of the three mechanisms under different cache sizes. CachePartition achieves higher throughput with more objects in the cache. Because the skewed workload still causes load imbalance between cache switches, the benefits of caching is limited for CachePartition. Some spine switches quickly become overloaded after caching some objects. As such, the throughput improvement is small for CachePartition. On the other hand, CacheReplication and DistCache gain big improvements by caching more objects, as they do not have the load imbalance problem between cache switches. The curves of CacheReplication and DistCache become flat after they achieve the saturated throughput.

Scalability. Figure 9(c) shows how the four mechanisms scale with the number of servers. NoCache does not scale because of the load imbalance between servers. Its throughput stops to improve after a few hundred servers, because the overloaded servers become the system bottleneck under the skewed workload. CachePartition performs better than NoCache as it uses the switches to absorb queries to hot objects. However, since the load imbalance still exists between the cache switches, the throughput of CachePartition stops to grow when there are a significant number of racks. CacheReplication provides the optimal solution, since replicating hot objects in all spine switches eliminates the load imbalance problem. DistCache provides the same performance as CacheReplication and scales out linearly.

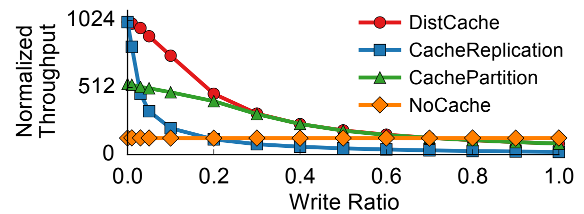
6.3 Cache Coherence

While read-only workloads provide a good benchmark to show the caching benefit, real-world workloads are usually *read-intensive* [5]. Write queries require the two-phase update protocol to ensure cache coherence, which (i) consumes the processing power at storage servers, and (ii) reduces the caching benefit as the cache cannot serve queries to hot objects that are frequently being updated. CacheReplication, while providing the optimal throughput under read-only workloads, suffers from write queries, since a write query to a cached object requires the system to update all spine switches. We use the basic setup as the previous experiment, and vary the write ratio.

Since both the workload skew and the cache size would



(a) Throughput vs. write ratio under Zipf-0.9 and cache size 640.



(b) Throughput vs. write ratio under Zipf-0.99 and cache size 6400.

Figure 10: Cache coherence result.

affect the result, we show two representative scenarios. Figure 10(a) shows the scenario for Zipf-0.9 and cache size 640 (i.e., 10 objects in each cache switch). Figure 10(b) shows the scenario for Zipf-0.99 and cache size 6400 (i.e., 100 objects in each cache switch), which is more skewed and caches more objects than the scenario in Figure 10(a). NoCache is not affected by the write ratio, as it does not cache anything (and our rate limiter for the emulated storage servers assumes same overhead for read and write queries, which is usually the case for small values in in-memory key-value stores [33]). The performance of CacheReplication drops very quickly, and it is highly affected by the workload skew and the cache size, as higher skewness and bigger cache size mean more write queries would invoke the two-phase update protocol. Since DistCache only caches an object once in each layer, it has minimal overhead for cache coherence, and its throughput reduces slowly with the write ratio. The throughputs of the three caching mechanisms eventually become smaller than that of NoCache, since the servers spend extra resources on the cache coherence. Thus, in-network caching should be disabled for write-intensive workloads, which is a general guideline for many caching systems.

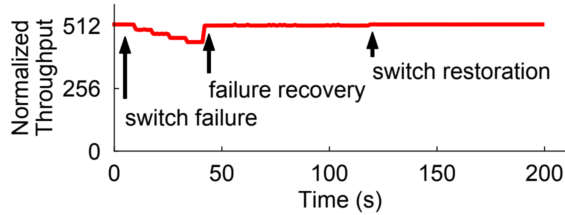


Figure 11: Time series for failure handling.

Switches	Match Entries	Hash Bits	SRAMs	Action Slots
Switch.p4	804	1678	293	503
Spine	149	751	250	98
Leaf (Client)	76	209	91	32
Leaf (Server)	120	721	252	108

Table 1: Hardware resource usage of DistCache.

6.4 Failure Handling

We now evaluate how DistCache handles failures. Figure 11 shows the time serious of this experiment, where x-axis denotes the time and y-axis denotes the system throughput. The system starts with 32 spine switches. We manually fail four spine switches one by one. Since each spine switch provides 1/32 of the total throughput, after we fail four spine switches, the system throughput drops to about 87.5% of its original throughput. Then the controller begins a failure recovery process, by redistributing the partitions of the failed spine switches to other alive spine switches. Since the maximum throughput the system can provide drops to 87.5% due to the four failed switches, the failure recovery would have no impact if all alive spine switches were already saturated. To show the benefit of the failure recovery, we limit the sending rate to half of the maximum throughput. Therefore, after the failure recovery, the throughput can increase to the original one. Finally, we bring the four failed switches back online.

6.5 Hardware Resources

Finally, we measure the resource usage of the switches. The programmable switches we use allow developers to define their own packet formats and design the packet actions by a series of match-action tables. These tables are mapped into different stages in a sequential order, along with dedicated resources (e.g., match entries, hash bits, SRAMs, and action slots) for each stage. DistCache leverages stateful memory to maintain the cached key-value items, and minimizes the resource usage. Table 1 shows the resource usage of the switches with the caching functionality. We show all the three roles, including a spine switch, a leaf switch in a client rack, and a leaf switch in a storage rack. Compared to the baseline Switch.p4, which is a fully functional switch, adding caching only requires a small amount of resources, leaving plenty room for other network functions.

7 Related Work

Distributed storage. Distributed storage systems are widely deployed to power Internet services [1–4]. One trend is to move storage from HDDs and SDDs to DRAMs for high performance [19, 20, 34, 35]. Recent work has explored both hardware solutions [36–47] and software optimizations [33, 48–52]. Most of these techniques focus on the single-node performance and are orthogonal to DistCache, as DistCache focuses on the entire system spanning many clusters.

Load balancing. Achieving load balancing is critical to scale out distributed storage. Basic data replication techniques [25, 53] unnecessarily waste storage capacity under skewed workloads. Selective replication and data migration techniques [54–56], while reducing storage overhead, increase system complexity and performance overhead for query routing and data consistency. EC-Cache [31] leverages erasure coding, but since it requires to split an object into multiple chunks, it is more suitable for large objects in data-intensive applications. Caching is an effective alternative for load balancing [9–11]. DistCache pushes the caching idea further by introducing a distributed caching mechanism to provide load balance for large-scale storage systems.

In-network computing. Emerging programmable network devices enable many new in-network applications. IncBricks [57] uses NPUs as a key-value cache. It does not focus on load balancing. NetPaxos [58, 59] presents a solution to implement Paxos on switches. SpecPaxos [60] and NOPaxos [61] use switches to order messages to improve consensus protocols. Eris [62] moves concurrency control to switches to improve distributed transactions.

8 Conclusion

We present DistCache, a new distributed caching mechanism for large-scale storage systems. DistCache leverages independent hash functions for cache allocation and the power-of-two-choices for query routing, to enable a "one big cache" abstraction. We show that combining these two techniques provides provable load balancing that can be applied to various scenarios. We demonstrate the benefits of DistCache by the design, implementation and evaluation of the use case for emerging switch-based caching.

Acknowledgments We thank our shepherd Ken Salem and the reviewers for their valuable feedback. Liu, Braverman, and Jin are supported in part by NSF grants CNS-1813487, CRII-1755646 and CAREER 1652257, Facebook Communications & Networking Research Award, Cisco Faculty Award, ONR Award N00014-18-1-2364, DARPA/ARO Award W911NF1820267, and Amazon AWS Cloud Credits for Research Program. Ion Stoica is supported in part by NSF CISE Expeditions Award CCF-1730628, and gifts from Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk and VMware.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM SOSP*, October 2003.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SOSP*, October 2007.

[3] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, *et al.*, "Finding a needle in Haystack: Facebook's photo storage.," in *USENIX OSDI*, October 2010.

[4] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *USENIX NSDI*, April 2013.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS*, June 2012.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *ACM Symposium on Cloud Computing*, June 2010.

[7] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse, "Characterizing load imbalance in real-world networked caches," in *ACM SIGCOMM HotNets Workshop*, October 2014.

[8] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites," in *WWW*, May 2002.

[9] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, "Small cache, big effect: Provable load balancing for randomly partitioned cluster services," in *ACM Symposium on Cloud Computing*, October 2011.

[10] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with SwitchKV," in *USENIX NSDI*, March 2016.

[11] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing key-value stores with fast in-network caching," in *ACM SOSP*, October 2017.

[12] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, October 2001.

[13] S. P. Vadhan *et al.*, "Pseudorandomness," *Foundations and Trends® in Theoretical Computer Science*, vol. 7, no. 1–3, pp. 1–336, 2012.

[14] J. Nešetril, "Graph theory and combinatorics," *Lecture Notes, Fields Institute*, pp. 11–12, 2011.

[15] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "Distcache: Provable load balancing for large-scale storage systems with distributed caching," *CoRR*, vol. abs/1901.08200, 2017.

[16] S. Foss and N. Chernova, "On the stability of a partially accessible multi-station queue with state-dependent routing," *Queueing Systems*, 1998.

[17] R. D. Foley and D. R. McDonald, "Join the shortest queue: stability and exact asymptotics," *Annals of Applied Probability*, pp. 569–607, 2001.

[18] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, May 1998.

[19] "Redis data structure store." https://redis.io/.

[20] "Memcached key-value store." https://memcached.org/.

[21] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "CONGA: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM*, August 2014.

[22] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *ACM SOSR*, March 2016.

[23] "Barefoot Tofino." https://www.barefootnetworks.com/technology/#tofino.

[24] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[25] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *ACM Symposium on Theory of Computing*, May 1997.

[26] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *ACM SOSP*, October 2001.

[27] 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 CCR*, July 2014.

[28] "Barefoot P4 Studio." https://www.barefootnetworks.com/products/brief-p4-studio/.

[29] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-min Sketch and Its Applications," *J. Algorithms*, 2005.

[30] "Hiredis: Redis library." https://redis.io/.

[31] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding," in *USENIX OSDI*, November 2016.

[32] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *ACM SIGMOD*, May 1994.

[33] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *USENIX NSDI*, April 2014.

[34] "Amazon DynamoDB accelerator (DAX)." https://aws.amazon.com/dynamodb/dax/.

[35] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud storage system," *ACM Transactions on Computer Systems*, August 2015.

[36] M. Blott, K. Karras, L. Liu, K. A. Vissers, J. Bär, and Z. István, "Achieving 10Gbps line-rate key-value stores with FPGAs," in *USENIX HotCloud Workshop*, June 2013.

[37] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA Memcached appliance," in *ACM/SIGDA FPGA*, February 2013.

[38] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing SoC accelerators for Memcached," in *ACM/IEEE ISCA*, June 2013.

[39] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-performance in-memory key-value store with programmable NIC," in *ACM SOSP*, October 2017.

[40] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *ACM SIGCOMM*, August 2014.

[41] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *USENIX ATC*, June 2016.

[42] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *USENIX NSDI*, April 2014.

[43] S. Li, K. Lim, P. Faraboschi, J. Chang, P. Ranganathan, and N. P. Jouppi, "System-level integrated server architectures for scale-out datacenters," in *IEEE/ACM MICRO*, December 2011.

[44] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, *et al.*, "Scale-out processors," in *ACM/IEEE ISCA*, June 2012.

[45] A. Gutierrez, M. Cieslak, B. Giridhar, R. G. Dreslinski, L. Ceze, and T. Mudge, "Integrated 3D-stacked server designs for increasing physical density of key-value stores," in *ACM ASPLOS*, March 2014.

[46] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in *ACM ASPLOS*, March 2014.

[47] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *ACM/IEEE ISCA*, June 2015.

[48] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A fast array of wimpy nodes," in *ACM SOSP*, October 2009.

[49] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *EuroSys*, April 2014.

[50] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent memcache with dumber caching and smarter hashing," in *USENIX NSDI*, April 2013.

[51] V. Vasudevan, M. Kaminsky, and D. G. Andersen, "Using vector interfaces to deliver millions of IOPS from a networked key-value storage server," in *ACM Symposium on Cloud Computing*, October 2012.

[52] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *ACM SOSP*, October 2011.

[53] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *ACM SOSP*, October 2001.

[54] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," in *EuroSys*, April 2015.

[55] M. Klems, A. Silberstein, J. Chen, M. Mortazavi, S. A. Albert, P. Narayan, A. Tumbde, and B. Cooper, "The Yahoo!: Cloud datastore load balancer," in *CloudDB*, October 2012.

[56] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, "E-Store: Fine-grained elastic partitioning for distributed transaction processing systems," in *VLDB*, November 2014.

[57] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "IncBricks: Toward in-network computation with an in-network cache," in *ACM ASPLOS*, April 2017.

[58] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "NetPaxos: Consensus at network speed," in *ACM SOSR*, June 2015.

[59] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *SIGCOMM CCR*, April 2016.

[60] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, "Designing distributed systems using approximate synchrony in data center networks," in *USENIX NSDI*, May 2015.

[61] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, "Just say NO to Paxos overhead: Replacing consensus with network ordering," in *USENIX OSDI*, November 2016.

[62] J. Li, E. Michael, and D. R. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *ACM SOSP*, October 2017.

GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction

Ting Yao[1,2], Jiguang Wan[1*], Ping Huang[2], Yiwen Zhang[1], Zhiwen Liu[1], Changsheng Xie[1], and Xubin He[2]

[1]WNLO, School of Computer Science and Technology,
Huazhong University of Science and Technology, China
[1]Key Laboratory of Information Storage System, Ministry of Education of China
[2]Temple University, USA

Abstract

Host-managed shingled magnetic recording drives (HM-SMR) give a capacity advantage to harness the explosive growth of data. Applications where data is sequentially written and randomly read, such as key-value stores based on Log-Structured Merge Trees (LSM-trees), make the HM-SMR an ideal solution due to its capacity, predictable performance, and economical cost. However, building an LSM-tree based KV store on HM-SMR drives presents severe challenges in maintaining the performance and space efficiency due to the redundant cleaning processes for applications and storage devices (i.e., compaction and garbage collections). To eliminate the overhead of on-disk garbage collections (GC) and improve compaction efficiency, this paper presents *GearDB*, a GC-free KV store tailored for HM-SMR drives. GearDB proposes three new techniques: a new on-disk data layout, compaction windows, and a novel gear compaction algorithm. We implement and evaluate GearDB with LevelDB on a real HM-SMR drive. Our extensive experiments have shown that GearDB achieves both good performance and space efficiency, i.e., on average 1.71× faster than LevelDB in random write with a space efficiency of 89.9%.

1 Introduction

Shingled Magnetic Recording (SMR) [12] is a core technology driving disk areal density increases. With millions of SMR drives shipped by drive vendors [32, 17], SMR presents a compelling solution to the big data challenge in an era of explosive data growth. SMR achieves higher areal density within the same physical footprint as conventional hard disks by overlapping tracks, like shingles on a roof. SMR drives are divided into large multi-megabyte zones that must be written sequentially. Reads can be processed precisely from any uncovered portion of tracks, but random writes risk corrupting data on overlapped tracks, imposing random write complexities [10, 12, 3]. The sequential write restriction makes log-structured writes to shingled zones a common practice [38, 32, 33], creating a potential garbage collection (GC) problem. GC reclaims disk space by migrating valid data to produce empty zones for new writes. The data migration overhead of GC severely degrades system performance.

*Corresponding author. Email: jgwan@hust.edu.cn

Among the three SMR types (i.e., drive-managed, host-managed, and host-aware), HM-SMR presents a preferred option due to its capacity, predictable performance, and low total cost of ownership (TCO). HM-SMR offers an ideal choice in data center environments that demand predictable performance and control of how data is handled [15], especially for domains where applications commonly write data sequentially and read data randomly, such as social media, cloud storage, online backup, life sciences as well as media and entertainment [15]. The key-value data store based on Log-Structured Merge trees (LSM-trees) [37] inherently creates that access pattern due to its batched sequential writes and thus becomes a desirable target application for HM-SMR.

LSM-tree based KV stores, such as Cassandra [28], RocksDB [9], LevelDB [11], and BigTable [5], have become the state-of-art persistent KV stores. They achieve high write throughput and fast range queries on hard disk drives and optimize for write-intensive workloads. The increasing demand on KV stores' capacities makes adopting HM-SMR drives an economical choice [24]. Researchers from both academia and industry have been attempting to build key-value data stores on HM-SMR drives by modifying applications to take advantage of the high capacity and predictable performance of HM-SMR, such as Kinetic from Seagate [41], SMR based key-value store from Huawei [31], SMORE form Netapp [32], and others [47, 38, 48].

However, building an LSM-tree based KV store on HM-SMR drives comes with a serious challenge: the redundant cleaning processes on both LSM-trees and HM-SMR drives harm performance. In an LSM-tree, the compaction processes are conducted throughout the lifetime to clean invalid data and keep data sorted in

multiple levels. In an HM-SMR drive, the zones with a log-structured data layout are fragmented as a result of data being invalidated by applications (e.g., compactions from LSM-trees). Therefore, garbage collection must be executed to maintain sizeable free disk space for writing new data. Existing applications on HM-SMR drives either leave the garbage collection problem unsolved [33, 22] or use a simple greedy strategy to migrate live data from partially empty zones [32]. Redundant cleaning processes, the garbage collection for storage devices in particular, degrade system performance dramatically. To demonstrate the impact of on-disk garbage collection, we implement a cost-benefit and a greedy garbage collection strategy similar to the free space management in log-structured files system and SSDs [40, 2]. Evaluation results in Section 2.3 indicate that garbage collection not only causes expensive overheads on system latency but also hurts the space utilization of HM-SMR drives. Conventional KV stores on HM-SMR drives face a dilemma: either obtain high space efficiency with poor performance or take good performance with poor space utilization. The space utilization is defined as the ratio between the on-disk valid data volume and the allocated disk space.

To obtain both good performance and high space efficiency in building an LSM-tree based KV store on HM-SMR drives, we propose GearDB with three novel design strategies. **First**, we propose a new on-disk data layout, where a zone only stores SSTables from the same level of an LSM-tree, contrary to arbitrarily logging SSTables of multiple levels to a zone as with the conventional log layout. In this way, SSTables in a zone share the same compaction frequency, remedying dispersed fragments on disks. The new on-disk data layout manages SSTables to align with the underlying SMR zones at the application level. **Second**, we design a *compaction window* for each level of an LSM-tree, which is composed of $1/k$ zones of that level. Compaction windows help to limit compactions and the corresponding fragments to a confined region of the disk space. **Third**, based on the new data layout and compaction windows, we propose a new compaction algorithm called *Gear Compaction*. Gear compaction proceeds in compaction windows and descends level by level only if the newly generated data overlaps the compaction window of the next level. Gear compaction not only improves the compaction efficiency but also empties compaction windows automatically so that SMR zones can be reused without garbage collection. By applying these design techniques, we implement GearDB based on LevelDB, a state-of-art LSM-tree based KV store. Evaluating GearDB and LevelDB on a real HM-SMR drive, test results demonstrate that GearDB is $1.71\times$ faster in random writes compared to LevelDB, and has an efficient space utilization of 89.9% in a bimodal distribution (i.e., zones are either nearly empty or nearly full).

2 Background and Motivation

In this section, we discuss HM-SMR and LSM-trees, as well as challenges and our motivation in building LSM-tree based KV stores on HM-SMR drives.

2.1 Shingled Magnetic Recording (SMR)

Shingled Magnetic Recording (SMR) techniques provide a substantial increase in disk areal density by overlapping adjacent tracks. SMR drives allow fast sequential writes and reads like any conventional HDDs, but have destructive random writes. SMR drives are classified into three types based on where the random write complexity is handled: in the drive, in the host, or co-operatively by both [17]. Drive-managed SMR (DM-SMR) implements a translation layer in firmware to accommodate both sequential and random writes. It acts as a drop-in replacement of existing HDDs but suffers highly unpredictable and inferior performance [1, 4]. Host-managed SMR (HM-SMR) requires host-software modifications to reap its advantages [33]. It accommodates only sequential writes and delivers predictable performance by exposing internal drive states. Host-aware SMR (HA-SMR) lies somewhere between HM-SMR and DM-SMR. However, it is the most complicated and obtains maximum benefit and predictability when it works as HM-SMR [45]. Research has demonstrated that SMR drives can fulfill modern storage needs without compromising performance [38, 32, 33, 41].

Like many production HA/HM-SMR drives [42, 18, 15], the drive used in this study is divided into 256 MB-sized zones. Each zone accommodates strict sequential writes by maintaining a write pointer to resume the subsequent write. A guard region separates two adjacent zones. A zone without valid data can be reused as an empty zone via resetting the zone's write pointer to the first block of that zone. All the intricacies of HM-SMR are exposed to the software by a new command set, the T10 Zone Block Commands [19]. To comply with the SMR sequential write restrictions, applications or operating systems are required to write data in a log-structured fashion [38, 32, 33, 4, 27]. However, the log-structured layout imposes additional overhead in the form of garbage collection (GC). GC blocks foreground requests and degrades system performance due to live data migration.

2.2 LSM-trees and Compaction

Log-Structured Merge trees (LSM-trees) [37] exploit the high-sequential write bandwidth of storage devices by writing sequentially [39]. Index changes are first deferred and buffered in memory, then cascaded to disks level by level via merging and sorting. The Stepped-Merge technique is a variant of LSM-trees [21], which changes a single index into k indexes at each level to reduce the cost of inserts.

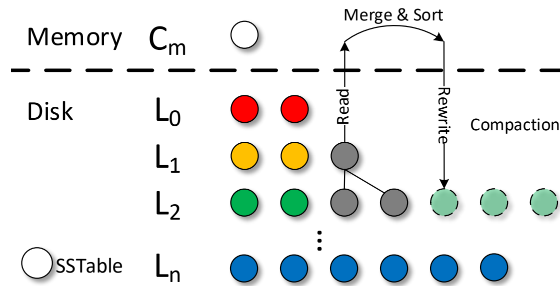


Figure 1: **A compaction process in an LSM-tree based KV store.** *This figure shows the LSM-tree data structure, which is composed of a memory component and a multi-leveled disk component. Compaction is conducted level by level to merge SSTables from the lower to higher levels.*

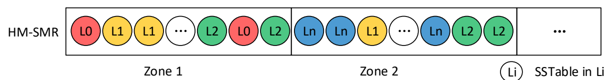


Figure 2: **Conventional on-disk data layout with log-structured writes.** *This figure shows that the conventional log write causes SSTables of different levels mixed in the zones on HM-SMR drives.*

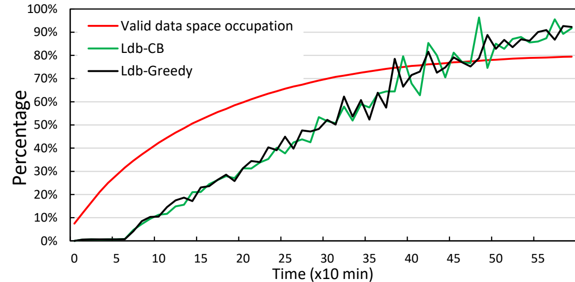


Figure 3: **Cost of garbage collections.** *The green line shows the ratio of valid data volumes to the test disk space; the other two lines show the ratio of time consumption of garbage collections in every ten minutes during the random loading.*

Due to their high update throughput, LSM-trees and their variants have been widely used in KV stores [39, 8, 24, 5, 28]. LevelDB [11] is a popular key-value store based on LSM-trees. In LevelDB, the LSM-tree batches writes in memory first and then flushes batched data to storage as sorted tables (i.e., SSTable) when the memory buffer is full. SSTables on storage devices are sorted and stored in multiple levels and merged from lower levels to higher levels. Level sizes increase exponentially by an amplification factor (e.g., AF=10). The process of merging and cleaning SSTables is called compaction, and it is conducted throughout the lifetime of an LSM-tree to clean invalid/stale KV items and keep data sorted on each level for efficient reads [37, 43]. Figure 1 illustrates the compaction in an LSM-tree data structure (e.g., LevelDB). When the size limit of L_i is reached, a compaction starts merging SSTables from L_i to L_{i+1} and proceeds in the following steps. First, a victim SSTable in L_i is picked in a round-robin manner, along with any SSTables in L_{i+1} whose key range overlaps that of the victim SSTable. Second, these SSTables are fetched into memory, merged and resorted to generate new SSTables. Third, the new SSTables are written back to L_{i+1}. Those stale SSTables, including the victim SSTable and the overlapped SSTables, then become invalid, leaving dispersed garbage data in the disk space.

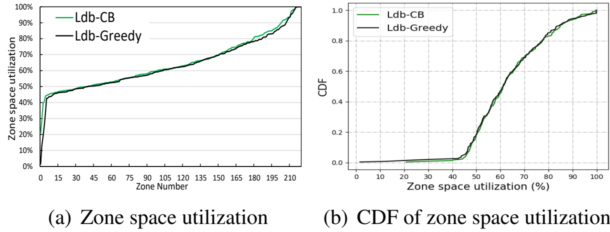
2.3 Motivation

The log-structured write fashion required by HM-SMR drives could lead to excessive disk fragments and therefore necessitates costly garbage collections to deal with them. Specifically, in an LSM-tree based KV store on HM-SMR drives, compaction cleans invalid KV items in LSM-trees but leaves invalid SSTables on the disk. Especially, the arbitrarily sequential writes of the conventional log result in SSTables from multiple levels that have different compaction frequency being mixed in the same zones, as shown in Figure 2. As compaction procedures constantly invalidate SSTables during the lifespan of LSM-trees, the fragments on HM-SMR drives become severely dispersed, necessitating many GCs. Due to the high write amplification of LSM-trees [29] (i.e., more than 12×) and the huge volume of dispersed fragments caused by compaction, passive GCs become inevitable. Passive GCs are triggered when the free disk space is under a threshold (i.e., 20% [36]) and clean zone space by migrating valid data from zones to zones.

To demonstrate the problems of garbage collections in the LSM-tree based KV store on HM-SMR drives, we implement LevelDB [11], a state-of-art LSM-tree based KV store, on a real HM-SMR drive using log-structured writes to zones. We implement both greedy and cost-benefit GC strategies [40, 2, 32] to manage the free space on HM-SMR drives. The greedy GC cleans the zone with the most invalid data by migrating its valid data to other zones. Cost-benefit GC selects a zone by considering the age and the space utilization of that zone (u) according to Equation 1 [40]. We define the age of a zone as the sum of SSTables' level ($\sum_0^n L_i$), based on the observation that SSTables in a higher level live longer and have a lower compaction frequency, where n is the number of SSTables in the zone and L_i is the level of an SSTable. The cost includes reading a zone and writing back u valid data.

$$\frac{benefit}{cost} = \frac{FreeSpaceGain \times ZoneAge}{cost} = \frac{(1-u) \times \sum_0^n L_i}{1+u} \quad (1)$$

With the parameters described in Section 5, we randomly load 20 million KV items to an HM-SMR drive using only 70



(a) Zone space utilization (b) CDF of zone space utilization

Figure 4: **Zone space utilization.** *Figure a shows the zone space utilization of each zone after random loading the first 40 GB database, plotting in the order of increasing space utilization. Figure b shows the CDF of the zone space utilization.*

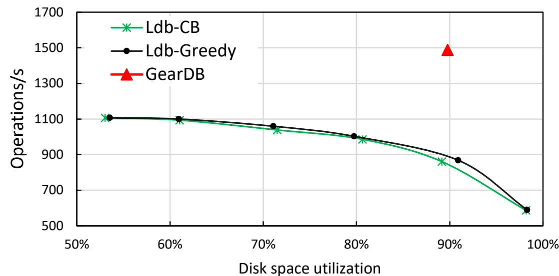


Figure 5: **The throughput vs. different space utilization.** *This figure shows that in the conventional KV store on an HM-SMR drive, the system performance decreases with the space utilization. It is unable to get both good performance and space efficiency simultaneously.*

GB disk space comprised of 280 shingled zones. The valid data volume of the workload is about 54 GB, approximating 80% of the disk space, due to duplicated and deleted KV entries. Through this experiment, we have made the following observations. First, we calculate the valid data volume and the GC time every ten minutes during the load process. As shown in Figure 3, the time consumption of GC grows with the valid data volume. When valid data grows to about 70% of the disk space, more than half of the time in that ten minutes is spent to perform GC. GC accounts for more than 80% of the execution time when valid data reaches 76% of the disk space. The test results demonstrate that garbage collection takes a substantial proportion of the total execution time, downgrading the system performance dramatically. Because SSTables from different levels in a zone are mixed, multiple zones show similar age in cost-benefit GC policy. The similar zone ages make greedy and cost-benefit policies present almost the same performance, as they both prefer reclaiming zones with the most invalid data.

Second, we record the space utilization of each occupied zone on the HM-SMR drive after loading 10 million KV items. Figure 4 (a) shows the percentage of valid data in each zone (in a sorted way), and Figure 4 (b) shows the cumulative distribution function (CDF) of the zone space utilization. Both greedy and cost-benefit have an unsatisfactory average space efficiency of 60%. More specifically, 85% zones have a space utilization ranging from 45% to 80%. We contend that this space utilization distribution results in the significant amount of time spent in doing GC, as discussed above. The more live data in zones that is migrated, the more disk bandwidth is needed for cleaning and not available for writing new data. A better and more friendly space utilization would be a bimodal distribution, where most of the zones are nearly full, a few are empty, and the cleaner can always work with the empty zones, eliminating the overhead of GC, i.e., valid data migration. In this way, we can achieve both high disk space utilization and eliminate on-disk garbage collection overheads. This forms the key objective of our GearDB design, as discussed in the next section.

Third, by changing the threshold of GC (from 100% to 50%) on the 110 GB restricted disk capacity, we test 6 groups of 80 GB random writes to show the performance variations with disk space utilization. **The disk space utilization, or space efficiency, is defined as the ratio of the on-disk valid data volume to the allocated disk space.** As shown in Figure 5, system performance decreases with space utilization. Running on an HM-SMR drive, LevelDB faces a dilemma where it only delivers a compromised trade-off between performance and space utilization. Our goal in designing GearDB is to achieve higher performance and better space efficiency simultaneously. The red triangle mark in Figure 5 denotes the measured performance and space efficiency of GearDB, i.e., 89.9% space efficiency and 1489 random load IOPS.

In summary, with log-structured writes, existing KV stores on HM-SMR suffer from redundant cleaning processes in both LSM-trees (i.e., compaction) and HM-SMR drives (i.e., garbage collection). The expensive GC degrades system performance, decreases space utilization, and creates a suboptimal trade-off between performance and space efficiency.

3 GearDB Design

In this section we present GearDB and three key techniques to eliminate the garbage collection and improve compaction efficiency. GearDB is an LSM-tree based KV store that achieves both high performance and space efficiency on an HM-SMR drive. Figure 6 shows the overall architecture of GearDB's design strategies. First, we propose a new on-disk data layout that provides application-specific data management for HM-SMR drives, where a zone only serves SSTables from one level to prevent data in different levels from being mixed and causing dispersed fragments. Second, based on the new on-disk data layout, we design a compaction window for each LSM-trees level. Compactions only proceed within compaction windows, which restricts frag-

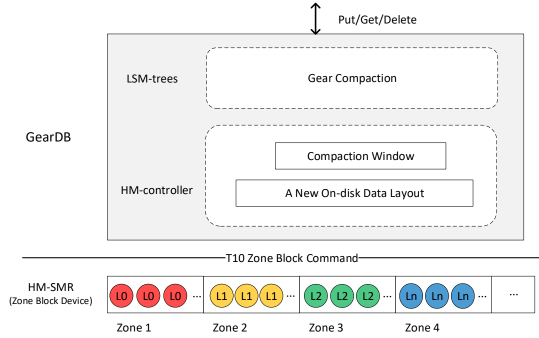


Figure 6: **The architecture of GearDB.** *This figure shows the overall structure of GearDB with three design strategies. GearDB accesses the HM-SMR drive directly via the T10 zone block command. For the data layout on an HM-SMR drive, SSTables form the same level are located in integral zones, and each zone only serves SSTables from the same level.* Ⓛi *represents the SSTable from* L_i.

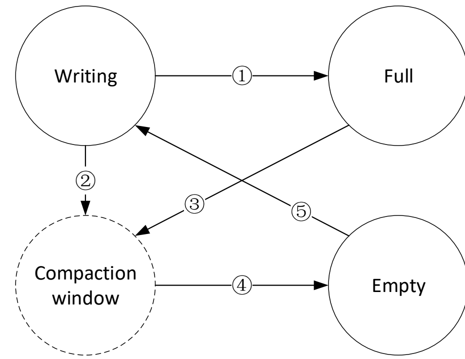


Figure 7: **Zone state transitions in GearDB.** *In GearDB, multiple zones are allocated to each level according to the data size of that level. These zones can be in three states during their lifetime, namely writing zone, full zone, and empty zone. Zones, including full zones and writing zones, rotate to construct a compaction window.*

ments in compaction windows. Third, we propose a novel compaction algorithm, called *gear compaction*, based on the new data layout and compaction windows. Gear compaction divides the merged data of each compaction into three portions and further compacts with the overlapped data in the compaction window of the next level. Gear compactions automatically empty SMR zones in compaction windows by invalidating all SSTables, so that zones can be reused without the need to garbage collection. We elaborate on the three strategies in the following subsections.

3.1 A New On-disk Data Layout

As discussed in Section 2.3, data fragments on HM-SMR drives are widely dispersed due to log-structured writes and compactions, which causes SSTables from different levels to be mixed within zones (Figure 2). To alleviate this problem, we propose a new data layout to manage HM-SMR drives in GearDB.

The key idea of the new data layout is that each zone only serves SSTables from one level, as shown in Figure 6. We dynamically assign zones to different levels of an LSM-tree. Initially, each level in use is attached to one zone. As the data volume of a level increases, additional zones are allocated to that level. Once a zone is assigned to Level L_i, it can only store sequentially written SSTables from L_i until it is released as an empty zone by GearDB. When an LSM-tree reaches a balanced state, each level is composed of multiple zones according to its size limit. Among the zones of each level, only one zone accepts incoming writes, named a *writing zone*. Sequential writes in each zone strictly respect the shingling constraints of HM-SMR drives.

Since SSTables in a zone belong to the same level, they share the same compaction frequency (or the same hotness). This data layout results in less fragmented disk space and offers convenience for the following design strategies, which potentially leads to the desired bimodal distribution and thus allows us to achieve high system performance at low cost. Additionally, sequential read performance is improved due to better spatial locality.

3.2 Compaction Windows

With our new data layout, each level in an LSM-tree has multiple zones corresponding to its data volume or size limit. To further address the dispersed fragments on HM-SMR drives based on the new data layout, we propose a compaction window for each level. A compaction window (CW) for a level is composed of a group of zones belonging to the level, which is used to limit compactions and fragments.

Specifically, GearDB presets a compaction window for each level of the LSM-tree. To construct a compaction window, a certain number of zones are picked from the zones belonging to that level in a rotating fashion. The compaction window size (S_{cwi}) of level L_i is given by Equation 2, where the compaction window size of each level is $1/k$ of the level size limit (L_{Li}). Hence, the compaction window size increases by the same amplification factor as the size for each level. By default, the compaction window size is $\frac{1}{4}$ of the corresponding level size. Note that the compaction window of L_0 and L_1 comprises the entire level since these two levels only take one zone in our study.

$$S_{cwi} = \frac{1}{k} \times L_{Li} \quad (1 \leq k \leq AF) \tag{2}$$

Compaction windows are not designed to directly improve the system performance. However, by limiting compactions within compaction windows, the corresponding fragments

are restricted to compaction windows instead of spanning over the entire disk space. Therefore, system performance benefits from gear compactions. Since a zone full of invalid data can be reused as an empty zone without data migration, compaction windows that filled with invalid SSTables can be released as a group of empty zones to serve future write requests. When zones of a compaction window are released, another group of zones of that level is selected to form a new compaction window. Different zones in a level rotate to constitute the compaction window, guaranteeing every SSTable gets involved in compactions evenly.

To facilitate the management of underlying SMR zones in GearDB, we divide zones into three states, namely writing zone, full zone, and empty zone. Each level only maintains a writing zone for sequentially appending newly arrived SSTables. Figure 7 shows the diagram of zone state transitions. ① A writing zone becomes a full zone once it is filled. ② ③ A writing zone or full zone can be added into a compaction window by rotation. ④ When all SSTables of a compaction window have been invalidated by gear compactions, the zones become empty and ⑤ ready to serve write requests without incurring device-level garbage collection.

3.3 Gear Compaction

Based on our new data layout and compaction windows, we develop a new compaction algorithm in this section. Gear compaction aims to automatically clean compaction windows during compactions and thus eliminate costly and redundant garbage collections.

Gear Compaction Algorithm. A gear compaction process starts by compacting L_0 and L_1, called active compaction. Active compaction triggers subsequent passive compactions, and compactions progress from lower levels to higher levels. For a conventional compaction between L_i and L_{i+1} in LevelDB, the merge-sorted data are directly written back to the next level (i.e., L_{i+1}). However, for a gear compaction between L_i and L_{i+1}, the merge-sorted data is divided into three parts according to its key range, including: out of L_{i+2}'s compaction window, out of L_{i+2}'s key range, and within L_{i+2}'s compaction window. These three parts of the merged data do not stay in memory. Instead, they are respectively 1) written to L_{i+1}, 2) dumped to L_{i+2}, or 3) processed to passive compactions (i.e., compacted with overlapped SSTables in the CW of L_{i+2}). The dump operation (i.e., step 2) helps to reduce the further write amplification of writing the data to L_{i+1} and dumping it to L_{i+2}. To avoid data being compacted to the highest level directly, L_{i+2} can only join the gear compaction if L_{i+1} reaches its size limit and L_{i+2} reaches the size of its compaction window. As a result, GearDB maintains the temporal locality of LSM-trees, where newer data resides in lower levels.

Figure 8 illustrates the gear compaction process. **Step 1**, the active compaction is performed between L_0 and L_1,

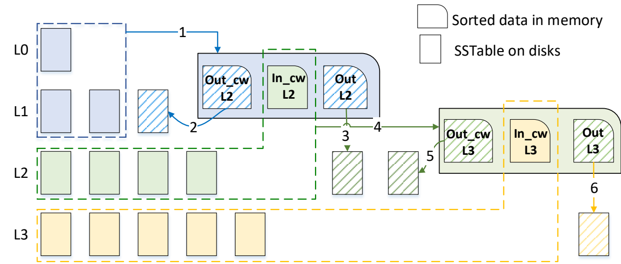


Figure 8: **Process of gear compaction.** *The active compaction of L_0 and L_1 drives passive compactions in higher level. The resultant data of each compaction is divided into three parts according to its key range, including out of L_i's compaction window (Out_cw L_i), in L_i's compaction window (In_cw L_i), and out of L_i's key range (Out L_i).*

and the resultant data in memory is divided into three parts according to their key range, i.e., out of L_2's compaction window, out of the next level, and within L_2's compaction window. **Step 2**, the data whose key range overlaps SSTables that is out of L_2's compaction window is written back to L_1. **Step 3**, the data whose key range does not overlap L_2's SSTables is dumped to L_2, avoiding further compaction and the associated write amplification. **Step 4**, the data whose key range overlaps with SSTables in L_2's compaction window remains in memory for further passive compaction with the overlapped SSTables in L_2' s compaction window. This gear compaction process proceeds recursively in compaction windows, level by level, until either the compaction reaches the highest level or the regenerated data does not overlap the compaction window of the next level. Thus, gear compaction only proceeds within compaction windows and therefore invalid SSTables only appear in compaction windows.

The gear compaction process is described in Algorithm 1. Lines 7-17 illustrate the key range division (detailed later in "**sorted data division**"). Active compaction starts from L_0 and L_1, and passive compaction continues level by level until the merge and sort results of L_i and L_{i+1} do not overlap L_{i+2}'s compaction window (Line 19 and 24). In addition, if the data volume written to L_{i+2} is less than the size of an SSTable (e.g., 4 MB), we write it back to L_{i+1} together with other data written to L_{i+1}. In this way, the size of each SSTable is kept at about 4 MB to ensure no small SSTable increases the overhead of metadata management.

Sorted data division. To divide the sorted data during gear compaction (e.g., L_i and L_{i+1}) into the above mentioned three categories, GearDB needs to compare the key range of the sorted data with the key ranges of SSTables in L_{i+2}. As in LevelDB, SSTables within a level do not overlap in GearDB. However, key ranges of some SSTables might not be successive. Key range gaps between SSTables complicate the division of the sorted data, and we need to compare the sorted datas keys with individual SSTables. Excessive com-

ALGORITHM 1: Gear Compaction Algorithm

```
    Input: V_i: victim SSTable in L_i
 1  do
 2      DoGearComp ← false;
 3      O_{i+1} ← GetOverlaps (V_i); /*O_{i+1}: overlapped SSTables in
        L_{i+1}'s compaction window*/
 4      result ← merge-sort(V_i, O_{i+1});
 5      iter.key ← MakeInputIterator(result);
 6      for iter.first to iter.end do
 7          if key In_CW L_{i+2} then
 8              write to buffer; /*wait in memory for the passive
                compaction*/
 9          else
10              if key Out_CW L_{i+2} then
11                  write to L_{i+1};
12              else
13                  if key Out L_{i+2} then
14                      write to L_{i+2};
15                  end
16              end
17          end
18      end
19      if buffer != Null then
20          i++;
21          V_i ←GetVictims(buffer);
22          DoGearComp ← true;
23      end
24  while DoGearComp == true;
```

parisons can slow down the division and increase the cost of gear compaction and metadata management. To remedy this problem, we divide each level into large granularity key ranges. Specifically, for SSTables in CW and out of CW respectively, if the key range gap between SSTables does not overlap with other SSTables in that level, we combine the key ranges into a large consecutive range. As a result, the sorted data only needs to compare with the minimum and maximum keys of several key ranges to do the division. For example, suppose the compaction window of L_{i+2} has two SSTables with respective key ranges of $a-b$ and $c-d$. We check other SSTables in L_{i+2} to find if any SSTable overlaps the key range $b-c$. If not, we amend the key range of L_{i+2}'s compaction window as $a-d$ to reduce the key range comparison during division.

How compaction windows are reclaimed. As discussed above, gear compactions only proceed within compaction windows. Since a compaction window filled with invalid data can be simply released as empty zones, compaction windows are reclaimed automatically by gear compactions. As a result, redundant garbage collection that requires valid data migration is avoided. To invalidate all SSTables in the CW of L_{i+1}, the SSTables in L_i whose key ranges overlap with

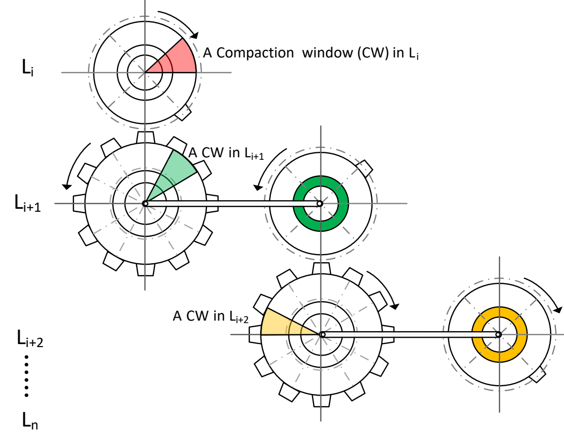


Figure 9: **Compaction windows are reclaimed in a gear fashion.** *The red, green, and yellow sectors represent the compaction windows of L_i, L_{i+1}, and L_{i+2}. Compaction windows are reclaimed by compaction like a group of gears. Reclaim k compaction windows (CW) in L_i mimics a full round moving of a gear, which leads to one move in the driven gear, that is cleaning one compaction window in L_{i+1}, and so on.*

L_{i+1}'s CW must be involved in gear compactions. Once all zones of L_i rotationally join the compaction window, we get these SSTables in L_i. In this fashion, when all compaction windows in L_i are reclaimed, the compaction window of L_{i+1} is reclaimed. As shown in Figure 9, when gear compactions clean k compaction windows in L_i, one compaction window in L_{i+1} is cleaned correspondingly; when gear compactions clean k compaction windows in L_{i+1}, one compaction window in L_{i+2} is cleaned; and so on. This process of releasing compaction windows works like a group of gears, where a complete rotation (i.e., k steps) in a driving gear (i.e., L_i) triggers one move in a driven gear (i.e., L_{i+1}), which gives our name as "gear compaction."

In summary, GearDB maintains the balance of LSM-trees by keeping the amplification factor of adjacent levels unchanged, keeping SSTables sorted and un-overlapped in each level, and rotating the compaction window at each level. The benefits of gear compaction include: 1) compactions and fragments are limited to the compaction window of each level; 2) compaction windows are reclaimed automatically during gear compactions, thereby eliminating the expensive on-disk garbage collections since compaction windows filled with invalid SSTables can be reused as free space; and 3) gear compaction compacts SSTables to a higher level with fewer reads and writes and no additional overhead.

4 Implementation

To verify the efficiency of GearDB's design strategies, we implement GearDB based on LevelDB 1.19 [11], a state-

of-art LSM-tree based KV store from Google. We use the libzbc [16] interface to access a 13 TB HM-SMR drive from Seagate. Libzbc allows applications to implement direct accesses to HM-SMR drives via T10/T13 ZBC/ZAC command set [20, 19], facilitating Linux application development.

As shown in Figure 6, GearDB maintains a standard interface for users, including GET/PUT/DELETE. The gear compaction is implemented on LSM-trees by modifying the conventional compaction processes of LevelDB. At the lowest level, we implement an HM-SMR controller to: 1) write sequentially in zones and manage per-zone write pointers using the new interface provided by libzbc; 2) map SSTables to dedicate zones and map zones to specific levels; and 3) manage the compaction window of each level. The mapping relationship is maintained by: 1) a Lbdfile structure denoting the indirection map of an SSTable and its zone location; 2) a zone_info structure recording all SSTables of a zone; and 3) a zone_info_list[L_i] structure containing all zones of Level L_i. Ldbfiles maintain the metadata of each SSTable. The size of the Ldbfile dominates the size of the metadata in the HM-SMR controller, and the other two structures only link Ldbfiles and zones with pointers (8 bytes). These data structures consume a negligible portion of memory, e.g., for an 80GB database, the overall metadata of the HM-SMR controller is less than 4 MB.

To keep metadata consistent, a conventional zone is allocated on HM-SMR drives to persist the metadata together with the version changes of the database after each compaction. In LevelDB, a manifest file is used to record the initial state of the database and the changes of each compaction. To recover from a system crash, the database starts from the initial state and replays the version changes. GearDB rebuilds the database in the same way.

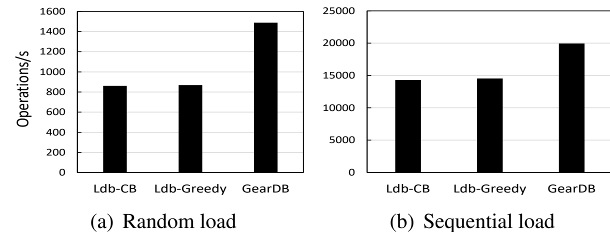
Other implementation details worth mentioning include: 1) for sequential write workloads that incur no compaction, GearDB dumps zones to the higher level by revising the zone_info_list[L_i] to avoid data migration. 2) To accelerate the compaction process in both GearDB and LevelDB, we fetch victim SSTables and overlapped SSTables into memory in the unit of SSTables instead of blocks. More details of the implementation can be found in our open source code with the link provided in Section 7.

5 Evaluation

GearDB is designed to deliver both high performance and space efficiency for building key-value stores on HM-SMR drives. In this section, we conduct extensive experiments to evaluate GearDB by focusing on answering the following questions: 1) what are the performance advantages of GearDB? (Section 5.1); 2) what factors contribute to these performance benefits? (Section 5.2); and 3) What space efficiency can GearDB achieve? (Section 5.3). In addition, we discuss the results of sensitivity studies of CW size, the performance of GearDB vs. SMRDB [38], and the performance of GearDB vs. LevelDB on HDDs(Section 5.4).

Table 1: System configuration for experiments

Linux	64-bit Linux 4.15.0-34-generic
CPU	8 * Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
Memory	32 GB
HM-SMR	13TB Seagate ST13125NM007 Random 4 KB request (IOPS): 163(R) Sequential (MB/s): 180(R), 178(W)



(a) Random load (b) Sequential load

Figure 10: **Load performance.** *GearDB shows its advantage in both random load and sequential load compared to LevelDB.*

We compare GearDB performance against LevelDB (version 1.19) [11] with greedy GC (Ldb-Greedy) and cost-benefit GC (Ldb-CB) policies. Our test environment is listed in Table 1. By default, we use 16-byte keys, 4 KB values, and 4 MB SSTables.

5.1 Performance Evaluation

In this section, we first evaluate the read and write performance of LevelDB and GearDB using the db_bench micro-benchmark released with LevelDB. Then, we evaluate performance using YCSB macro-benchmark suite [7].

5.1.1 Load Performance

We evaluate random load performance by inserting 20 million key-value items (i.e., 80 GB) in a uniformly distributed random order. Since the random load benchmark includes repetitive and deleted keys, the actual valid data volume of the database is around 54 GB. We restrict the capacity of our HM-SMR drive by using only the first 280 shingled zones (i.e., 70 GB). The final valid data takes up 77.14% of the usable disk space. The random write performance is shown in Figure 10 (a). GearDB outperforms Ldb-Greedy and Ldb-CB by 1.71× and 1.73× respectively. The two LevelDB solutions have lower throughput because of the time-consuming compaction and redundant GCs. Compaction in LevelDB produces write amplification and dispersed fragments on disk. Costly garbage collections clean disk space by migrating valid data, thus slowing down the random write performance. GearDB delivers better performance because

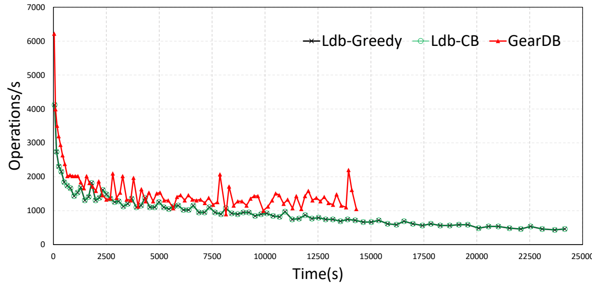


Figure 11: **Detail of random load.** *This figure shows the incremental performance of every 1 GB randomly loaded during the process of loading an 80 GB database. GearDB has higher throughput and a shorten execution time for loading the same sized database compared to LevelDB.*

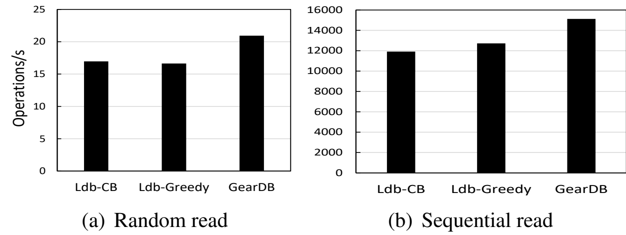
fragments are limited to compaction windows, garbage collections are eliminated by gear compactions, and compaction efficiency is improved. We further investigate detailed reasons for GearDB's performance improvements in Section 5.2.

Figure 11 shows the incremental throughput by recording the performance for every 1 GB of randomly loaded data (i.e., 250k KV entries). We make four observations from this figure. First, GearDB is faster than LevelDB for randomly loading the same volume of data. Second, GearDB achieves higher throughput than LevelDB, and the performance advantage becomes more pronounced as the volume of the database grows. Third, both LevelDB and GearDB's performance decrease with time, because the overhead of compaction and GCs (only LevelDB has GCs) increases with the data volume. Fourth, the performance variation of GearDB comes from the variation of the data volume in gear compactions. On the contrary, LevelDB shows less fluctuation in performance due to the relatively stable data volume involved in each compaction.

Similarly, we evaluate the sequential load performance by inserting 20 million KV items in sequential order. No compactions or garbage collections were incurred for sequential writes. Figure 10(b) shows that GearDB is 1.37× and 1.39× faster than Ldb-Greedy and Ldb-CB respectively. This performance gain is attributed to the more efficient dump strategy of GearDB as presented in Section 4. GearDB dumps SSTables to the next level by simply revising the metadata of zones.

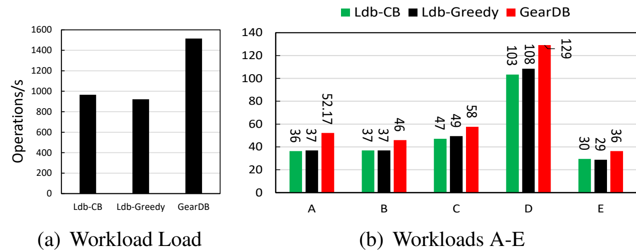
5.1.2 Read Performance

Read performance is evaluated by reading one million key-value items from the randomly loaded database. Figure 12 shows the results. The performance of sequential reads is much better than random reads due to the natural characteristics of disk drives. GearDB gets its performance advantage in both random and sequential reads because it consolidates SSTables of the same level. Our new data layout helps reduce the seek time of searching and locating SSTables by ensuring each zone stores SSTables from just one level.



(a) Random read (b) Sequential read

Figure 12: **Read performance.**



(a) Workload Load (b) Workloads A-E

Figure 13: **Throughput on Macro-benchmarks.** *This figure shows the throughput of three key-value stores on load and other five workloads. In the left figure, the x-axis represents different workloads. The load workload corresponds to constructing an 80 GB database. Workload A is composed with 50% reads and 50% updates; Workload-B has 95% reads and 5% updates; Workload-C includes 100% reads; Workload-D has 95% reads and 5% latest keys insert; Workload-E has 95% range queries and 5% keys insert.*

5.1.3 Macro-benchmark

To evaluate performance with more realistic workloads, we run the YCSB benchmark [7] on GearDB and LevelDB. The YCSB benchmark is an industry standard macro-benchmark suite delivered by Yahoo!. Figure 13 shows the results of the macro-benchmark in load and five other representative workloads. GearDB is 1.56× and 1.64× faster than Ldb-CB and Ldb-Greedy on the load workload for the same reasons discussed in Section 5.1.1. Workloads A-E are evaluated based on the randomly loaded database. The performance gains of GearDB under workloads A-E are 1.44×, 1.24×, 1.22×, 1.25×, and 1.23× compared to LevelDB, which are consistent with the results of micro-benchmarks.

5.2 Performance Gain Analysis

In this section, we investigate GearDB's performance improvement when compared to LevelDB.

Operation Time Breakdown. To figure out the advan-

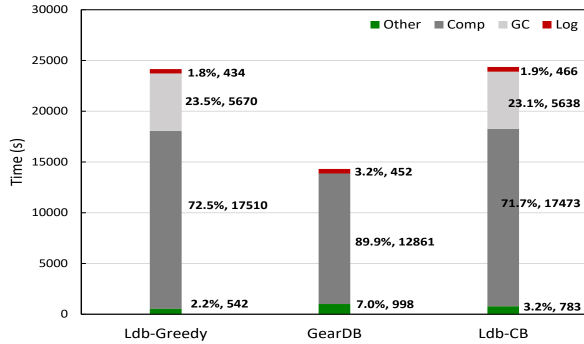


Figure 14: **Random load time breakdown.** *This figure shows the time spent (the y-axis) on different operations when we randomly load an 80 GB database. The numbers next to each bar show their time consumption and the ratio to the overall run time. For LevelDB, compaction and garbage collections take the most significant percentage of the overall runtime. GearDB eliminates the garbage collections and improves compaction efficiency.*

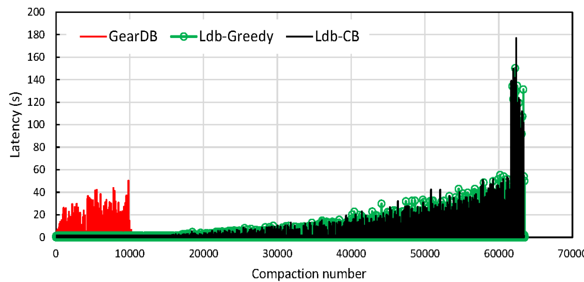


Figure 15: **Compaction analysis.** *This figure shows the latency of every individual compaction during the 80 GB random load.*

tages and disadvantages of GearDB and LevelDB, we break down the time of all KV store operations (i.e., log, compaction, garbage collection, and other write operations) for the random load process. As shown in Figure 14, we observe that compared to Ldb-Greedy and Ldb-CB: 1) GearDB adds no overhead to any operations; and 2) GearDB's performance advantage mainly comes from the more efficient compaction and eliminated garbage collection. LevelDB has a longer random load time because garbage collections take about a quarter of the overall runtime and the compaction is less efficient than GearDB. We record the detailed information of garbage collections for Ldb-CB and Ldb-Greedy as follows: 1) the overall garbage collection time is 5,638 s and 5,670 s, which account for 23.14% and 23.47% of the overall random load time (24,360 s and 24,156 s); and 2) the migration data volume in garbage collection is 417 GB and 430 GB, which is 25.53% and 25.77% of the overall disk writes.

Compaction Efficiency. To understand the compaction efficiency of the three key-value stores specifically, we

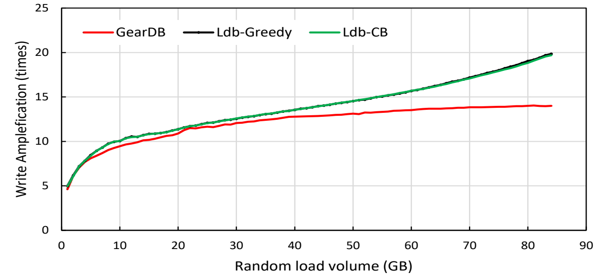


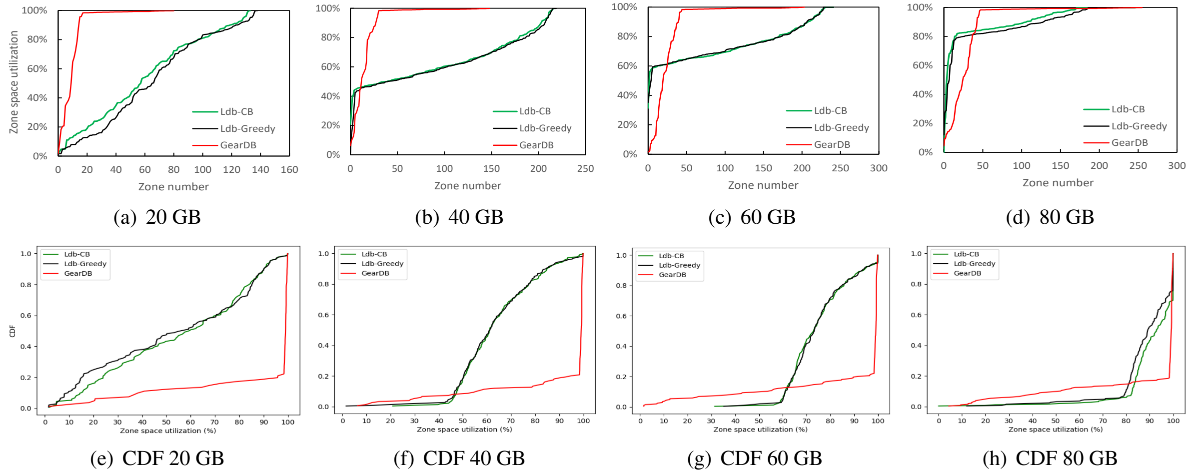
Figure 16: **Write amplification.** *This figure shows the write amplification factor of three KV systems when we load different sizes of the database (i.e., from 1 GB to 80 GB).*

record the compaction latency for every compaction during the random loading process, which is shown in Figure 15. From this figure, we make the following three observations. First, GearDB dramatically reduces the number of compactions (i.e., 53,311 and 53,203 less than Ldb-Greedy and Ldb-CB respectively). Since GearDB continues gear compaction to higher levels when key ranges overlap with the compaction window of the adjacent level, more data are compacted in one compaction process, reducing the number of compactions. Second, the average compaction latency of GearDB is higher than LevelDB because gear compaction involves more data in each compaction. Third, the overall compaction latency is 1.80× shorter in GearDB than LevelDB with greedy or cost-benefit strategies.

Write Amplification. Write amplification (WA) is an important factor in the performance of key-value stores. We calculate the write amplification factor by dividing the overall disk-write volume by the total volume of user data written. The WA of the three systems is shown in Figure 16. In both LevelDB and GearDB, the WA increases with the volume of the database as the compaction data volume increases. Ldb-Greedy and Ldb-CB have a similar large write amplification because they need to migrate data in both compactions and garbage collections. GearDB reduces the write amplification since it performs no on-disk garbage collections.

5.3 Space Efficiency Evaluation

Figure 17 shows a comparison of zone usage and zone space utilization after randomly loading 20, 40, 60, and 80 GB databases. From these results, we find GearDB occupies fewer zones than LevelDB after loading the same size database. For example, GearDB saves 71 zones (i.e., 17.75 GB) when storing a 40 GB database. Moreover, GearDB has higher zone space utilization than LevelDB, i.e., GearDB's average space utilization of loading the 80 GB database is 89.9%. We show the corresponding CDFs of zone space utilization in Figure 17. These results show that GearDB restricts fragments in a small portion of occupied zones (i.e.,



(a) 20 GB (b) 40 GB (c) 60 GB (d) 80 GB

(e) CDF 20 GB (f) CDF 40 GB (g) CDF 60 GB (h) CDF 80 GB

Figure 17: **Zone space utilization.** *Figures a-d show the zone space utilization of each occupied zone when we randomly load 20, 40, 60, and 80 GB database. Figures e-h show the corresponding CDF of the zone space utilization. The results show that our design consistently maintains a high space efficiency.*

compaction windows). GearDB achieves a bimodal zone space utilization, where most zones are nearly full, and a few zones are nearly empty since they are in compaction windows. This bimodal distribution not only improves space utilization but also wipes out garbage collections, since an empty zone can be reused by resetting write pointers without incurring data migration. LevelDB suffers from low space utilization, especially for smaller databases, since fewer GCs are triggered when the database is small. Once more garbage collections are triggered, LevelDB's space efficiency improves at the cost of system performance. However, GearDB achieves a high space utilization during its lifetime without sacrificing system performance. The overall performance and space efficiency gain of GearDB is denoted by the red triangle mark in Figure 5.

5.4 Extended Evaluations

Sensitivity Study of the Compaction Window Size. We evaluate GearDB with 5 different compaction window sizes by changing the k in Equation 2 (i.e., k=2, 4, 6, 8, 10). The experimental setup is the same as used for the microbenchmarks. Consistently, GearDB maintains a random load throughput ranging from 1,314 to 1,470 operations/s, and a space utilization ranging from 86% to 91%. Since the compaction window size does not have a significant influence on system performance and space efficiency, we set $k = 4$ as the default CW size for GearDB.

GearDB vs. SMRDB. SMRDB [38] is an HM-SMR friendly KV store, which enlarges the SSTable to a zone size (e.g., 256 MB) to prevent overwriting and reduces the number of levels to two to reduce compactions. We implement and then evaluate SMRDB using db_bench. Test results show that SMRDB is slower than GearDB by 1.97× for random loading. SMRDB brings severe compaction latency due to the large data volume involved in each compaction. GearDB has similar sequential read performance, and 1.68× faster random read performance compared to SMRDB since the large SSTable increases the overhead of fetching KV items in SSTables.

GearDB on HM-SMR vs. LevelDB on HDD. To further demonstrate the potential of GearDB, we compare GearDB on HM-SMR to LevelDB on a Seagate hard disk drive (ST1000DM003). The original LevelDB uses the standard file system interface (i.e., Ext4 in our evaluation), and we call it Ldb-hdd. The basic performance evaluation on db_bench shows that GearDB on HM-SMR outperforms Ldb-hdd by 2.38× for randomly loading an 80 GB database. GearDB has higher sequential write performance and similar random read performance with Ldb-hdd. However, Ldb-hdd has the superiority on sequential read (i.e., 7.02× faster) due to the file system cache. Our GearDB bypasses the file system and thus does not benefit from the cache.

6 Related Work

GearDB is an LSM-tree based key-value store tailored for HM-SMR drives, which aims to realize both good performance and high space utilization. We first discuss existing works that exploit HM-SMR drives without compromising system performance. ZEA provides HM-SMR software abstractions that map zone block addresses to the logical block addresses of HDDs [33]. SMRDB [38] is an HM-SMR friendly KV store described and evaluated in Section 5.4. Caveat-Scriptor [23] and SEALDB [47] allow to write anywhere on HM-SMR drives by letting the host write beware.

Kinetic [41] provides KV Ethernet HM-SMR drives plus an open API to support object storage. Huawei's key-value store (KVS) [31] provides simple and redundant KV accesses on HM-SMR drives via a core design of a log-structured database. SMORE [32] is an object store on an array of HM-SMR drives, which also accesses disks in a log-structured approach. HiSMRfs [22] stores file metadata on SSDs and stores file data on SMR drives.

Next, we discuss research to enhance and improve LSM-trees by reducing the write amplification caused by compactions. Lwc-tree [48] performs lightweight compactions by appending data to SSTables and only merging metadata. PebblesDB [39] mitigates writes by using guards to maintain partially sorted levels. WiscKey [29] separates keys from values and compacts keys only, thus reducing compaction overhead by eliminating value migrations. VTtrees [44] use an extra layer of indirection to avoid reprocessing sorted data, at the cost of fragmentation. TRIAD [34] uses a holistic of three technologies on memory, disk, and log to reduce write amplification. Blsm [43] proposes a new merge scheduler to synchronize merge completions, and hence obviates upstream writes from waiting downstream merges. LSM-trie [46] de-amortizes compaction overhead with hash-range based compaction for better read performance. [25] and [24, 13] optimize LSM-trees tailored for specific storage devices and specific application scenarios. In contrast, GearDB improves system performance via providing a new data layout that facilitates the data fetching in compaction and eliminating write amplification from redundant GC.

Third, recent works have sought to optimize or manage SSDs at the application layer [30, 14, 35]. They aim to solve the double logging problem in both FTLs and append-only applications via new block I/O interfaces [30] or application-driven FTLs [14]. However, there still exists the need to employ GC policies for reclaiming flash segments in FTLs. By contrast, GearDB eliminates the overhead of disk space cleaning via three design strategies.

Finally, SSD streams [6, 26] associate data with similar update frequencies or lifetimes to the same stream and place it into the same unit for multi-stream SSD. The data layout of GearDB shares the initial consideration of separating data with similar lifetimes. However, the methodology is different entirely, e.g., [6] assigns write requests of multiple levels to dedicated streams.

7 Conclusion

In this paper, we present GearDB, an LSM-tree based key-value store tailored for HM-SMR drives. GearDB is designed to achieve both good performance and high space utilization with three techniques: a new data layout, compaction windows, and a novel gear compaction algorithm. We implement GearDB on a real HM-SMR drive. Experimental results show that GearDB improves the overall system performance and space utilization, i.e., 1.71× faster than LevelDB in random write with a space efficiency of 89.9%. GearDB's performance gains mainly come from efficient gear compaction by eliminating garbage collections. The open source GearDB is available at *https://github.com/PDS-Lab/GearDB*.

8 Acknowledgement

We thank our shepherd Bill Jannen and the anonymous reviewers for their insightful comments and feedback. We also thank Seagate Technology LLC for providing the sample HM-SMR drive to run our experiments. This work was sponsored in part by the National Natural Science Foundation of China under Grant No.61472152, No.61300047, No.61432007, and No.61572209; the 111 Project (No.B07038); the Director Fund of WNLO. The work performed at Temple was partially supported by the U.S. National Science Foundation grants CCF-1717660 and CNS-1702474.

References

[1] AGHAYEV, A., AND DESNOYERS, P. Skylight—a window on shingled disk operation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 135–149.

[2] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M. S., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference* (2008).

[3] AMER, A., LONG, D. D. E., MILLER, E. L., PARIS, J.-F., AND SCHWARZ, S. J. T. Design issues for a shingled write disk system. In *Proceedings of the 2010 IEEE 26th Symposium on MSST* (2010).

[4] CASSUTO, Y., SANVIDO, M. A. A., GUYOT, C., HALL, D. R., AND BANDIC, Z. Z. Indirection systems for shingled-recording disk drives. In *Proceedings of the 2010 IEEE 26th Symposium on MSST* (2010), pp. 1–14.

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)* (2006), pp. 205–218.

[6] CHOI, C. Increasing ssd performance and lifetime with multi-stream technology. `https://www.snia.org/sites/default/files/DSI/2016/presentations/sec/ChanghoChoi_Increasing_SSD_Performance-rev.pdf`, 2016.

[7] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'10)* (2010).

[8] DAYAN, N., ATHANASSOULIS, M., AND IDREOS, S. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 79–94.

[9] FACEBOOK. RocksDB, a persistent key-value store for fast storage enviroments. `http://rocksdb.org/`.

[10] FELDMAN, T., AND GIBSON, G. Shingled magnetic recording: Areal density increase requires new data management. *USENIX; login: Magazine 38*, 3 (2013), 22–30.

[11] GHEMAWAT, S., AND DEAN, J. Leveldb. https://github.com/Level/leveldown/issues/298, 2016.

[12] GIBSON, G., AND GANGER, G. Principles of operation for shingled disk devices. *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-11-107* (2011).

[13] GOLAN-GUETA, G., BORTNIKOV, E., HILLEL, E., AND KEIDAR, I. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)* (2015).

[14] GONZÁLEZ, J., BJØRLING, M., LEE, S., DONG, C., AND HUANG, Y. R. Application-driven flash translation layers on open-channel ssds. In *Nonvolatile Memory Workshop (NVMW)* (2014).

[15] HGST. Hgst delivers world's first 10tb enterprise hdd for active archive applications. http://investor.wdc.com/news-releases/news-release-details/hgst-delivers-worlds-first-10tb-enterprise-hdd-active-archive, 2015.

[16] HGST. Libzbc version 5.4.1. https://github.com/hgst/libzbc, 2017.

[17] HGST. Ultrastar Hs14 — 14tb 3.5 inch helium platform enterprise smr hard drive. https://www.hgst.com/products/hard-drives/ultrastar-hs14, 2017.

[18] HGST. Ultrastar dc hc600 smr series, 15TB. https://www.westerndigital.com/products/data-center-drives/ultrastar-dc-hc600-series-hdd, 2018.

[19] INCITS T10 TECHNICAL COMMITTEE. Information technology-zoned block commands (ZBC). draft standard t10/bsr INCITS 550, american national standards institute, inc. http://www.t10.org/drafts.htm, 2017.

[20] INCITS T13 TECHNICAL COMMITTEE. Zoned-device ata command set (ZAC) working draft.

[21] JAGADISH, H., NARAYAN, P., SESHADRI, S., SUDARSHAN, S., AND KANNEGANTI, R. Incremental organization for data recording and warehousing. In *VLDB* (1997), vol. 97, pp. 16–25.

[22] JIN, C., XI, W.-Y., CHING, Z.-Y., HUO, F., AND LIM, C.-T. HiSMRfs: A high performance file system for shingled storage array. In *Proceedings of the 2014 IEEE 30th Symposium on MSST* (2014), IEEE, pp. 1–6.

[23] KADEKODI, S., PIMPALE, S., AND GIBSON, G. A. Caveat-Scriptor: Write anywhere shingled disks. In *7th USENIX Workshop on HotStorage* (2015).

[24] KAI, R., QING, Z., JOY, A., AND GARTH, G. SlimDB—a space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment 10*, 13 (2017).

[25] KANNAN, S., BHAT, N., GAVRILOVSKA, A., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Redesigning lsms for nonvolatile memory with novelsm. In *USENIX Annual Technical Conference* (2018), pp. 993–1005.

[26] KIM, T., HAHN, S. S., LEE, S., HWANG, J., LEE, J., AND KIM, J. Pcstream: Automatic stream allocation using program contexts. In *10th USENIX Workshop on HotStorage* (2018).

[27] KU, S. C.-Y., AND MORGAN, S. P. An smr-aware append-only file system. In *Storage Developer Conference* (2015).

[28] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware* (2009).

[29] LANYUE, L., SANKARANARAYANA, P. T., C, A.-D. A., AND H, A.-D. R. WiscKey: separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016), pp. 133–148.

[30] LEE, S., LIU, M., JUN, S. W., XU, S., KIM, J., AND ARVIND, A. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016), pp. 339–353.

[31] LUO, Q., AND ZHANG, L. Implement object storage with smr based key-value store. In *Storage Developer Conference* (2015).

[32] MACKO, P., GE, X., KELLEY, J., SLIK, D., ET AL. SMORE: A cold data object store for smr drives. In *Proceedings of the 2017 IEEE 33th Symposium on MSST* (2017).

[33] MANZANARES, A., WATKINS, N., GUYOT, C., LEMOAL, D., MALTZAHN, C., AND BANDIC, Z. ZEA, a data management approach for smr. In *8th USENIX Workshop on HotStorage* (2016).

[34] MARIA, B. O., DIEGO, D., RACHID, G., WILLY, Z., HUAPENG, Y., AASHRAY, A., KARAN, G., AND PAVAN, K. TRIAD: creating synergies between memory, disk and log in log structured key-value stores. In *USENIX Annual Technical Conference* (2017).

[35] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. NVMKV: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on HotStorage* (2014).

[36] MARTIN, M., TIM, H., KRSTE, A., AND JOHN, K. Trash day: Coordinating garbage collection in distributed systems. In *HotOS* (2015).

[37] ONEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. The log-structured merge-tree (lsm-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[38] PITCHUMANI, R., HUGHES, J., AND MILLER, E. L. SMRDB: key-value data store for shingled magnetic recording disks. In *Proceedings of the 8th ACM International Systems and Storage Conference* (2015).

[39] RAJU, P., KADEKODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (2017), ACM, pp. 497–514.

[40] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 26–52.

[41] SEAGATE. The seagate kinetic open storage vision. https://www.seagate.com/tech-insights/kinetic-vision-how-seagate-new-developer-tools-meets-the-needs-of-cloud-storage-platforms-master-ti/.

[42] SEAGATE. Archive hdds from seagate. http://www.seagate.com/www-content/product-content/hdd-fam/seagate-archive-hdd/en-us/docs/100757960a.pdf, 2014.

[43] SEARS, R., AND RAMAKRISHNAN, R. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD 12)* (2012).

[44] SHETTY, P., SPILLANE, R. P., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with VT-trees. In *11th USENIX Conference on File and Storage Technologies (FAST 13)* (2013), pp. 17–30.

[45] WU, F., YANG, M.-C., FAN, Z., ZHANG, B., GE, X., AND H.C.DU, D. Evaluating host aware smr drives. In *8th USENIX Workshop on HotStorage* (2016).

[46] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An lsm-tree-based ultra-large key- value store for small data. In *USENIX Annual Technical Conference* (2015).

[47] YAO, T., TAN, Z., WAN, J., HUANG, P., ZHANG, Y., XIE, C., AND HE, X. A set-aware key-value store on shingled magnetic recording drives with dynamic band. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2018), IEEE, pp. 306–315.

[48] YAO, T., WAN, J., HUANG, P., HE, X., GUI, Q., WU, F., AND XIE, C. A light-weight compaction tree to reduce i/o amplification toward efficient key-value stores. In *Proceedings of the 2017 IEEE 33th Symposium on MSST* (2017).

SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution

Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia
The University of Edinburgh

Christof Fetzer[†], Michio Honda[‡], Kapil Vaswani[*]
[†]*TU Dresden,* [‡]*NEC Labs,* [*]*Microsoft Research*

Abstract

We introduce SPEICHER, a secure storage system that not only provides strong confidentiality and integrity properties, but also ensures data freshness to protect against rollback/forking attacks. SPEICHER exports a Key-Value (KV) interface backed by Log-Structured Merge Tree (LSM) for supporting secure data storage and query operations. SPEICHER enforces these security properties on an untrusted host by leveraging shielded execution based on a hardware-assisted trusted execution environment (TEE)—specifically, Intel SGX. However, the design of SPEICHER extends the trust in shielded execution beyond the secure SGX enclave memory region to ensure that the security properties are also preserved in the stateful (or non-volatile) setting of an untrusted storage medium, including system crash, reboot, or migration.

More specifically, we have designed an authenticated and confidentiality-preserving LSM data structure. We have further hardened the LSM data structure to ensure data freshness by designing asynchronous trusted counters. Lastly, we designed a direct I/O library for shielded execution based on Intel SPDK to overcome the I/O bottlenecks in the SGX enclave. We have implemented SPEICHER as a fully-functional storage system by extending RocksDB, and evaluated its performance using the RocksDB benchmark. Our experimental evaluation shows that SPEICHER incurs reasonable overheads for providing strong security guarantees, while keeping the trusted computing base (TCB) small.

1 Introduction

With the growth in cloud computing adoption, online data stored in data centers is growing at an ever increasing rate [11]. Modern online services ubiquitously use persistent key-value (KV) storage systems to store data with a high degree of reliability and performance [39, 65]. Therefore, persistent KV stores have become a fundamental part of the cloud infrastructure.

At the same time, the risks of security violations in storage systems have increased significantly for the third-party cloud computing infrastructure [66]. In an untrusted environment, an attacker can compromise the security properties of the stored data and query operations. In fact, many studies show that software bugs, configuration errors, and security vulnerabilities pose a serious threat to storage systems [9, 12, 16, 20, 24, 35, 37].

However, securing a storage system is quite challenging because modern storage systems are quite complex [9, 49, 64, 72]. For instance, a persistent KV store based on the Log-Structured Merge Tree (LSM) data structure [54] is composed of multiple software layers to enable a data path to the storage persistence layer. Thereby, the enforcement of security policies needs to be carried out by various layers in the system stack, which could expose the data to security vulnerabilities. Furthermore, since the data is stored outside the control of the data owner, the third-party storage platform provides an additional attack vector. The clients currently have limited support to verify whether the third-party operator, even with good intentions, can handle the data with the stated security guarantees.

In this landscape, the advancements in trusted execution environments (TEEs), such as Intel SGX [4] or ARM TrustZone [7], provide an appealing approach to build secure systems. In fact, given the importance of security threats in the cloud, there is a recent surge in leveraging TEEs for shielded execution of applications in the untrusted infrastructure [8, 10, 55, 69, 75]. *Shielded execution* aims to provide strong security properties using a hardware-protected secure memory region or *enclave*.

While the shielded execution frameworks provide strong security guarantees against a powerful adversary, they are primarily designed for securing "stateless" (or volatile) in-memory computations and data. Unfortunately, these stateless techniques are not sufficient for building a secure storage system, where the data is persistently stored on an untrusted storage medium, such as an SSD or HDD. The challenge is *how to extend the trust beyond the "secure, but stateless/volatile" enclave memory region to the "untrusted and persistent" storage medium, while ensuring that the security properties are preserved in the "stateful settings", i.e., even across the system reboot, migration, or crash.*

To answer this question, we aim to build a secure storage sys-

tem using shielded execution targeting all three important security properties for the data storage and query processing: (a) *confidentiality* — unauthorized entities cannot read the data, (b) *integrity* — unauthorized changes to the data can be detected, and (c) *freshness* — stale state of data can be detected as such.

To achieve these security properties, more specifically, we need to address the following three architectural limitations of shielded execution in the context of building a secure storage system: Firstly, the secure enclave memory region is quite limited in size, and incurs high performance overheads for memory accesses. It implies that the storage engine cannot store the data inside the enclave memory; thus, the in-memory data needs to be stored in the untrusted host memory. Furthermore, the storage engine persists the data on an untrusted storage medium, such as SSDs. Since the TEE cannot give any security guarantees beyond the enclave memory, we need to design mechanisms for extending the trust to secure the data in the untrusted host memory and also on the persistent storage medium.

Secondly, the syscall-based I/O operations are quite expensive in the context of shielded execution since the thread executing the system call has to exit the enclave, and perform a secure context switch, including TLB flushing, security checks, etc. While existing shielded execution frameworks [8, 55] proposed an *asynchronous* system call interface [70], it is clearly not well-suited for building a storage system that requires frequent I/O calls. To mitigate the expensive enclave exits caused by I/O syscalls, we need to design a direct I/O library for shielded execution to completely eliminate the expensive context switch from the data path.

Lastly, we also aim to ensure data freshness to protect against *rollback* (replay old state) or *forking attacks* (create second instance). Therefore, we need a protection mechanism based on a trusted monotonic counter [57], for example, SGX trusted counters [3]. Unfortunately, the SGX trusted counters are extremely slow and they wear out within a couple of days of operation. To overcome the limitations of the SGX counters, we need to redesign the trusted monotonic counters to suit the requirements of modern storage systems.

To overcome these design challenges, we propose SPEICHER, a secure LSM-based KV storage system. More specifically, we make the following contributions.

- **I/O library for shielded execution:** We have designed a direct I/O library for shielded execution based on Intel SPDK. The I/O library performs the I/O operations without exiting the secure enclave; thus it avoids expensive system calls on the data path.
- **Asynchronous trusted monotonic counter:** We have designed trusted counters to ensure data freshness. Our counters leverage the lag in the sync operations in modern KV stores to asynchronously update the counters. Thus, they overcome the limitations of the native SGX counters.
- **Secure LSM data structure:** We have designed a secure LSM data structure that resides outside of the enclave memory while ensuring the integrity, confidentiality and freshness of the data. Thus, our LSM data structure overcomes the memory and I/O limitations of Intel SGX.
- **Algorithms:** We present the design and implementation of all storage and query operations in persistent KV stores: get, put, range queries, iterators, compaction, and restore.

We have built a fully-functional prototype of SPEICHER based on RocksDB [65], and extensively evaluated it using the RocksDB benchmark suite. Our evaluation shows that SPEICHER incurs reasonable overheads, while providing strong security properties against powerful adversaries.

2 Background and Threat Model

2.1 Intel SGX and Shielded Execution

Intel Software Guard Extension (SGX) is a set of x86 ISA extensions for Trusted Execution Environment (TEE) [15]. SGX provides an abstraction of secure *enclave*—a hardware-protected memory region for which the CPU guarantees the confidentiality and integrity of the data and code residing in the enclave memory. The enclave memory is located in the Enclave Page Cache (EPC)—a dedicated memory region protected by an on-chip Memory Encryption Engine (MEE). The MEE encrypts and decrypts cache lines with writes and reads in the EPC, respectively. Intel SGX supports a call-gate mechanism to control entry and exit into the TEE.

Shielded execution based on Intel SGX aims to provide strong confidentiality and integrity guarantees for applications deployed on an untrusted computing infrastructure [8, 10, 55, 69, 75]. Our work builds on the SCONE [8] shielded execution framework. In SCONE, the applications are statically compiled and linked against a modified standard C library (SCONE libc). In this model, application's address space is confined to the enclave memory, and interaction with the untrusted memory is performed via the system call interface. In particular, SCONE runtime provides an *asynchronous system call* mechanism [70] in which threads outside the enclave asynchronously execute the system calls. SCONE protects the executing application against Iago attacks [13] through *shields*. Furthermore, it ensures memory safety for the applications running inside the SGX enclaves [36]. Lastly, SCONE provides an integration to Docker for seamlessly deploying containers.

2.2 Persistent Key-Value (KV) Stores

Our work focuses on persistent KV stores based on the LSM data structure [54], such as LevelDB [39] and RocksDB [65]. In particular, we base our design on RocksDB. RocksDB organizes the data using three constructs: MemTable, static sorted table (SSTable), and log files.

RocksDB inserts `put` requests to a memory-resident *MemTable* that is organized as a skip list [62]. For crash recovery, these `puts` are also sequentially logged to the write-ahead-log (WAL) file backed by persistent storage medium with checksums. When the MemTable fills up, it is moved to an *SSTable* file backed by an SSD or HDD in a batch to ensure sequential device access (this thus can cause scanning the skip list).

The SSTable files are grouped into levels with increasing size (typically 10×). The process of moving data to the next level is called *compaction*, which ensures the SSTables to be sorted by keys, including the ones being merged from the previous level. Since SSTables are immutable, compaction always creates new SSTables on the persistent storage medium. Any state changes in the entire storage system, such as creation and deletion of SSTable and WAL files, are recorded to the *Manifest*, which is a transactional and persistent log.

On a `get` request, RocksDB first searches the MemTable for the key, then searches the SSTables from the lowest level in turn; at each level, it binary-searches the corresponding SSTable. Using this primitive, it is trivial to process range and iterator queries, where the latter only differs in the client interface. RocksDB maintains an index table with a Bloom filter attached to each SSTable in order to avoid searching unnecessary SSTables.

While restarting, RocksDB establishes the latest state in a `restore` operation. To this end, the Manifest and the WAL are read and replayed.

2.3 Threat Model

In addition to the standard SGX threat model [10], we also consider the security attacks that can be launched using an *untrusted* storage medium, e.g., persistent state stored on an SSD or HDD. More specifically, we aim to protect against a powerful adversary in the virtualized cloud computing infrastructure [10]. In this setting, the adversary can control the entire system software stack, including the OS or hypervisor, and is able to launch physical attacks, such as performing memory probes.

For the untrusted storage component, we also aim to protect against rollback attacks [57], where the adversary can arbitrarily shut down the system, and replay from a stale state. We also aim to protect against forking attacks [40], where the adversary can attempt to fork the storage system, e.g., by running multiple replicas of the storage system.

Even under the extreme threat model, our goal is to guarantee the data integrity, confidentiality, and freshness. Lastly, we also aim to provide crash consistency for the storage system [58].

However, we do not protect against side-channel attacks, such as exploiting cache timing and speculative execution [78], or memory access patterns [25, 81]. Mitigating side channel attacks in the TEEs is an active area of research [53]. Further, we do not consider the denial of service attacks since these attacks are trivial for a third-party operator controlling the underlying infrastructure [10]. Lastly, we assume that the adversary cannot physically open the processor packaging to extract secrets or corrupt the CPU system state.

3 Design

SPEICHER is a secure persistent KV storage system designed to operate on an untrusted host. SPEICHER provides strong confidentiality, integrity, and freshness guarantees for the data storage and query operations: `get`, `put`, `range queries`, `iterators`, `compaction`, and `restore`. In this paper, we implemented SPEICHER by extending RocksDB [65], but our architecture can be generalized to other LSM-based KV stores.

3.1 Design Challenges

As a strawman design, we could try to secure a storage system by running the storage engine inside the enclave memory. However, the design of a practical and secure system requires addressing the following four important architectural limitations of Intel SGX.

I: Limited EPC size. The strawman design would be able to protect the in-memory state of the MemTable using the EPC memory. However, EPC is a limited and shared resource. Currently, the size of EPC is 128 MiB. Approximately 94 MiB are available to the user, the rest is reserved for the metadata. To allow creation of enclaves with sizes beyond that of EPC, SGX features a secure paging mechanism. The OS can evict EPC pages to an unprotected memory using SGX instructions. During eviction, the page is re-encrypted. Similarly, when an evicted page is brought back, it is decrypted and its integrity is checked. However, the EPC paging incurs high performance overheads (2×—2000×) [8].

Therefore, we need to redesign the shielded storage engine, where we allocate the MemTable(s) outside the enclave in the untrusted host memory. Since the secure enclave region cannot give any guarantees for the data stored in the host memory, and the native MemTable is not designed for security—we designed a new MemTable data structure to guarantee the confidentiality, integrity and freshness properties.

II: Untrusted storage medium. The storage engine does not exclusively store the data in the in-memory MemTable, but also on a persistent storage medium, such as on an SSD or HDD. In particular, the storage engine stores three types of files on a persistent storage medium: SSTable, WAL and the Manifest. However, Intel SGX is designed to protect only the volatile state residing in the enclave memory. Unfortunately, SGX does not provide any security guarantees for stateful computations, i.e., across system reboot or crash. Further, the trust from the TEE does not naturally extend to the untrusted persistent storage medium.

To achieve the end-to-end security properties, we further redesigned the LSM data structure, including the persistent storage state in the SSTable and log files, to extend the trust to the untrusted storage medium.

III: Expensive I/O syscall. To access data stored on an SSD or HDD (in the SSTable, WAL or Manifest files), conventional systems leverage the system call interface. However, the system call execution in the SGX environment incurs high performance overheads. This is because the thread executing the system call has to exit the enclave, and the syscall arguments need to be copied in and out of the enclave memory. These enclave transitions are expensive because of security checks and TLB flushes.

To mitigate the context switch overhead, shielded execution frameworks, such as SCONE [8] or Eleos [55], provide an

asynchronous system call interface [70], where a thread outside the enclave asynchronously executes the system calls without forcing the enclave threads to exit the enclave. While such an asynchronous interface is useful for many applications, it is not clearly suited for building a storage system that needs to support frequent I/O system calls.

To support frequent I/O calls within the enclave, we designed a new I/O mechanism based on a direct I/O library for shielded execution leveraging storage performance development kit (SPDK) [28].

IV: Trusted counter. In addition to guaranteeing the integrity and confidentiality, we also aim to ensure the freshness of the stored data to protect against rollback attacks [57]. To achieve the freshness property, we need to protect the data stored in the untrusted host memory (MemTable), and those on the untrusted persistent storage medium (SSTable, WAL and Manifest files).

For the first part, i.e., to ensure the freshness of MemTable allocated in the untrusted host memory, we can leverage the EPC of SGX. In particular, the Memory Encryption Engine (MEE) in SGX already protects the EPC against rollback attack. Therefore, we use the EPC to store a *freshness signature* of the MemTable, which we use at runtime to verify the freshness of data stored as part of the MemTable in the untrusted host memory.

However, the second part is quite tedious, i.e., to ensure the freshness of the data stored on untrusted persistent storage (SSTables and log files), because the rollback protected EPC memory is stateless, or it cannot be used to verify the freshness properties after the system reboots or crashes. Therefore, we need a rollback protection mechanism based on a trusted monotonic counter [57]. For example, we could use SGX trusted counters [3]. Unfortunately, the SGX trusted counters are extremely slow (60 − 250 ms) [45]. Furthermore, the counter memory allows only a limited number of write operations to NVRAM, and it easily becomes unusable due to wear out within a couple of days of operation. Therefore, the SGX counters are impractical to design a storage system.

To overcome the limitations of SGX counters, we designed an asynchronous trusted monotonic counter that drastically improves the throughput and mitigates wear-out by taking advantage of the crash consistency properties of modern storage systems.

3.2 System Components

We next detail the system components of SPEICHER. Figure 1 illustrates the high-level architecture and building blocks of SPEICHER. The system is composed of the controller, a direct-I/O library for shielded execution, a trusted monotonic counter, the storage engine (RocksDB engine), and a secure LSM data structure (MemTable, SSTable, and log files).

SPEICHER controller. The controller provides the trusted execution environment based on Intel SGX [8]. Clients communicate over a mutually authenticated encrypted channel (TLS) to the controller. The TLS channel is terminated inside

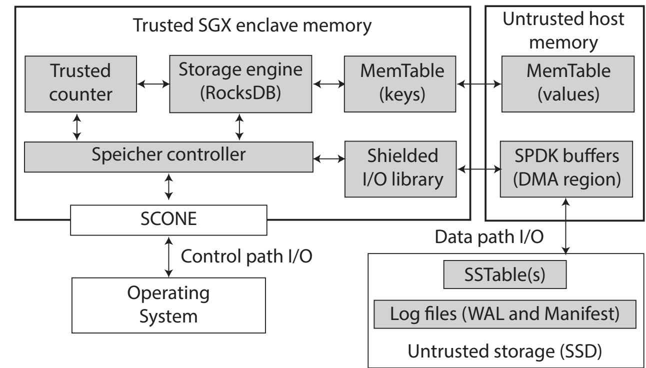


Figure 1: SPEICHER overview (shaded boxes depict the system components)

the controller. In particular, we built the controller based on the SCONE shielded execution framework [8], where we leverage SCONE's container support for secure deployment of the SPEICHER executable on an untrusted host.

The controller provides the remote attestation service to the clients [6, 32]. In particular, the SGX enclave generates a signed measured of its identity, whose authenticity can be verified by a third party. After successful attestation, the client provides its encryption keys to the controller. The controller uses the client certificate to perform the access control operation. The controller also provides runtime support for user-level multithreading and memory management inside the enclave. The controller leverages the asynchronous system calls interface (SCONE libc) on the control path for the system configuration. For the data path I/O, we built a direct I/O library, which we describe next.

Shielded direct I/O library. The I/O library allows the storage engine to access the SSD or HDD from inside the SGX enclave, without issuing the expensive enclave exit operations. We achieve this by building a direct I/O library for shielded execution based on SPDK [28].

SPDK is a high-performance user-mode storage library, based on Data Plane Development Kit (DPDK) [2]. It eliminates the need to issue system calls to the kernel for read and write operations by having the NVMe driver in the user space. SPDK enables zero-copy I/O by mapping DMA buffers to the user address space. It relies on actively polling the device instead of interrupts.

These SPDK features align with the goal of SPEICHER of exit less I/O operations in the enclave, i.e., to allow the shielded storage engine to interact with the SSD directly. However, we need to adapt the design of SPDK to overcome the limitations of the enclave memory region. In particular, our shielded I/O library allocates huge pages and SPDK ring buffers outside the enclave for DMA. The host system maps the device in an allocated DMA region. Afterwards SPDK can initialize the device. To reduce the number of enclave exits, SPDK's device driver runs inside the enclave. This enables efficient delivery of requests from the storage engine to the driver, which explicitly copies the data between the host and the enclave memory.

Trusted counter. In order to protect the system from rollback attacks, we need a trusted counter whose value is stored alongside with the LSM data structure. Intel SGX provides monotonic counters, but their update frequency is in a range of 10 updates per second, and we indeed measured approximately 250 ms to increment a counter once. This is far too slow for modern KV stores [26].

To overcome the limitations of SGX counters, we designed an Asynchronous Monotonic Counter (AMC) based on the observation that many contemporary KV stores do not persist their inserted data immediately. This allows AMC to defer the counter increment until the data is persisted without loosing any availability guarantees. As a result, AMC achieves 70K updates per second in the current implementation.

AMC provides an asynchronous increment interface, because it takes a while since the counter value is incremented until it becomes *stable*, which means the counter value cannot be rolled back without being detected. At an increment, AMC returns three pieces of information: the current stable value, the incremented counter value, and the *expected time* for the value to be stable. Due to the expected time and the controller having to be re-authenticated after a shutdown, the client only has to keep the values until the stable time has elapsed, to prevent any data loss in case of a sudden shutdown.

AMC's flexible interface allows us to optimize update throughput and latency by increasing the time until a trusted counter is stable. This also allows users to adjust trade-off between the wear out of the SGX monotonic counter and the maximum number of unstable counter increments, which a client might have to account for. SPEICHER generates multiple counters by storing their state to a file, whose freshness is guaranteed through the use of a synchronous trusted monotonic counter. For instance, we can employ SGX monotonic counters [3], ROTE [45] or Ariadne [71] to support our asynchronous interface. Therefore, we can have a counter with deterministic increments for WAL and the Manifest, making it possible to argue about the freshness of each record in the files.

MemTable. As detailed in §3.1, the EPC is limited in size and the EPC paging incurs very high overheads. Therefore, it is not judicious to store large MemTables or multiple MemTables within the EPC. Further, since SPEICHER uses the EPC memory region to secure the storage engine (RocksDB) and the shielded I/O library driver, it further shrinks the available space.

Due to this memory restriction, we need to store the MemTable in the host memory. Since the host memory is untrusted, we need to devise a mechanism to ensure the confidentiality, integrity, and freshness of the MemTable.

In our project, we tried three different designs for the MemTable. Firstly, we explored a native Merkle tree that generates hashes of the leafs and stores them in each node. Thus, we can verify the data integrity by checking the root node hash and each hash down to the leaf storing the KV, while allowing the MemTable to be stored outside the EPC memory. However, the native Merkle tree suffers from slow lookups as the key has

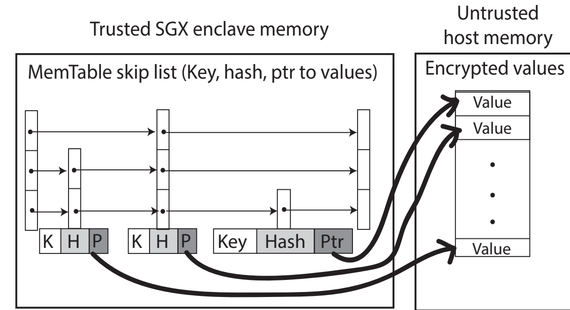


Figure 2: SPEICHER MemTable format

to be decrypted on each traversal. Further, it requires multiple hash recalculations on each lookup and insertion.

Secondly, we tried a modified Merkle tree design based on a prefix array, where a fixed size prefix is used as an index into the array of Merkle trees. An array entry holds the root node of a Merkle tree, which holds the actual data. This should reduce the depth of the search tree compared to the native Merkle tree; thus, reducing the number of necessary hash calculations and decryptions of keys. However, while we were able to increase the lookup speed compared to the native Merkle tree, it still suffered from the same problem of having to decrypt a large number of keys in a lookup, and causing a large number of hash calculations.

Lastly, our third attempt of the MemTable design reuses the existing skip list data structure for the MemTable in RocksDB. Figure 2 shows SPEICHER's MemTable format. In particular, we partition the existing MemTable in two parts: key path and value path. In the key path, we store the keys as part of the skip list inside the enclave. Whereas, the encrypted values in the MemTable are stored in the untrusted host memory as part of the value path. This partitioning allows SPEICHER to provide confidentially by encrypting the value, while still enabling fast key lookups inside the enclave. To prevent attacks on the integrity or the freshness of the values, SPEICHER stores a cryptographic hash of the value in each skip list node together with the host memory location of the value.

While the first two designs removed almost the entire MemTable from the EPC, the last design still maintains the keys and hash values inside the enclave memory. To determine the space requirements of our MemTable in comparison to the regular RocksDB's MemTable, we use the following formula:

$$S = n*(k+v) + \sum_{i=0}^{m} p^i * n * ptr$$

Where S represents the entire size of the skip list, n is the number of KV pairs, k is the key size, v is the value size or the size of the pointer plus hash value for our skip list, p is the probability for being added into a specific layer of the skip list, m is the maximum number of layers, and ptr is the size of a pointer in the system.

For instance, in case of the default setting for RocksDB, with a maximum size of 64 MiB, key size of 16 B, value size

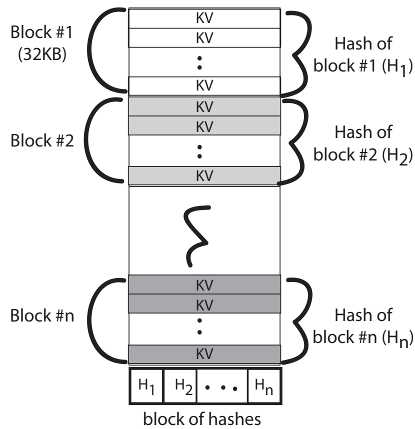


Figure 3: SPEICHER SSTable file format

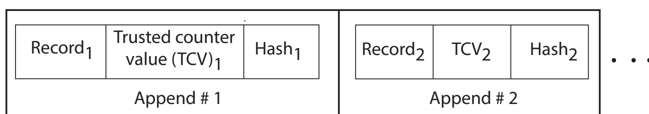


Figure 4: SPEICHER append-only log file format

of 1024 B, pointer size of 8 B, p of 1/4, m of 12 and for SPEICHER's skip list a hash size of 16 B — SPEICHER's MemTable achieves a space reduction of approximately 95.2 %. Further, the reduction ratio increases with increased value size.

SSTables. The SSTable files maintain the KV pairs persistently. These files store KV pairs in the ascending order of keys. This organization allows for a binary search within the SSTable, requiring only a few reads to find a KV-pair within the file. Since SSTable files are optimized for block devices, such as SSDs, they group KV pairs together into blocks (the default block size is 32 KiB).

SPEICHER adapts SSTable file format to ensure the security properties (see Figure 3 for SPEICHER's SSTable file format). The confidentiality is secured by encrypting each block of the SSTable file before it is written to the persistent storage medium. Additionally, SPEICHER calculates a cryptographic hash over each block. These hashes are then grouped together in a block of hashes and appended at the end of the SSTable file. When reading SPEICHER can check the integrity of each block by calculating the block's hash and comparing it to the corresponding hash stored in the footer. To protect the integrity of the footer an additional hash over the footer is calculated and stored in the Manifest. Since the Manifest is protected against rollback attacks using a trusted counter, the footer hash value stored in the Manifest is also protected from the rollback attacks. Thus, SPEICHER can use this hash to guarantee the freshness of the SSTable file's footer and transitively the freshness of each block in the SSTable file.

Log files. RocksDB uses two different log files to keep track of the state of the KV store: (a) WAL for persisting inserted KV pairs until a top-level compaction; and (b) the Manifest to keep track of live files, i.e., the set of files of which the current state of the KV store consists. SPEICHER adapted these log files to ensure the desired security properties, as shown in Figure 4.

Regarding WAL, every `put` operation appends a record to the current WAL. This record consists of the encrypted KV pair, and an encrypted trusted counter value for the WAL at the moment of insertion, and a cryptographic hash over both. Since the records are only appended to the WAL, SPEICHER can use the trusted counter value and the hash value to verify the KV pair, and to replay the operations in a restore event.

The Manifest is similar to the WAL; it is a write-append log consisting of records storing changes of live files. We use the same scheme for the Manifest file as we do for the WAL.

3.3 Algorithms

We next present the algorithms for all storage operations in SPEICHER. The associated pseudocodes are detailed in the appendix.

I: Put. Put is used to insert a new KV pair into the KV store, or to update an existing one. We need to perform two operations to insert the KV pair into the store (see Algorithm 1). First, we need to append the KV pair to the WAL for persistence. Second, we need to write the KV pair to the MemTable for fast lookups.

Inserting the KV pair into the WAL guarantees that the state of the KV store can be restored after an unexpected reboot. Therefore, the KV pair should be inserted into the WAL before it is inserted into the MemTable. To add a KV pair to the WAL, SPEICHER encrypts the pair together with the next WAL trusted counter value and a cryptographic hash over both the data and the counter. The encrypted block is then appended to the WAL (see the log file format in Figure 4). Thereafter, the trusted counter is incremented to the value stored in the appended block. In addition, the client is notified when the KV pair will be stable; thereafter, the state cannot be rolled back. In case of a system crash between generating the data block and increasing the trusted counter value, the data block would be invalid at reboot, because the trusted counter would point the block to a future time. This operation is safe as the client can detect a reboot when SPEICHER tries to authenticate itself. After the reboot the client can ask the KV store about what the last added key was, or can simply `put` the KV pair again in the store as another request with the same key supersedes any old value with the same key.

In the second step, SPEICHER writes the KV pair into the MemTable and thereby making the `put` visible to later `gets`. SPEICHER first encrypts the value of the KV pair and generates a hash over the encrypted data. The encrypted value is then copied to the untrusted host memory, while the hash with a pointer to the value is inserted into the skip list in the enclave, in accordance to SPEICHER's MemTable format (Figure 2). Since the KV pair is first inserted into the WAL, and only if this is successful, i.e., the WAL and trusted counter are updated, we can guarantee that only KV value pairs whose freshness is secured by the trusted counter are returned.

II: Get. Get may involve searching multiple levels in the LSM data structure to find the latest value. Within each level, SPEICHER has to generate either the proof of existence, or the

proof of non-existence of the key. This is necessary to detect insertion or deletion of the KV pairs by an attacker.

Algorithm 2 details the `get` operation in SPEICHER. In particular, SPEICHER begins with searching the MemTable. SPEICHER searches the skip list for the node with the key. Either the key is in the MemTable, then the hash value is calculated over the value and compared to the hash stored in the skip list, or the key could not be found in the skip list. Since the skip list resides inside the protected memory region, SPEICHER does not need to make the non-existence proof for the MemTable because an attacker cannot access the skip list. If the KV store finds a key in the MemTable and the existence proof is correct, i.e., the calculated hash value is equal to the stored hash value, the value is returned to the client. If the proof is incorrect, the client is informed that the MemTable is corrupted. Since the MemTable can be reconstructed from the WAL, the client can then instruct the SPEICHER to recreate the KV store state in the case of an incorrect proof.

When the key is not found in the MemTable, the next level is searched. All levels below the MemTable are stored in SSTables. The SSTable files are organized in a way that no two SSTables in the same level have an overlapping key-range. Additionally, all the keys are sorted within an SSTable file. Due to this, any given key can only exist in one position in one SSTable file per level. This allows SPEICHER to construct a Merkle tree on top of the SSTable files of a level. With the ordering inside the SSTable, SPEICHER can correlate a block in the file with the key. This allows SPEICHER to calculate a hash over this block, which then can be checked against the stored hash in the footer. The hash of the footer can then be checked against the Merkle tree over the SSTable files in that level. It gives SPEICHER the proof of non-/existence for the lookup, and possibly the value belonging to the key. If the proof fails, the client is informed. In contrast to an incorrect proof in the MemTable, SPEICHER is not able to recover from this problem since the data is stored on the untrusted storage medium. If SPEICHER finds the KV pair and the proof is correct, it returns the value to the client. If the key does not exist, that is SPEICHER could not find it in any level and all level proofs are correct, an empty value is returned.

The freshness of data is guaranteed either by checking the value against the securely stored hash in the EPC for the case where the key has been found in the MemTable, or by checking the hash values of the SSTables against a Merkle tree. Additionally, as any key can only be stored in one position within a level, SPEICHER can also check against deletion of the key in a higher level, which is also necessary to guarantee freshness.

III: Range queries. Range queries are used to access all KV pairs, with a key greater than or equal to a start key and lesser than an end key (see Algorithm 3). To find the start KV pair, we need to do the same operation as in get requests. Furthermore, it requires to initialize an iterator in each level, pointing to the KV pair with a key greater or equal to the starting key. These iterators are necessary as higher levels have the more recent updates, due to keys being inserted into the highest level and being compacted over time to the lower levels, and lower being larger in size and therefore having more KV pairs. If the next KV pair is requested the next key of all iterators is checked and the iterators with the smallest next key are forwarded.

In case the next key is in multiple levels, the highest level KV pair is chosen. Therefore, SPEICHER has to do a non-/existence proof at all the levels, before it returns the chosen KV pair. If any of these proofs fails, the client is informed about the failed proof. Identical to the `get` operation, the client can then decide to either restore the KV store or to restore a backup.

Similar to the `get` operation, the hash value stored in the EPC and the Merkle tree over the SSTables are used to guarantee the freshness of the returned values.

IV: Iterators. Iterators work identical to the range queries; they just have a different interface (see Algorithm 4).

V: Restore. After a reboot, the KV store has to restore its last state (see Algorithm 5). This process is performed in two steps, first collecting all files belonging to the KV store, and then replaying all changes to the MemTable. In the first step the Manifest file is read. It contains all necessary information about the other files, such as live SSTable files, live WAL files, smallest key of each SSTable file. Each changing event about the live file is logged into the Manifest by appending a record describing the event. Therefore, at a restore all changes committed in the Manifest have to be replayed. This means that the SSTable files have to be put in the correct level. Each record in the Manifest is integrity-checked by a hash, and the freshness is guaranteed by the trusted counter for the Manifest. Since the counter value is incremented in a deterministic way, SPEICHER can use this value to check if all blocks are present in the Manifest. After the SSTable files in the levels are restored, and the freshness of all the SSTable files is checked against the Manifest by comparing the hash with the hash stored in the Manifest, the WAL is replayed.

Since each `put` operation is persisted in the WAL before it is written into the MemTable, replaying the `put` operations from the WAL allows SPEICHER to reconstruct the MemTable at the moment of the shutdown. Each `put` in the WAL has to be checked against the stored hash in the record, and the stored counter value. Additionally, since the counter value of the WAL is checked whether it equals to that of the Manifest counter, SPEICHER can check for the missing records. Records that have a counter value being in the future, i.e. a counter value higher than the stored stable trusted counter value are ignored at restore. Further, due to the deterministic increase of the counter, SPEICHER can check against the missing records in the log files. If in any of these steps one of the checks fails, SPEICHER returns the information to the client, because SPEICHER is not able to recover from such a state.

VI: Compaction. Compaction is triggered when a level holds data beyond a pre-defined threshold in size. In compaction (see Algorithm 6), a file from Level_n is merged with all SSTable files in Level_{n+1} covering the same key range. The

new SSTables are added to Level$_{n+1}$, while all SSTables in the previous level are discarded. Before keys are added to the new SSTable file, the non-/existence proof is done on the files being merged. This is necessary to prevent the compaction process from skipping keys or writing old KV to the new SSTable files.

Since hash values are calculated over blocks of the SSTable files, a new block has to be constructed in the enclave memory, before it is written to the SSD. Also, all hash values of the blocks have to be stored in the protected memory until the footer is written and a hash over the footer is created. The file names of newly created SSTables and footer hashes are then written to the Manifest file, with the new trusted counter value. This is similar to the `put` operation. After the write operation to the Manifest completes and the trusted counter is incremented, the old SSTable files are removed from the KV store and the new files are added to Level$_{n+1}$. Since the hash values of the new SSTables are secured with a trusted counter value in the Manifest file, the SSTables cannot be rolled back after the compaction process.

3.4 Optimizations

Timer performance. As described in §3.2, in order to prevent every request from blocking for the trusted counter increment, we leverage asynchronous counters written in files whose freshness is guaranteed by synchronous counters (or SGX counters). We use one counter for the WAL and another for the Manifest so that SPEICHER can operate on them independently. Although this method drastically improves throughput by allowing SPEICHER to process many requests without waiting for the counter to be stable, it also poses on the client the need for holding its write requests until the counter value is stable. This is why we designed and implemented the interface of AMC that reports the expected time for the counter to be stable. Because of this interface, the client does not need to frequently issue the requests to check the current stable counter value.

SPDK performance. SPDK is designed to eliminate system calls from the data path, but in reality its data path issues two system calls on every I/O request: one for obtaining the process identifier and the other for obtaining the time. They are executed once in an I/O request that covers multiple blocks and their costs are normally amortized. However, since the context switch to and from the enclave is an order of magnitude more expensive, these costs are not amortized enough. We modified them to obtain the values from a cache within the enclave that are updated only at the vantage points. As a result, we achieved 25× improvements over the naive port of SPDK to the enclave.

4 Implementation

Direct I/O library. Our direct I/O library for shielded execution extends Intel SPDK. Further, the memory management routines and the `uio` kernel module that maps the device memory to the user space are based on Intel DPDK [2]. Although the device DMA target is configured outside the enclave, the SSD device driver and library code, including BlobFS in which SPEICHER stores RocksDB files, entirely run within the enclave.

We use SPDK 18.01.1-pre and DPDK 18.02. In SPDK, 56 LoC are added, and 22 LoC are removed. In DPDK, 138 LoC are added and 72 LoC are removed. These changes were made to replace the routines that cannot be executed in the enclave.

Trusted counters. AMCs are implemented using the Intel SGX SDK. A dedicated thread continually checks if any monotonic counter value has changed. If a counter value has been incremented, the thread writes the current value to the file. The storage engine can query the *stable value* of any of its counters, i.e., the last value that has been written to disk. Note that this value cannot be rolled back since it is protected by the synchronous SGX monotonic counter. Overall, our trusted counter consists of 922 LoC.

SPEICHER controller. The SPEICHER controller is based on SCONE. We leverage the Docker integration in SCONE to seamlessly deploy SPEICHER binary on an untrusted host. Further, we implemented a custom memory allocator for the storage engine. The memory allocator manages the unprotected host memory, and exploits RocksDB's memory allocation pattern, which allows us to build a lightweight allocator with just 119 LoC. Further, the controller employs our direct I/O library on the data path, and the asynchronous `syscall` interface of SCONE on the control path for system configuration. The controller also implements a TLS-based remote attestation for the clients [32]. Lastly, we integrated the trusted counter as a part of the controller, and exported the APIs to the storage engine.

Storage engine. We implemented the storage engine by extending a version of RocksDB that leverages SPDK. In particular, we extended the RocksDB engine to run within the enclave, also integrated our direct I/O library. Since the RocksDB engine with SPDK does not support data encryption and decryption, we also ported encryption support from the regular RocksDB engine using the Botan Library [1] (1000 LoC). In addition to encrypting data files, we extended the encryption support to ensure the confidentiality of the WAL and Manifest files. We further modified the storage engine to replace the LSM data structure and log files with our secure MemTable, SSTables, and log files. Altogether, the changes in RocksDB account for 5029 new LoC and 319 changed LoC.

MemTables. RocksDB as default uses a skip list for MemTable. However, it does not offer any authentication or freshness guarantees. Therefore, we replaced MemTable with an authenticated data structure coupled with mechanisms to ensure the freshness property. Our MemTable uses the `Inlineskiplist` of RocksDB and replaces the value part of the KV-pair with a node storing a pointer to and the size of the value as well as an HMAC. For the en-/decryption as well as for the HMAC we used OpenSSLs AES128 in GCM mode. This results in a 16 B wide HMAC. This implementation consists of 459 LoC. As discussed previously, we also implemented MemTable with a native Merkle tree (1186 LoC) and a Merkle tree with a prefix array (528 LoC). However, we did not use them eventually since their performance was quite low.

SSTables. To preserve the integrity of the SSTable blocks, we changed the block layer in RocksDB to calculate the hash before it issues a write request to the underlying layer. The hash is then cached until the file is flushed (258 LoC). Thereafter, hashes of all blocks are appended to the file coupled with the information about the total number of blocks, and the hash of this footer. When a file is opened, our hash layer loads the footer into the protected memory and calculates the hash of the footer. It then compares the value against the hash stored in the Manifest file. Only if these checks are passed, it opens the corresponding SSTable file and normal operations proceed. At reading, the hash of the block is calculated and checked against the hashes stored in the protected memory area, before the block data is handed to the block layer of RocksDB. We further enabled AES128 encryption to ensure the confidentiality of the blocks (188 LoC). The hashes used in the SSTables are SHA-3 with 384 bit.

Log files. Log files including the WAL and the Manifest use the same encryption layer as the SSTable files. However, the validation layer is different, and comes before the block layer since the operation requires knowledge of the record size. While writing, the validation layer adds the hash and the trusted counter value to the log files.

The validation layer uses the knowledge that log files are only read sequentially at startup for restoring purpose. Therefore, at the start up, the layer allows any action written in the log file as long as the hash is correct, and the stored counter increases as expected. At the end of the file, SPEICHER checks if the stored counter is equal to the trusted counter. The last record's freshness is guaranteed through the trusted counter. Integrity of all the records is guaranteed through the hash value protecting also the stored counter value. This value can then be checked against the expected counter value for that block. Since the counter lives longer than the log files, the start record value has to be secured too. In case of WAL, this is achieved by storing the start counter value of the WAL in the Manifest. The start record of the Manifest is implicitly secured, since the record must describe the state of the entire KV store.

5 Evaluation

Our evaluation answers the following questions.

- What is the performance (IOPS and throughput) of the direct I/O library for shielded execution? (§5.2)
- What is the impact of the EPC paging on the MemTable? (§5.3)
- What are the performance overheads of SPEICHER in terms of throughput and latency measurements? (§5.4)
- What is the performance of our asynchronous trusted counter? And what stability guarantees it has to provide to be compatible with modern KV stores? (§5.5)
- What is the I/O amplification overhead? (§5.6)

5.1 Experimental Setup

Testbed. We used a machine with Intel Xeon E3-1270 v5 (3.60 GHz, 4 cores, 8 hyper-threads) with 64 GiB RAM running Linux kernel 4.9. Each core has private 32 KiB L1 and 256 KiB L2 caches, and all cores share a 8 MiB L3 cache. For the storage device our testbed uses a Intel DC P3700 SSD. The SSD has a capacity of 400 GB and is connected over PCIe x4.

Workload	Pattern	Read/Write ratio
A *(default)*	Read-write	90R—10W
B	Read-write	80R—20W
C	Read only	100R—0W

Table 1: RocksDB benchmark workloads.

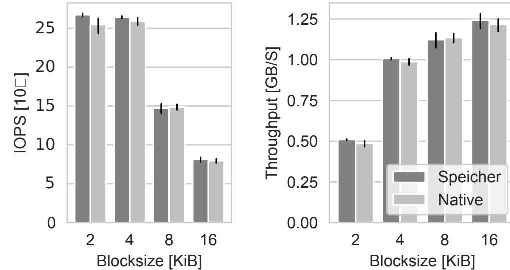


Figure 5: Performance of direct I/O library for shielded execution vs native SPDK.

Methodology for measurements. We compare the performance of SPEICHER with an unmodified version of RocksDB. The native version of RocksDB does not provide any security guarantees, i.e., it provides no support for confidentiality, integrity and freshness of the data and query operations.

Importantly, we stress-test the system by running a client on the same machine as the KV store. This is the worst-case scenario for SPEICHER since the client is not communicating over the network. Usually, the network slows down client's requests, and therefore, such an experimental setup is unable to stress-test the KV store. We avoid this scenario by running the client as part of the same process on the same host. This eliminates further the need for enclave enters and exits, which would add a high overhead, making a stress-test impossible.

Compiler and software versions. We used the RocksDB version with SPDK support (git commit `3c30815`). We used SPDK version 18.01.1-pre (git commit `73fee9c`), which we compiled with DPDK version 18.02 (commit `92924b2`). The native version of SPDK/ DPDK and RocksDB was compiled with `gcc 6.3.0` and the default release flags. The SPEICHER version of SPDK/DPDK and RocksDB was compiled with the same release flags but `gcc version 7.3.0` of the SCONE project.

RocksDB benchmark suite. We use the RocksDB benchmark suite for the evaluation. In particular, we used the `db_bench` benchmarking tool which is shipped with RocksDB [5] and Fex [52]. The benchmark consists of three workloads as shown in Table 1. Workload A is the default workload.

5.2 Performance of the Direct I/O Library

We first evaluate the performance of SPEICHER's I/O library for shielded execution. The I/O library is designed to have fast access to the persistent storage for accessing the KV pair stored

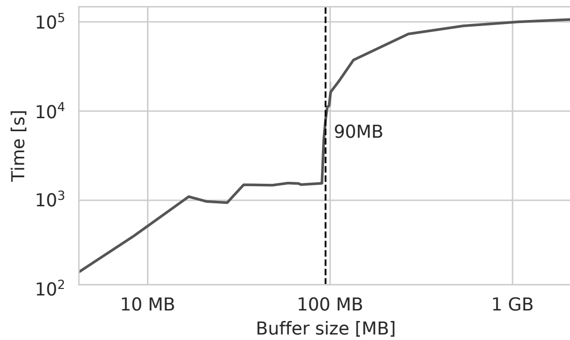


Figure 6: Impact of the EPC paging on the MemTable.

on the SSD (§3.2). We run the performance measurement 20 times for every configuration of block size for the native execution and SPEICHER. Figure 5 shows the mean throughput and IOPS with our I/O library and those with the native RocksDB-SPDK with a confidence interval of 95%. We use Workload B (80%R—20%W). Since the communication between SPDK and the device is handled completely over DMA, our direct I/O library does not suffer from context switches. Additionally, due to storing the buffers outside of the enclave, we also do not require expensive EPC paging, which would drastically reduce the performance of the I/O library. Our performance evaluation of the direct I/O library shows that it does not suffer from any performance deprecation compared to the native SPDK setup.

5.3 Impact of the EPC paging on MemTable

We next study the impact of EPC paging on MemTable(s). Note that a naive solution of storing a large or many MemTables in the EPC memory would incur high performance overheads due to the EPC paging. Therefore, we adopted a split MemTable approach, where we store only the keys along with metadata (hashes and pointers to value) inside the EPC, but the values are stored in the untrusted host memory (§3.3). To confirm the overheads of the EPC paging on accessing a large MemTable which are incurred in our rejected design, we measure the overheads of accessing random nodes in a MemTable completely resident in the enclave memory.

Figure 6 shows the performance overhead of accessing memory within the SGX enclave. The result shows that as soon as SGX has to page out MemTable memory from the EPC, which happens at 96 MiB, the performance drops dramatically. This is due to the en-/decryption and integrity checks employed by the MEE in Intel SGX. Therefore, it is important for our system design to keep the data values in the untrusted host memory to avoid the expensive EPC paging. Our approach of only keeping the key path of the MemTable inside the EPC requires a small EPC memory footprint. Therefore, our MemTable does not incur the EPC paging overhead.

5.4 Throughput and Latency Measurements

We next present the end-to-end performance of SPEICHER with different workloads, value sizes and thread counts. We measured the average throughput and latency for each of our benchmarks. Figure 7 shows the measurement results as a ratio of slowdown to the native SPDK-based RocksDB.

Effect of varying workloads. In the first experiment, we used different workloads listed in Table 1. The workloads were evaluated with 5 million KV pairs each. Each key was 16 B and value was 1024 B. The benchmarks were run single threaded.

We get a throughput of 34.2k request/second (rps) for Workload A down to 20.8k rps for Workload C, while RocksDB archived 512.8k rps or 676.8k rps respectively. The results show that SPEICHER overheads 15×—32.5× for different workloads. The overheads in Workloads A and B are mainly due to the operations performed in the MemTable, since SPEICHER has to encrypt the value and generate a cryptographic hash for every write to the MemTable. Furthermore, for each read operation the data has to be decrypted and the hash has to be recalculated and compared to one in the Skip list. However, even with AES-NI instructions, this decryption operation takes at least 1.3 cycles/byte for encryption, limiting the maximal reachable performance. The overhead in Workload C is due to reading a very high percentile of the KV pairs from the SSTable files, which uses currently an un-optimized code path for en-/decryption and hash calculations. We expect performance improvement by further optimizing the code path.

Effect of varying byte sizes. In the second experiment, we investigate the overheads with varying value sizes, since it changes the amount of data SPEICHER has to en-/decrypt and hash for each request. We used the default Workload A, and changed the value size from 64 B up to 4 KiB.

SPEICHER incurs an overhead of 6.7× for small value size, i.e. 64 B, up to an overhead of 16.9× for values of size 4 KiB. As in the previous experiment, the overhead is mainly dominated by the en-/decryption and hash calculation for the values in the MemTable. The benchmark shows a higher overhead for larger value sizes, since the amount of data SPEICHER has to en-/decrypt increases with the size of the values.

Effect of varying threads. We also investigated the scaling capabilities of SPEICHER. For that we increased the number of threads up to 8 and compared the overhead to native RocksDB with the default Workload A. Note that the current SGX server machine has 4 physical cores / 8 hyperthread cores.

In the test the overhead increased from around 13.6× for two threads to 17.5× for 8 threads. This implies SPEICHER scales slightly worse than RocksDB. This is due to less optimal caching for random memory access in SPEICHER's memory allocator. SPEICHER has to manage two different memory regions (host and EPC) for the MemTable, which leads to sub-optimal caching. We plan to optimize our memory allocator and data structures to exploit the cache locality.

Latency measurements. In the benchmarks, SPEICHER has an average latency ranging from 16 µs for single threaded and 64 B value size up to 256 µs for 8 threads and 1024 B value size, native RocksDB had for the same benchmark a latency of 1.6 µs or 14 µs respectively. However, RocksDB's best latencies were in Workload C with an average of 1.5 µs.

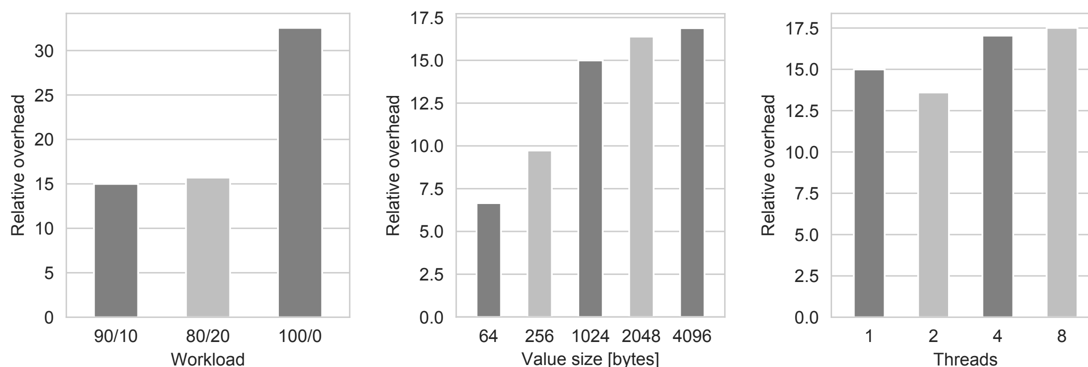


Figure 7: SPEICHER performance normalized to the native RocksDB (with no security): (a) different workloads with constant value size of 1024 and constant number of threads, (b) varying value sizes, and (c) increasing number of threads.

KV store	Default time for persistence (ms)	Configurable
RocksDB	0 (flushing)	yes
LevelDB	0 (non-flushing)	yes
Cassandra	1000	yes
HBase	10000	yes

Table 2: Default time for data persistence in KV stores.

5.5 Performance of the Trusted Counter

The synchronous trusted counter rate of SGX is limited to one increment at every 60 ms. This would limit our approach to only 20 `Put` operations per second since each `Put` has to be appended to the WAL, which requires a counter increment. However, our latency suggest that we have a lot more put operations to deal with. Even in our worse latency case with 256 µs per request we would expect 234.4 request per 60 ms, with a write rate of 10% this would amount to 23.4 required counter increases every possible sequential counter increase. In practice SPEICHER should reach far higher update rates as this calculation used worst case values from our benchmarks.

Table 2 shows the time before different KV stores guarantee that the values are persisted. We argue that these times can be used to hide the stability time of our asynchronous counters, which is a maximum of 60 ms. This is far less than the maximum time to persist the data in the default configuration of Cassandra and HBase. If the client expects the value is persisted only after a specific period of time, we can relax our freshness guarantees to match to the same time window.

5.6 I/O Amplification

We measured the relative I/O amplification increase in data for SPEICHER compared to the native RocksDB. We report the I/O amplification results using the default workload (A) with the key size of 16 B and value size of 4 KiB. We observed an overhead of 30% for read and write in the I/O amplification. This overhead mainly comes from the footer we have to add to each SSTable as well as from the hashes and counter values we have to add to the log files. This overhead is not only present in the write case but also in the read, as the additional data has also to be read to be able to verify the files.

6 Related Work

Shielded execution. Shielded execution frameworks provide strong security guarantees for legacy applications running on an untrusted infrastructure. Prominent examples include Haven [10], SCONE [8], Graphene-SGX [75], Panoply [69], and Eleos [55]. Recently, there has been a significant interest in designing secure systems based on shielded execution, such as VC3 [68], Opaque [82], Ryoan [27], Ohrimenko et al. [51], SGXBounds [36], etc. However, these systems are primarily designed to secure stateless computation and data. (Pesos [34] is an exception, see the policy-based storage systems section for the details.) In contrast, we present the first secure persistent LSM-based KV storage system based on shielded execution.

I/O for shielded execution. To mitigate the I/O overheads in SGX, shielded execution frameworks, such as Eleos [55] and SCONE [8], proposed the usage of an asynchronous system call interface [70]. While the asynchronous interface is sufficient for the low I/O rate applications—it can not sustain the performance requirements of modern storage/networked systems. To mitigate the I/O bottleneck, ShieldBox [73] proposed a direct I/O library based on Intel DPDK [2] for building a secure middlebox framework. Our direct I/O library is motivated by this advancement in the networking domain. However, we propose the first direct I/O library for shielded execution based on Intel SPDK [28] for the I/O acceleration in storage systems.

Trusted counters. A trusted monotonic counter is one of the important ingredients to protect against rollback and equivocation attacks. In this respect, Memoir [57] and TrInc [40] proposed the usage of TPM-based [74] trusted counters. However, TPM-based solutions are quite impractical because of the architectural limitations of TPMs. For instance, they are rate-limited (only one increment every 5 seconds) to prevent wear out. Therefore, they are mainly used for secure data access in the offline settings, e.g., Pasture [33].

Intel SGX has recently added support for monotonic counters [3]. However, SGX counters are also quite slow, and they wear out quickly (§3). To overcome the limitations, ROTE [45] proposed a distributed trusted counter service based on a consensus protocol. Likewise, Ariadne [71]

proposed an optimized technique to increment the counter by a single bit flip. Our asynchronous trusted counter interface is complimentary to these synchronous counter implementations. In particular, we take advantage of the properties of modern storage systems, where we can use these synchronous counters to support our asynchronous interface.

Policy-based storage systems. Policy-based storage systems allow clients to express fine-grained security policies for data management. In this context, a wide range of storage systems have been proposed to express client capabilities [22], enforce confidentiality and integrity [21], or enable new features that include data sharing [44], database interface [46], policy-based storage [19, 77], or policy-based data seal/unseal operations [67]. Amongst all, Pesos [34] is the most relevant system since it targets a similar threat model. In particular, Pesos proposes a policy-based secure storage system based on Intel SGX and Kinetic disks [31]. However, Pesos relies on trusted Kinetic disks to achieve its security properties, whereas SPEICHER targets an untrusted storage, such as an untrusted SSD. Secondly, Pesos is designed for slow trusted HDDs, where the additional overheads of the SGX-related operations are eclipsed by slow disk operations. In contrast, SPEICHER is designed for high-performance SSDs.

Secure databases/datastores. Encrypted databases, such as CryptDB [60], Seabed [56], Monomi [76], and DJoin [50], are designed to ensure the confidentiality of computation in untrusted environments. However, they are primarily for preserving confidentiality. In contrast, SPEICHER preserves all three security properties: confidentiality, integrity, and freshness.

EnclaveDB [61] and CloudProof [59] target a threat model and security properties similar to SPEICHER. In particular, EnclaveDB [61] is a shielded in-memory SQL database. However, it uses the secondary storage only for checkpoint and logging unlike SPEICHER. Hence, it does not solve the problem of freshness guarantee for the data stored in the secondary storage. Furthermore, the system implementation does not consider the architectural limitations of SGX. Secondly, CloudProof [59] is a key-value store designed for untrusted cloud environment. Unlike SPEICHER, it requires the clients to encrypt or decrypt data to ensure confidentiality, as well as to perform attestation procedures with the server, introducing a significant deployment barrier.

TDB [43] proposed a secure database on untrusted storage. It provides confidentiality, integrity, and freshness using a log-structured data store. However, TBD is based on a hypothetical TCB, and it does not address many practical problems addressed in our system design.

Obladi [17] is a KV store supporting transactions while hiding the access patterns. While it can effectively hide the values and their access pattern against the cloud provider, it needs a trusted proxy. In contrast, SPEICHER does not rely on a trusted proxy. Furthermore, Obladi does not consider rollback attacks.

Lastly, in parallel with our work, ShieldStore [30] uses a Merkle tree to build a secure in-memory KV store using Intel SGX. Since ShieldStore is an in-memory KV Store, it does not persist the data using the LSM data structure unlike SPEICHER.

Authenticated data structures. Authenticated data structures (ADS) [47] enable efficient verification of the integrity of operations carried out by an untrusted entity. The most relevant ADS for our work is mLSM [63], a recent proposal to provide integrity guarantee for LSM. In contrast to mLSM, our system provides stronger security properties, i.e., we ensure not only integrity, but also confidentiality and freshness. Furthermore, our system targets a stronger threat model, where we have to design a secure storage system leveraging Intel SGX.

Robust storage systems. Robust storage systems provide strong safety and liveness guarantees in the untrusted cloud environment [14, 42, 79]. In particular, Depot [42] protects data from faulty infrastructure in terms of durability, consistency, availability, and integrity. Likewise, Salus [79] proposed a block store robust storage system while ensuring data integrity in the presence of commission failures. A2M [14] is also a robust system against Byzantine faults, and provides consistent, attested memory abstraction to thwart equivocation. In contrast to SPEICHER, this line of work neither provides confidentiality nor freshness guarantees.

Secure file systems. There is a large body of work on software-based secure storage systems. SUNDR [41], Plutus [29], jVPFS [80], SiRiUS [23], SNAD [48], Maat [38] and PCFS [21] employ cryptography to provide secure storage in untrusted environments. None of them protect the system from rollback attacks, and our challenges to overcome overheads of shielded execution are irrelevant for them. Among all, StrongBox [18] provides file system encryption with rollback protection; however, it does not consider untrusted hosts.

7 Conclusion

In this paper, we presented SPEICHER, a secure persistent LSM-based KV storage system for untrusted hosts. SPEICHER targets all the three important security properties: strong confidentiality and integrity guarantees, and also protection against rollback attacks to ensure data freshness. We base the design of SPEICHER on hardware-assisted shielded execution leveraging Intel SGX. However, the design of SPEICHER extends the trust in shielded execution beyond the secure enclave memory region to ensure that the security properties are also preserved in the stateful setting of an untrusted storage medium.

To achieve these security properties while overcoming the architectural limitations of Intel SGX, we have designed a direct I/O library for shielded execution, a trusted monotonic counter, a secure LSM data structure, and associated algorithms for storage operations. We implemented a fully-functional prototype of SPEICHER based on RocksDB, and evaluated the system using the RocksDB benchmark. Our experimental evaluation shows that SPEICHER achieves reasonable performance overheads while providing strong security guarantees.

Acknowledgement. We thank our shepherd Umesh Maheshwari for the helpful comments.

8 Appendix

In this appendix, we present the pseudocode for all data storage and query operations in SPEICHER.

Algorithm 1: Put algorithm of SPEICHER

```
Input: KV-pair which should be inserted into the store.
Result: Freshness of MemTable
/* Generating a block with the trusted counter */
hash_Block ← hash(KV, counter_WAL + 1);
block ← encrypt(KV, counter_WAL + 1, hash_Block);
/* Writing the block to the persistent storage, before
   the trusted counter gets incremented */
write_WAL(block);
counter_WAL ← counter_WAL + 1;
/* Generating hash over the KV-pair for the Memtable */
hash_KV ← hash(KV);
/* Trying to insert into the memtable, if the memtable is
   corrupted return a failure */
freshness ← putIntoMemtable(KV, hash_KV);
return freshness
```

Algorithm 2: Get algorithm of SPEICHER

```
Input: Key in the format of the KV-store
Result: Freshness of the KV-pair and Value
for level = 0 to numberoflevels do /* Check in each level if
   key-value is existend, from highest to lowest */
    if level = Level_0 then /* First level lookup therefore
       lookup in MemTable */
        path, value ← lookupMemtable(key) /* It is
           possible that the value is empty, however we
           still have to do a proof of non-existence */
        foreach node ∈ path do /* Validate hash values of
           the trace to the leaf node */
            if hash(node.left, node.right) ≠ node.hash then
               /* check that the hash value of the child
               nodes is equal to the stored hash value */
                /* The integrity and freshness proof
                   failed */
                return stale_MemTable, value
            end
        end
        return fresh, value
    end
    else /* Lookup in a level backup by SST files */
        SST ← findSSTFile(level, key) /* Lookup over
           authentication structures similar to MemTable
           */
        block, value ← lookup(SSTs_level, key);
        if hash(block) ≠
        SST.hashBlock(block) or !freshness(SST) then
            return stale_SST, value
        end
        return fresh, value
    end
end
```

Algorithm 3: Range query algorithm of SPEICHER

```
Input: KV-pair with the lowest key and callback method to the
       client
/* Build an iterator pointing to the first KV-pair */
iterator ← constructIterator(key_min);
next ← True;
/* Call the provided function until the iterator is not
   valid anymore or a freshness proof failed or the
   client request to end */
while isValid(iterator) and state = fresh and next do
    state, value ← Iterator.key_value;
    next ← callback(state, value);
    Iterator ← Iterator.next;
end
```

Algorithm 4: Iterator functions of SPEICHER

```
Input: Start key
Result: Result of freshness proof or iterator
Function constructIterator(key_min)
    /* Build an iterator for each level of the LSM
       pointing to the KV-pair or the next pair in the
       level */
    foreach level ∈ Level do
        iterator_level ← lowerBound(level, key);
        if iterator_level.state ≠ fresh then
            return state
        end
        iterator.add(iterator_level);
    end
end
Input: iterator
Result: Iterator points to the next KV-pair and freshness of the
        iterator
Function next(iterator)
    /* Forward all iterators pointing to the current key
       */
    foreach iterator_level ∈ iterator where iterator_level.key =
    iterator.key do
        next(iterator_level);
        if iterator_level.state ≠ fresh then
            return iterator_level.state
        end
    end
    /* Find the level iterator pointing to the lowest key
       */
    for i = 0 to number_levels do
        iter ← iterator[i];
        if iter.state ≠ fresh then
            return iter.state
        end
        if key_lowest > iter.key then
            key_lowest ← iter.key;
            level ← i
        end
    end
    iterator.currentLevel(i);
    return fresh
end
```

Algorithm 5: Restore algorithm of SPEICHER

```
Input: Manifest File
Result: Restored KV-store
/* Get the counter value of the first record in the
   manifest and check that the first record is an inital
   record                                                */
counter ← Manifest.firstCounterValue;
/* Iterate over all records in the Manifest              */
foreach record_encrypted ∈ Manifest do
    record ← decrypt;
    hash ← hash(record);
    /* Check the records hash and counter value, if they
       do not match, report an error to the client       */
    if hash ≠ record.hash then
        return Hash does not match
    end
    if counter ≠ record.counter then
        return Counter does not match
    end
    /* If hash and counter match apply the change to the
       KV-store                                          */
    apply(record);
    inc(counter);
end
/* Check if the last counter in the Manifest matches the
   trusted counter, if not report an error to the client
   */
if counter ≠ tusted_counter_Manifest then
    return Counter does not match
end
/* Get the current WAL and its initial counter value from
   the Manifest                                          */
counter ← Manifest.firstWALCounter;
/* Apply each record of the WAL to the KV if the counter
   and hash are correct, similar to the Manifest         */
foreach record_encrypted ∈ WAL do
    record ← decrypt;
    hash ← hash(record);
    if hash ≠ record.hash then
        return Hash does notmatch
    end
    if counter ≠ record.counter then
        return Counter does not match
    end
    apply(record);
    inc(counter);
end
/* Check if the last counter value is the same as the
   trusted counter                                       */
if counter ≠ trusted_counter_WAL then
    return Counter does not match
end
/* KV-store was successfully restored and no integrity
   or rollbacks problem were found                       */
return Success
```

Algorithm 6: Compaction algorithm of SPEICHER

```
Input: SSTable file to be compacted one from level_n
Result: Multiple SSTable files for level_n+1
// Creating an Iterator over the higher level SSTable file create a new file
and a new data block
iterator_n ← createIterator(SSTable_n);
NewSSTable ← createNewSST();
block ← createNewBlock();
last_key ← iterator_n.key − 1;
// As long as their are KV-pairs remaining in the SSTable open the
SSTable file in the next level which has the range of the smallest possible
next key based on the last key compacted. while
has_next(iterator_n) do
    SSTable_n+1 ← findSSTFile(n+1, last_key+1);
    iterator_n+1 ← createIterator(SSTable_n+1);
    // As long as the currently open SST_n+1 file has KV-pairs find the
    smaller next key of SST_n and SST_n+1 file. If both have the same next
    key choose from SST_n file.
    while has_next(iterator_n+1 i) do
        iterator_min ← min(iterator_n, iterator_n+1);
        // test if the key value is still fresh, that is check the hash of the
        block compare in the SSTable file hash footer and check
        against the Manifest
        if iterator_min ≠ fresh then
            // If the key value is not fresh return error to client
            return iterator_min.state
        end
        // Add key to block, if the block is then over the size limit for
        blocks calculate a hash add the hash to the footer of the new
        file and write the block to persistent storage, and create a new
        block
        block.add(iterator_min.kv);
        if size(block) > block_size_limit then
            hash ← hash(block);
            encrypted_block ← encrypt(block);
            NewSSTable.write(encrypted_block);
            NewSSTable.addHash(hash);
            // If the file reaches the size limit after an append, write
            the footer to the storage and create a new SSTable
            if size(NewSSTable) > SSTable_size_limit then
                NewSSTable.writeFooter();
                NewSSTable ← createNewSST();
            end
            block ← createNewBlock;
        end
        last_key = iterator_min.key;
        next(iterator_min);
    end
end
// After compaction, flush the block & write the footer.
hash ← hash(block);
encrypted_block ← encrypt(block);
NewSSTtable.write(encrypted_block);
NewSSTable.addHash(hash);
NewSSTable.writeFooter();
// Write the changes to the Manifest file.
Manifest.remove(SST_n, SST_n+1 inrangeofSST_n);
Manifest.add(∀NewSSTfile);
```

References

[1] Botan Library. https://botan.randombit.net/. Last accessed: Jan, 2019.

[2] Intel DPDK. http://dpdk.org/. Last accessed: Jan, 2019.

[3] Intel, "SGX documentation: sgx create monotonic counter". https://software.intel.com/en-us/sgx-sdk-dev-reference-sgx-create-monotonic-counter/. Last accessed: Jan, 2019.

[4] Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx. Last accessed: Jan, 2019.

[5] RocksDB Benchmarking Tool. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools. Last accessed: Jan, 2019.

[6] I. Anati, S. Gueron, P. S. Johnson, and R. V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[7] ARM. Building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009. Last accessed: Jan, 2019.

[8] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, 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)*, 2016.

[9] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[10] 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)*, 2014.

[11] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-Accelerated Incremental Storage and Computation. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[12] P. Bhatotia, A. Wieder, R. Rodrigues, F. Junqueira, and B. Reed. Reliable data-center scale computations. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.

[13] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[14] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[15] V. Costan and S. Devadas. Intel SGX Explained, 2016.

[16] CRN. The ten biggest cloud outages of 2013. https://www.crn.com/slide-shows/cloud/240165024/the-10-biggest-cloud-outages-of-2013.htm, 2013. Last accessed: Jan, 2019.

[17] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[18] B. Dickens III, H. S. Gunawi, A. J. Feldman, and H. Hoffmann. Strongbox: Confidentiality, integrity, and performance using stream ciphers for full drive encryption. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[19] E. Elnikety, A. Mehta, A. Vahldiek-Oberwagner, D. Garg, and P. Druschel. Thoth: Comprehensive Policy Compliance in Data Retrieval Systems. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, 2016.

[20] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[21] D. Garg and F. Pfenning. A proof-carrying file system. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[22] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[23] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proceed-*

ings of the Network and Distributed System Security Symposium (NDSS), 2003.

[24] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[25] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[26] M. Honda, G. Lettieri, L. Eggert, and D. Santry. PASTE: A network programming interface for non-volatile main memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[27] 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)*, 2016.

[28] Intel Storage Performance Development Kit. http://www.spdk.io. Last accessed: Jan, 2019.

[29] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[30] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys)*, 2019.

[31] Kinetic Data Center Comparison. https://www.openkinetic.org/technology/data-center-comparison. Last accessed: Jan, 2019.

[32] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij. Integrating Remote Attestation with Transport Layer Security. 2018.

[33] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. Pasture: Secure offline data access using commodity trusted hardware. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[34] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[35] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer. Haft: Hardware-assisted fault tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[36] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)*, 2017.

[37] D. Kuvaiskii, O. Oleksenko, P. Bhatotia, P. Felber, and C. Fetzer. Elzar: Triple modular redundancy using intel avx. In *proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[38] A. W. Leung, E. L. Miller, and S. Jones. Scalable security for petascale parallel file systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2007.

[39] LevelDB. http://leveldb.org/. Last accessed: Jan, 2019.

[40] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[41] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[42] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. *Depot: Cloud Storage with Minimal Trust*. 2011.

[43] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation (OSDI)*, 2000.

[44] K. Mast, L. Chen, and E. Gün Sirer. Enabling Strong Database Integrity using Trusted Execution Environments. 2018.

[45] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security)*, 2017.

[46] A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel. Qapla: Policy compliance for database-backed systems. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[47] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

[48] E. L. Miller, D. D. Long, W. E. Freeman, and B. Reed.

Strong Security for Network-Attached Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[49] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[50] A. Narayan and A. Haeberlen. DJoin: differentially private join queries over distributed databases. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[51] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security)*, 2016.

[52] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, and C. Fetzer. Fex: A software systems evaluator. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[53] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[54] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). In *Acta Inf.*, 1996.

[55] M. Orenbach, M. Minkin, P. Lifshits, and M. Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the 12th ACM European ACM Conference in Computer Systems (EuroSys)*, 2017.

[56] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[57] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*, 2011.

[58] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Crash consistency. *ACM Queue*, 2015.

[59] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2011.

[60] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[61] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A Secure Database using SGX. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

[62] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communication of ACM (CACM)*, 1990.

[63] P. Raju, S. Ponnapalli, E. Kaminsky, G. Oved, Z. Keener, V. Chidambaram, and I. Abraham. mlsm: Making authenticated storage faster in ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[64] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[65] RocksDB | A persistent key-value store. https://rocksdb.org/. Last accessed: Jan, 2019.

[66] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards Trusted Cloud Computing. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[67] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security)*, 2012.

[68] F. Schuster, M. Costa, 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)*, 2015.

[69] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[70] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[71] R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *25th USENIX Security Symposium (USENIX Security)*, 2016.

[72] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[73] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2018.

[74] Trusted Computing Group. TPM Main Specification. `https://trustedcomputinggroup.org/tpm-main-specification`, 2011. Last accessed: Jan, 2019.

[75] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[76] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th international conference on Very Large Data Bases (VLDB)*, 2013.

[77] A. Vahldiek-Oberwagner, E. Elnikety, A. Mehta, D. Garg, P. Druschel, R. Rodrigues, J. Gehrke, and A. Post. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*, 2015.

[78] 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 (USENIX Security)*, 2018.

[79] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the salus scalable block store. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[80] C. Weinhold and H. Härtig. jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[81] 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)*, 2015.

[82] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

SLM-DB: Single-Level Key-Value Store with Persistent Memory

Olzhas Kaiyrakhmet, Songyi Lee
UNIST

Beomseok Nam
Sungkyunkwan University

Sam H. Noh, Young-ri Choi
UNIST

Abstract

This paper investigates how to leverage emerging byte-addressable persistent memory (PM) to enhance the performance of key-value (KV) stores. We present a novel KV store, the *Single-Level Merge DB* (SLM-DB), which takes advantage of both the B+-tree index and the Log-Structured Merge Trees (LSM-tree) approach by making the best use of fast persistent memory. Our proposed SLM-DB achieves high read performance as well as high write performance with low write amplification and near-optimal read amplification. In SLM-DB, we exploit persistent memory to maintain a B+-tree index and adopt an LSM-tree approach to stage inserted KV pairs in a PM resident memory buffer. SLM-DB has a single-level organization of KV pairs on disks and performs selective compaction for the KV pairs, collecting garbage and keeping the KV pairs sorted sufficiently for range query operations. Our extensive experimental study demonstrates that, in our default setup, compared to LevelDB, SLM-DB provides 1.07 - 1.96 and 1.56 - 2.22 times higher read and write throughput, respectively, as well as comparable range query performance.

1 Introduction

Key-value (KV) stores have become a critical component to effectively support diverse data intensive applications such as web indexing [15], social networking [8], e-commerce [18], and cloud photo storage [12]. Two typical types of KV stores, one based on B-trees and the other based on Log-Structured Merge Trees (LSM-tree) have been popularly used. B-tree based KV stores and databases such as KyotoCabinet [2] support fast read (i.e., point query) and range query operations. However, B-tree based KV stores show poor write performance as they incur multiple small random writes to the disk and also suffer from high write amplification due to dynamically maintaining a balanced structure [35]. Thus, they are more suitable for read-intensive workloads.

LSM-tree based KV stores such as BigTable [15], LevelDB [3], RocksDB [8] and Cassandra [28] are optimized to efficiently support write intensive workloads. A KV store based on an LSM-tree can benefit from high write throughput that is achieved by buffering keys and values in memory and sequentially writing them as a batch to a disk. However, it has challenging issues of high write and read amplifications and slow read performance because an LSM-tree is organized with multiple levels of files where usually KV pairs are merge-sorted (i.e., compacted) multiple times to enable fast search.

Recently, there has been a growing demand for data intensive applications that require high performance for both read and write operations [14, 40]. Yahoo! has reported that the trend in their typical workloads has changed to have similar proportions of reads and writes [36]. Therefore, it is important to have optimized KV stores for both read and write workloads.

Byte-addressable, nonvolatile memories such as phase change memory (PCM) [39], spin transfer torque MRAM [21], and 3D XPoint [1] have opened up new opportunities to improve the performance of memory and storage systems. It is projected that such persistent memories (PMs) will have read latency comparable to that of DRAM, higher write latency (up to 5 times) and lower bandwidth (5∼10 times) compared to DRAM [19, 23, 24, 27, 42]. PM will have a large capacity with a higher density than DRAM. However, PM is expected to coexist with disks such as HDDs and SSDs [23, 25]. In particular, for large-scale KV stores, data will still be stored on disks, while the new persistent memories will be used to improve the performance [4, 20, 23]. In light of this, there have been earlier efforts to redesign an LSM-tree based KV store for PM systems [4, 23]. However, searching for a new design for KV stores based on a hybrid system of PM and disks, in which PM carries a role that is more than just a large memory write buffer or read cache, is also essential to achieve even better performance.

In this paper, we investigate how to leverage PM to en-

hance the performance of KV stores. We present a novel KV store, the *Single-Level Merge DB* (SLM-DB), which takes advantage of both the B+-tree index and the LSM-tree approach by making the best use of fast PM. Our proposed SLM-DB achieves high read performance as well as high write performance with low write amplification and near-optimal read amplification. In SLM-DB, we exploit PM to maintain a B+-tree for indexing KVs. Using the persistent B+-tree index, we can accelerate the search of a key (without depending on Bloom filters). To maintain high write throughput, we adopt the LSM-tree approach to stage inserted KV pairs in a PM resident memory buffer. As an inserted KV pair is persisted immediately in the PM buffer, we can also eliminate the write ahead log completely while providing strong data durability.

In SLM-DB, KV pairs are stored on disks with a *single-level* organization. Since SLM-DB can utilize the B+-tree for searches, it has no need to keep the KV pairs in sorted order, which significantly reduces write amplification. However, obsolete KV pairs should be garbage collected. Moreover, SLM-DB needs to provide some degree of sequentiality of KV pairs stored on disks in order to provide reasonable performance for range queries. Thus, the selective compaction scheme, which only performs restricted *merge* of the KV pairs organized in the single level, is devised for SLM-DB.

The main contributions of this work are as follows:

- We design a single-level KV store that retains the benefit of high write throughput from the LSM-tree approach and integrate it with a persistent B+-tree for indexing KV pairs. In addition, we employ a PM resident memory buffer to eliminate disk writes of recently inserted KV pairs to the write ahead log.
- For selective compaction, we devise three compaction candidate selection schemes based on 1) the live-key ratio of a data file, 2) the leaf node scans in the B+-tree, and 3) the degree of sequentiality per range query.
- We implement SLM-DB based on LevelDB and also integrate it with a persistent B+-tree implementation [22]. SLM-DB is designed such that it can keep the B+-tree and the single-level LSM-tree consistent on system failures, providing strong crash consistency and durability guarantees. We evaluate SLM-DB using the db_bench microbenchmarks [3] and the YCSB [17] for real world workloads. Our extensive experimental study demonstrates that in our default setup, compared to LevelDB, SLM-DB provides up to 1.96 and 2.22 times higher read and write throughput, respectively, and shows comparable range query performance, while it incurs only 39% of LevelDB's disk writes on average.

The rest of this paper is organized as follows. Section 2 discusses LSM-trees with the issue of slow read performance and high read/write amplification and also discusses PM technologies for KV stores. Section 3 presents the design and implementation of SLM-DB, Section 4 discusses how KV store operations are implemented in SLM-DB, and Section 5 discusses the recovery of SLM-DB on system failures. Section 6 evaluates the performance of SLM-DB and presents our experimental results, and Section 7 discusses issues of PM cost and parallelism. Section 8 discusses related work, and finally Section 9 concludes the paper.

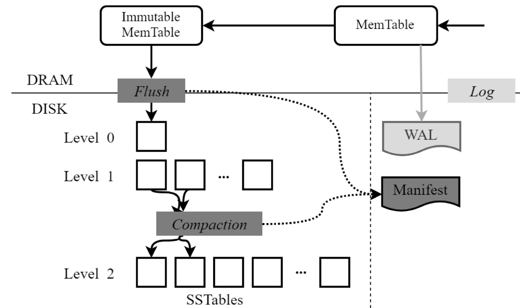


Figure 1: LevelDB architecture.

2 Background and Motivation

In this section, we first discuss an LSM-tree based KV store and its challenging issues by focusing on LevelDB [3]. Other LSM-tree based KV stores such as RocksDB [8] are similarly structured and have similar issues. We then discuss considerations for using PM for a KV store.

2.1 LevelDB

LevelDB is a widely used KV store inspired by Google's Bigtable [15], which implements the Log-Structured Merge-tree (LSM-tree) [33]. LevelDB supports basic KV store operations of *put*, which adds a KV pair to a KV store, *get*, which returns the associated value for a queried key, and *range query*, which returns all KV pairs within a queried range of keys by using *iterators* that scan all KV pairs. In LevelDB's implementation, the LSM-tree has two main modules, *MemTable* and *Immutable MemTable* that reside in DRAM and multiple levels of *Sorted String Table* (SSTable) files that reside in persistent storage (i.e., disks), as shown in Figure 1.

The memory components of MemTable and Immutable MemTable are basically sorted skiplists. MemTable buffers newly inserted KV pairs. Once MemTable becomes full, LevelDB makes it an Immutable MemTable and creates a new MemTable. Using a background thread, it flushes recently inserted KV pairs in the Immutable MemTable to the disk as an on-disk data structure SSTable where sorted KV pairs are stored. Note that the deletion of a KV pair is treated as an update as it places a deletion marker.

Table 1: Locating overhead breakdown (in microseconds) of a read operation in LevelDB

KV store	File search	Block search	Bloom filter	Unnecessary block read
LevelDB w/o BF	1.28	19.99	0	40.62
LevelDB w BF	1.33	19.38	7.22	13.58
SLM-DB	1.32		0	0

During the above insertion process, for the purpose of recovery from a system crash, a new KV pair must first be appended to the write ahead log (WAL) before it is added to MemTable. After KV pairs in Immutable MemTable are finally dumped into the disk, the log is deleted. However, by default, LevelDB does not commit KV pairs to the log due to the slower write performance induced by the `fsync()` operations for commit. In our experiments when using a database created by inserting 8GB data with a 1KB value size, the write performance drops by more than 12 times when `fsync()` is enabled for WAL. Hence, it trades off durability and consistency against higher performance.

For the disk component, LevelDB is organized as multiple levels, from the lowest level L_0 to the highest level L_k. Each level, except L_0, has one or more sorted SSTable files in which key ranges of the files in the same level do not overlap. Each level has limited capacity, but a higher level can contain more SSTable files such that the capacity of a level is generally around 10 times larger than that of its previous level.

To maintain such hierarchical levels, when the size of a level L_x grows beyond its limit, a background *compaction* thread selects one SSTable file in L_x. It then moves the KV pairs in that file to the next level L_{x+1} by performing a *merge sort* with all the SSTable files that overlap in level L_{x+1}. When KV pairs are being sorted, if the same key exists, the value in L_{x+1} is overwritten by that in L_x, because the lower level always has the newer value. In this way, it is guaranteed that keys stored in SSTable files in one level are unique. Compaction to L_0 (i.e., flushing recently inserted data in Immutable MemTable to L_0) does not perform merge sort in order to increase write throughput, and thus, SSTables in L_0 can have overlapped key ranges. In summary, using compaction, LevelDB not only keeps SSTable files in the same level sorted to facilitate fast search, but it also collects garbage in the files.

LevelDB also maintains the SSTable metadata of the current LSM-tree organization in a file referred to as MANIFEST. The metadata includes a list of SSTable files in each level and a key range of each SSTable file. During the compaction process, changes in the SSTable metadata such as a deleted SSTable file are captured. Once the compaction is done, the change is first logged in the MANIFEST file, and then obsolete SSTable files are deleted. In this way, when the system crashes even during compaction, LevelDB can return to a consistent KV store after recovery.

2.2 Limitations of LevelDB

Slow read operations For read operations (i.e., point queries), LevelDB first searches a key in MemTable and then Immutable MemTable. If it fails to find the key in the memory components, it searches the key in each level from the lowest one to the highest one. For each level, LevelDB needs to first find an SSTable file that may contain the key by a binary search based on the starting keys of SSTable files in that level. When such an SSTable file is identified, it performs another binary search on the SSTable file index, which stores the information about the first key of each 4KB data block in the file. Thus, a read operation requires at least two block reads, one for the index block and the other for the data block. However, when the data block does not include that key, LevelDB needs to check the next level again, until it finds the key or it reaches the highest level. To avoid unnecessary block reads and reduce the search cost, LevelDB uses a Bloom filter for each block.

Table 1 presents the overhead breakdown (in microseconds) for locating a KV pair in LevelDB with and without a Bloom filter for a random read operation. For the results, we measure the read latency of LevelDB for the random read benchmark in db_bench [3] (which are microbenchmarks built in LevelDB). In the experiments, we use 4GB DRAM and run a random read workload right after creating a database by inserting 20GB data with a 1KB value size (without waiting for the compaction process to finish, which is different from the experiments in Section 6.3). The details of our experimental setup are discussed in Section 6.

The locating overhead per read operation includes time spent to search an SSTable file that contains the key (i.e., "File search"), to find a block in which the key and its corresponding value are stored within the file (i.e., "Block search"), and to check the Bloom filter (BF). Moreover, included in the locating overhead is time for, what we refer to as the "Unnecessary block read". Specifically, this refers to the time to read blocks unnecessarily due to the multi-level search based on the SSTable index where BF is not used and, where BF is used, the time to read the false positive blocks. As shown in the table, with our proposed SLM-DB using the B+-tree index, which we discuss in detail later, the locating overhead can be significantly reduced to become almost negligible. In the above experiment, the average time of reading and processing a data block for the three KV stores is 682 microseconds.

Figure 2 shows the overhead for locating a KV pair for a random read operation over varying value sizes. The figure shows the ratio of the locating overhead to the total read operation latency. We observe that the overhead increases as the size of the value increases. When a Bloom filter is used in LevelDB, the locating overhead becomes relatively smaller, but the overhead with a Bloom filter is still as high as up to 36.66%. In the experiment, the random read workload is exe-

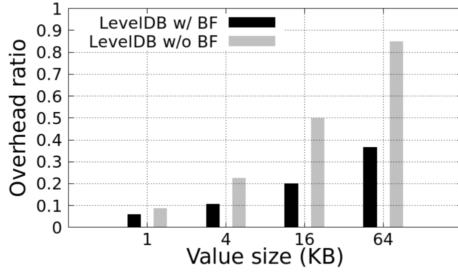


Figure 2: Overhead for locating a KV pair for different size of values as a fraction of the read operation latency.

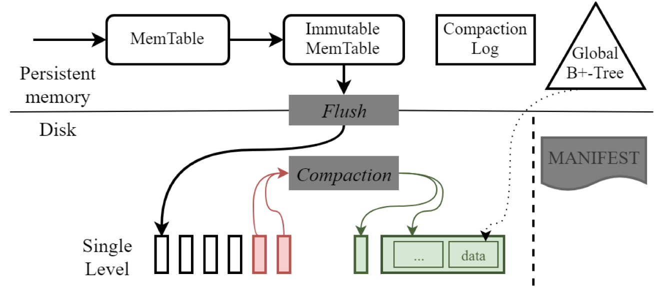


Figure 3: SLM-DB architecture.

cuted while the compaction of SSTable files from the lowest to the highest levels is in progress. With a larger value size, the number of files in multiple levels that LevelDB needs to check to see if a given key exists, and consequently the number of queries to a Bloom filter for the key, increases. Therefore, for a 64KB value size, LevelDB with a Bloom filter shows 6.14 times higher overhead largely incurred by unnecessary block reads, compared to a 1KB value size.

High write and read amplification Well-known issues of any LSM-tree based KV store are high write and read amplification [30, 31, 35, 40]. It maintains hierarchical levels of sorted files on a disk while leveraging sequential writes to the disk. Therefore, an inserted KV needs to be continuously merge-sorted and written to the disk, moving toward the highest level, via background compaction processes. For an LSM-tree structure with level k, the write amplification ratio, which is defined as the ratio between the total amount of data written to disk and the amount of data requested by the user, can be higher than $10 \times k$ [30, 31, 40].

The read amplification ratio, which is similarly defined as the ratio between the total amount of data read from a disk and the amount of data requested by the user, is high by nature in an LSM-tree structure. As discussed above, the cost of a read operation is high. This is because LevelDB may need to check multiple levels for a given key. Moreover, to find the key in an SSTable file at a level, it not only reads a data block but also an index block and a Bloom filter block, which can be much larger than the size of the KV pair [30].

2.3 Persistent Memory

Emerging persistent memory (PM) such as phase change memory (PCM) [39], spin transfer torque MRAM [21], and 3D XPoint [1] is byte-addressable and nonvolatile. PM will be connected via the memory bus rather than the block interface and thus, the failure atomicity unit (or granularity) for write to PM is generally expected to be 8 bytes [29, 42]. When persisting a data structure in PM, which has a smaller failure atomicity unit compared to traditional storage devices, we must ensure that the data structure remains consistent even when the system crashes. Thus, we need to carefully update or change the data structure by ensuring the memory write ordering.

However, in modern processors, memory write operations may be reordered in cache line units to maximize the memory bandwidth. In order to have ordered memory writes, we need to explicitly make use of expensive memory fence and cache flush instructions (`CLFLUSH` and `MFENCE` in Intel x86 architecture) [16, 22, 23, 29, 34, 42]. Moreover, if the size of the data written to PM is larger than 8 bytes, the data structure can be partially updated in system failure, resulting in an inconsistent state after recovery. In this case, it is necessary to use well-known techniques like logging and Copy-on-Write (CoW). Thus, a careful design is required for data structures persisted in PM.

PM opens new opportunities to overcome the shortcomings of existing KV stores. There has been a growing interest to utilize PM for KV stores [4, 23, 41]. LSM-tree based KV stores have been redesigned for PM [4, 23]. However, it is also important to explore new designs for PM based KV stores. In this work, we investigate a design of KV stores, which employs an index persisted in PM.

3 Single-Level Merge DB (SLM-DB)

This section presents the design and implementation of our Single-Level Merge DB (SLM-DB). Figure 3 shows the overall system architecture of SLM-DB. SLM-DB leverages PM to store MemTable and Immutable MemTable. The persistent MemTable and Immutable MemTable allow us to eliminate the write ahead log (WAL), providing stronger durability and consistency upon system failures. SLM-DB is organized as a single level L_0 of SSTable files, unlike LevelDB, hence the name Single Level Merge DB (SLM-DB). Thus, SLM-DB does not rewrite KV pairs stored on disks to merge them with pairs in the lower level, which can occur multiple times. Having the persistent memory component and single-level disk component, write amplification can be reduced significantly.

To expedite read operations on a single-level organization of SSTable files, SLM-DB constructs a persistent B+-tree index. Since the B+-tree index is used to search a KV pair stored on a disk, there is no need to fully sort KV pairs in the level, in contrast with most LSM-tree based KV stores.

Algorithm 1 Insert(*key*, *value*, *prevNode*)

```
1: curNode := NewNode(key, value);
2: curNode.next := prevNode.next;
3: mfence();
4: clflush(curNode);
5: mfence();
6: prevNode.next := curNode;
7: mfence();
8: clflush(prevNode.next);
9: mfence();
```

However, obsolete KV pairs in the SSTable files that have been updated by fresh values should be deleted to avoid disk space waste. Moreover, SLM-DB needs to maintain a sufficient level of sequentiality of KVs (i.e., a degree of how well KV pairs are stored in sorted order) in SSTables so that it can provide reasonable range query performance. Therefore, a selective compaction scheme, which selectively merges SSTables, is integrated with SLM-DB. Also, to keep the B+-tree and (single-level) LSM-tree consistent on system failures, the state of on-going compaction needs to be backed up by the compaction log stored in PM.

We implement SLM-DB based on LevelDB (version 1.20). We inherit the memory component implementation of MemTable and Immutable MemTable with modifications to persist them in PM. We keep the on-disk data structure of SSTable and the file format as well as the multiple SSTable file compaction (i.e., merge-sort) scheme. We also utilize the LSM-tree index structure, which maintains a list of valid SSTable files and SSTable file metadata, the mechanism to log any change in the LSM-tree structure to the MANIFEST file, and the recovery scheme of LevelDB. We completely change the random read and range query operations of the LevelDB implementation using a persistent B+-tree.

Among the many persistent B-tree implementations, we use the FAST and FAIR B-tree [22][1] for SLM-DB. Particularly, FAST and FAIR B-tree was shown to outperform other state-of-the-art persistent B-trees in terms of range query performance because it keeps all keys in a sorted order. It also yields the highest write throughput by leveraging the memory level parallelism and the ordering constraints of dependent store instructions.

3.1 Persistent MemTable

In SLM-DB, MemTable is a persistent skiplist. Note that a persistent skiplist has been discussed in previous studies [22, 23]. Skiplist operations such as insertion, update, and deletion can be done using an atomic 8-byte write operation. Algorithm 1 shows the insertion process to the lowest level of a skiplist. To guarantee the consistency of KVs in MemTable, we first persist a new node where its next pointer is set by calling memory fence and cacheline flush instructions. We then update the next pointer, which is 8 bytes, of its previous node and persist the change. Updating an existing KV pair in MemTable is done in a similar way, without in-place update of a value (similar to LevelDB's MemTable update operation). By having the PM resident MemTable, SLM-DB has no need to depend on WAL for data durability. Similarly, no consistency guarantee mechanism is required for higher levels of a skiplist as they can be reconstructed easily from the lowest level upon system failures.

3.2 B+-tree Index in PM

To speed up the search of a KV pair stored in SSTables, SLM-DB employs a B+-tree index. When flushing a KV pair in Immutable MemTable to an SSTable, the key is inserted in the B+-tree. The key is added to a leaf node of the B+-tree with a pointer that points to a PM object that contains the location information about where this KV pair is stored on the disk. The location information for the key includes an SSTable file ID, a block offset within the file, and the size of the block.

If a key already exists in the B+-tree (i.e., update), a fresh value for the key is written to a new SSTable. Therefore, a new location object is created for the key, and its associated pointer in the B+-tree leaf node is updated to point to the new PM object in a failure-atomic manner. If a deletion marker for a key is inserted, the key is deleted from the B+-tree. Persistent memory allocation and deallocation for location objects is managed by a persistent memory manager such as PMDK [5], and obsolete location objects will be garbage collected by the manager. Note that SLM-DB supports string-type keys like LevelDB does and that the string-type key is converted to an integer key when it is added to the B+-tree.

Building a B+-tree In SLM-DB, when Immutable MemTable is flushed to L_0, KV pairs in Immutable MemTable are inserted in the B+-tree. For a flush operation, SLM-DB creates two background threads, one for file creation and the other for B+-tree insertion.

In the file creation thread, SLM-DB creates a new SSTable file and writes KV pairs from Immutable MemTable to the file. Once the file creation thread flushes the file to the disk, it adds all the KV pairs stored on the newly created file to a queue, which is created by a B+-tree insertion thread. The B+-tree insertion thread processes the KV pairs in the queue one by one by inserting them in the B+-tree. Once the queue becomes empty, the insertion thread is done. Then, the change of the LSM-tree organization (i.e., SSTable metadata) is appended to the MANIFEST file as a log. Finally, SLM-DB deletes Immutable MemTable.

Scanning a B+-tree SLM-DB provides an iterator, which can be used to scan all the keys in the KV store, in a way similar to LevelDB. Iterators support `seek`, `value`, and `next` methods. The `seek(k)` method positions an iterator in the KV store such that the iterator points to key `k` or the smallest

[1] Source codes are available at https://github.com/DICL/FAST_FAIR

key larger than `k` if `k` does not exist. The `next()` method moves the iterator to the next key in the KV store and the `value()` method returns the value of the key, currently pointed to by the iterator.

In SLM-DB, a B+-tree iterator is implemented to scan keys stored in SSTable files. For the `seek(k)` method, SLM-DB searches key `k` in the B+-tree to position the iterator. In the FAST+FAIR B+-tree, keys are sorted in leaf nodes, and leaf nodes have a sibling pointer. Thus, if `k` does not exist, it can easily find the smallest key that is larger than `k`. Also, the `next()` method is easily supported by moving the iterator to the next key in B+-tree leaf nodes. For the `value()` method, the iterator finds the location information for the key and reads the KV pair from the SSTable.

3.3 Selective Compaction

SLM-DB supports a selective compaction operation in order to collect garbage of obsolete KVs and improve the sequentiality of KVs in SSTables. For selective compaction, SLM-DB maintains a *compaction candidate list* of SSTables. A background compaction thread is basically scheduled when some changes occur in the organization of SSTable files (for example, by a flush operation), and there are a large number of seeks to a certain SSTable (similar to LevelDB). In SLM-DB, it is also scheduled when the number of SSTables in the compaction candidate list is larger than a certain threshold. When a compaction thread is executed, SLM-DB chooses a subset of SSTables from the candidate list as follows. For each SSTable s in the list, we compute the overlapping ratio of key ranges between s and each t of the other SSTables in the list as $\frac{MIN(s_p,t_q)-MAX(s_1,t_1)}{MAX(s_p,t_q)-MIN(s_1,t_1)}$, where the key ranges of s and t are $[s_1,\dots s_p]$ and $[t_1,\dots t_q]$, respectively. Note that if the computed ratio is negative, then s and t do not overlap and the ratio is set to zero. We compute the total sum of overlapping ratios for s. We then decide to compact an SSTable s' with the maximum overlapping ratio value with SSTables in the list, whose key ranges are overlapped with s'. Note that we limit the number of SSTables that are simultaneously merged so as not to severely disturb foreground user operations.

The compaction process is done by using the two threads for file creation and B+-insertion described above. However, when merging multiple SSTable files, we need to check if each KV pair in the files is valid or obsolete, which is done by searching the key in the B+-tree. If it is valid, we merge-sort it with the other valid KV pairs. If the key does not exist in the B+-tree or the key is currently stored in some other SSTable, we drop that obsolete KV pair from merging to a new file.

During compaction, the file creation thread needs to create multiple SSTable files unlike flushing Immutable MemTable to L_0. The file creation thread creates a new SSTable file (of a fixed size) as it merge-sorts the files and flushes the new file to disk. It then adds all the KV pairs included in the new file to the queue of the B+-tree insertion thread. The file creation thread starts to create another new file, while the insertion thread concurrently updates the B+-tree for each KV pair in the queue. This process continues until the creation of merge-sorted files for compaction is completed. Finally, after the B+-tree updates for KV pairs in the newly created SSTable files are done, the change of SSTable metadata is committed to the MANIFEST file and the obsolete SSTable files are deleted. Note that SLM-DB is implemented such that B+-tree insertion requests for KVs in the new SSTable file are immediately queued right after file creation, and they are handled in order. In this way, when updating the B+-tree for a KV in the queue, there is no need to check the validity of the KV again.

To select candidate SSTables for compaction, SLM-DB implements three selection schemes based on *the live-key ratio* of an SSTable, *the leaf node scans* in the B+-tree, and *the degree of sequentiality per range query*. For the selection based on the live-key ratio of an SSTable, we maintain the ratio of valid KV pairs to all KV pairs (including obsolete ones) stored in each SSTable. If the ratio for an SSTable is lower than the threshold, called the `live-key threshold`, then the SSTable contains too much garbage, which should be collected for better utilization of disk space. For each SSTable s, the total number of KV pairs stored in s is computed at creation, and initially the number of valid KV pairs is equal to the total number of KV pairs in s. When a key stored in s is updated with a fresh value, the key with the fresh value will be stored in a new SSTable file. Thus, when we update a pointer to the new location object for the key in the B+-tree, we decrease the number of valid KV pairs in s. Based on these two numbers, we can compute the live-key ratio of each SSTable.

While the goal of the live-key ratio based selection is to collect garbage on disks, the selection based on the leaf node scans in the B+-tree attempts to improve the sequentiality of KVs stored in L_0. Whenever a background compaction is executed, it invokes a leaf node scan, where we scan B+-tree leaf nodes for a certain fixed number of keys in a round-robin fashion. During the scan, we count the number of unique SSTable files, where scanned keys are stored. If the number of unique files is larger than the threshold, called the `leaf node threshold`, we add those files to the compaction candidate list. In this work, the number of keys to scan for a leaf node scan is decided based on two factors, the average number of keys stored in a single SSTable (which depends on the size of a value) and the number of SSTables to scan at once.

For the selection based on the degree of sequentiality per range query, we divide a queried key range into several sub-ranges when operating a range query. For each sub-range, which consists of a predefined number of keys, we keep track of the number of unique files accessed. Once the range query operation is done, we find the sub-range with the maximum number of unique files. If the number of unique

files is larger than the threshold, called the `sequentiality degree threshold`, we add those unique files to the compaction candidate list. This feature is useful in improving sequentiality especially for requests with Zipfian distribution (like YCSB [17]) where some keys are more frequently read and scanned.

For recovery purposes, we basically add the compaction candidate list, the total number of KV pairs, and the number of valid KV pairs for each SSTable to the SSTable metadata (which is logged in the MANIFEST file). In SLM-DB, the compaction and flush operations update the B+-tree. Therefore, we need to maintain a compaction log persisted in PM for those operations. Before starting a compaction/flush operation, we create a compaction log. For each key that is already stored in some SSTable but is written to a new file, we add a tuple of the key and its old SSTable file ID to the compaction log. Also, for compaction, we need to keep track of the list of files merged by the on-going compaction. The information about updated keys and their old file IDs will be used to recover a consistent B+-tree and live-key ratios of SSTables if there is a system crash. After the compaction/flush is completed, the compaction log will be deleted. Note that some SSTable files added to the compaction candidate list by the selection based on the leaf node scans and the degree of sequentiality per range query may be lost if the system fails before they are committed to the MANIFEST file. However, losing some candidate files does not compromise the consistency of the database. The lost files will be added to the list again by our selection schemes.

4 KV Store Operations in SLM-DB

Put: To put a KV pair to a KV store, SLM-DB inserts the KV pair to MemTable. The KV pair will eventually be flushed to an SSTable in L_0. The KV pair may be compacted and written to a new SSTable by the selective compaction of SLM-DB.

Get: To get a value for a given key k from a KV store, SLM-DB searches MemTable and Immutable MemTable in order. If k is not found, it searches k in the B+-tree, locates the KV pair on disk (by using the location information pointed to by the B+-tree for k), reads its associated value from an SSTable and returns the value. If SLM-DB cannot find k in the B+-tree, it returns "not exist" for k, without reading any disk block.

Range query: To perform a range query, SLM-DB uses a B+-tree iterator to position it to the starting key in the B+-tree by using the `seek` method and then, scans a given range using `next` and `value` methods. KV pairs inserted to SLM-DB can be found in MemTable, Immutable MemTable or one of the SSTables in L_0. Therefore, for the `seek` and `next` methods, the result of the B+-tree iterator needs to be merged with the results of the two iterators that search the key in MemTable and Immutable MemTable, respectively, to determine the final result, in a way similar to LevelDB.

"Insert if not exists" and **"Insert if exists"**: "Insert if not exists" [36], which inserts a key to a KV store only if the key does not exist, and "Insert if exists", which updates a value only for an existing key, are commonly used in a KV store. Update workloads such as YCSB Workload A [17] are usually performed on an existing key such that for a non-existing key, the KV store returns without inserting the key [9]. To support these operations, SLM-DB simply searches the key in the B+-tree to check for the existence of the given key. In contrast, most LSM-tree based KV stores must check multiple SSTable files to search the key in each level in the worst case.

5 Crash Recovery

SLM-DB provides a strong crash consistency guarantee for in-memory data persisted in PM, on-disk data (i.e. SSTables) as well as metadata on SSTables. For KV pairs recently inserted to MemTable, SLM-DB can provide stronger durability and consistency compared to LevelDB. In LevelDB, the write of data to WAL is not committed (i.e., `fsync()`) by default because WAL committing is very expensive and thus, some recently inserted or updated KVs may be lost on system failures [23, 26, 32]. However, in SLM-DB, the skiplist is implemented such that the linked list of the lowest level of the skiplist is guaranteed to be consistent with an atomic write or update of 8 bytes to PM, without any logging efforts. Therefore, during the recovery process, we can simply rebuild higher levels of the skiplist.

To leverage the recovery mechanism of LevelDB, SLM-DB adds more information to the SSTable metadata such as the compaction candidate list and the number of valid KV pairs stored for each SSTable along with the total number of KV pairs in the SSTable. The additional information is logged by the MANIFEST file in the same way as for the original SSTable metadata.

When recovering from a failure, SLM-DB performs the recovery procedure using the MANIFEST file as LevelDB does. Also, similar to NoveLSM [23], SLM-DB remaps the file that represents a PM pool and retrieves the root data structure that stores all pointers to other data structures such as MemTable, Immutable MemTable, and B+-tree through support from a PM manager such as PMDK [5]. SLM-DB will flush Immutable MemTable if it exists. SLM-DB also checks if there is on-going compaction. If so, SLM-DB must restart the compaction of the files that are found in the compaction log.

For flush, SLM-DB uses the information on an updated key with its old SSTable file ID to keep the number of valid KV pairs in each SSTable involved in the failed flush consistent. In case of compaction, it is possible that during the last failed compaction, the pointers in the B+-tree leaf nodes for

some subset of valid KV pairs have been committed to point to new files. However, when the system restarts, the files are no longer valid as they have not been committed to the MANIFEST file. Based on the information of keys and their old SSTable file IDs, SLM-DB can include these valid KVs and update the B+-tree accordingly during the restarted compaction. Note that for PM data corruption caused by hardware errors, the recovery and fault-tolerance features such as checksum and data replication can be used [7].

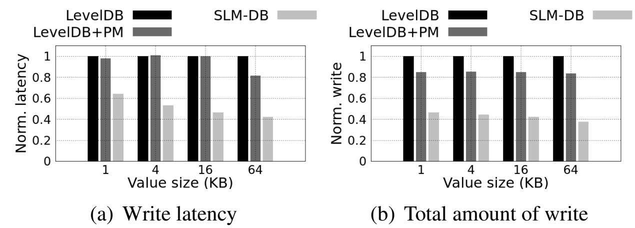
6 Experimental Results

6.1 Methodology

In our experiments, we use a machine with two Intel Xeon Octa-core E5-2640V3 processors (2.6Ghz) and Intel SSD DC S3520 of 480GB. We disable one of the sockets and its memory module and only use the remaining socket composed of 8 cores with 16GB DRAM. For the machine, Ubuntu 18.04 LTS with Linux kernel version 4.15 is used.

When running both LevelDB and SLM-DB, we restrict the DRAM size to 4GB by using the mem kernel parameter. As PM is not currently available in commercial markets, we emulate PM using DRAM as in prior studies [23, 29, 41]. We configure a DAX enabled ext4 file system and employ PMDK [5] for managing a memory pool of 7GB for PM. In the default setting, the write latency to PM is set to 500ns (i.e., around 5 times higher write latency compared to DRAM [23] is used). PM write latency is applied for data write persisted to PM with memory fence and cacheline flush instructions and is emulated by using `Time Stamp Counter` and spinning for a specified duration. No extra read latency to PM is added (i.e., the same read latency as DRAM is used) similar to previous studies [23, 41]. We also assume that the PM bandwidth is the same as that of DRAM.

We evaluate the performance of SLM-DB and compare its performance with that of LevelDB (version 1.20) over varying value sizes. For all experiments, data compression is turned off to simplify analysis and avoid any unexpected effects as in prior studies [23, 30, 35]. The size of MemTable is set to 64MB, and a fixed key size of 20 bytes is used. Note that all SSTable files are stored on an SSD. For LevelDB, default values for all parameters are used except the MemTable size and a Bloom filter (configured with 10 bits per key) is enabled. In all the experiments of LevelDB, to achieve better performance, we do not commit the write ahead log trading off against data durability. For SLM-DB, the `live-key threshold` is set to 0.7. If we increase this threshold, SLM-DB will perform garbage collection more actively. For the leaf node scan selection, we scan the average number of keys stored in two SSTable files and the `leaf node threshold` is set to 10. Note that the average number of keys stored in an SSTable varies depending on the value size. For selection based on the sequentiality degree per range query, we divide a queried key range into sub-ranges of 30 keys each, and the `sequentiality degree threshold` is set to 8. If we increase the `leaf node threshold` and `sequentiality degree threshold`, SLM-DB will perform less compaction. For the results, the average value of three runs is presented.



(a) Write latency (b) Total amount of write

Figure 4: Random write performance comparison.

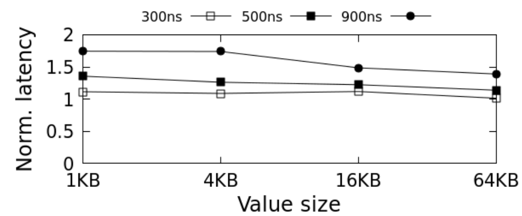
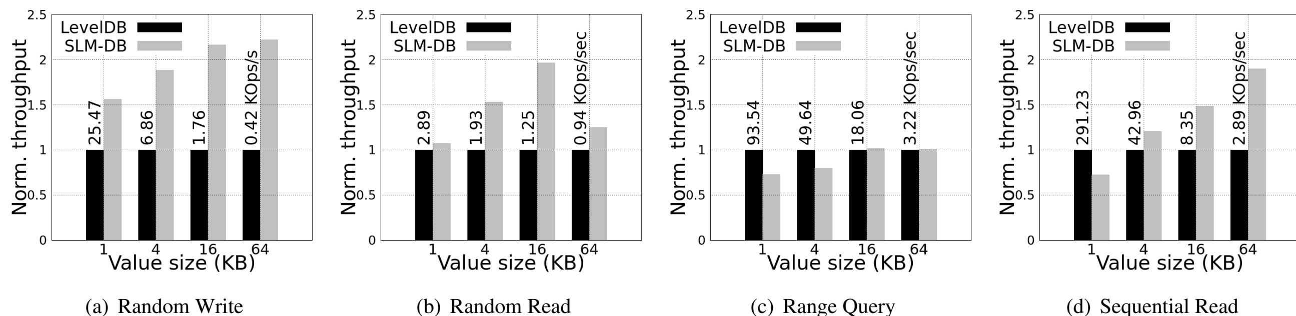


Figure 5: SLM-DB write latency over various PM write latencies, normalized to that with DRAM write latency.

To evaluate the performance of SLM-DB, we use the db_bench benchmarks [3] as microbenchmarks and the YCSB [17] as real world workload benchmarks. The benchmarks are executed as single-threaded workloads as LevelDB (upon which SLM-DB is implemented) is not optimized for multi-threaded workloads, a matter that we elaborate on in Section 7. In both of the benchmarks, for each run, a random write workload creates the database by inserting 8GB data unless otherwise specified, where N write operations are performed in total. Then, each of the other workloads performs $0.2 \times N$ of its own operations (i.e., 20% of the write operations) against the database. For example, if 10M write operations are done to create the database, the random read workload performs 2M random read operations. Note that the size of the database initially created by the random write workload is less than 8GB in db_bench, since the workload overwrites (i.e., updates) some KV pairs.

6.2 Using a Persistent MemTable

To understand the effect of using a PM resident MemTable, we first investigate the performance of a modified version of LevelDB, i.e., LevelDB+PM, which utilizes the PM resident MemTable without the write ahead log as in SLM-DB. Figure 4 shows the performance of LevelDB, LevelDB+PM, and SLM-DB for the random write workload from db_bench over various value sizes. In Figures 4(a) and 4(b), the write latency and total amount of data written to disk normalized to those of LevelDB, respectively, are presented.



(a) Random Write (b) Random Read (c) Range Query (d) Sequential Read

Figure 6: Throughput of SLM-DB normalized to LevelDB with the same setting for db_bench.

As in the figures, in general, the write latency of LevelDB+PM is similar to that of LevelDB, but the total amount of write to disk is reduced by 16% on average as no write ahead log is used. When a large value size is used as in the case of 64KB, the write latency of LevelDB+PM is reduced by 19%. LevelDB+PM also achieves stronger durability of data as inserted KVs are persisted immediately in MemTable. For SLM-DB, the write latency and total amount of data written are reduced by 49% and 57%, compared to LevelDB, on average. This is because SLM-DB further reduces write amplification by organizing SSTables in a single level and performing restricted compaction.

Figure 5 presents the effects of PM write latency on the write performance of SLM-DB. In the figure, the write operation latencies of SLM-DB with PM write latencies of 300, 500 and 900ns, normalized to that of SLM-DB with DRAM write latency, are presented for the random write workload of db_bench. In SLM-DB, as the PM write latency increases, the write performance is degraded by up to 75% when the 1KB value size is used. However, the effect of long PM write latency is diluted as the value size becomes larger.

6.3 Results with Microbenchmarks

Figure 6 shows the operation throughputs with SLM-DB for random write, random read, range query, and sequential read workloads, normalized to those with LevelDB. In the figures, the numbers presented on the top of the bars are the operation throughput of LevelDB in KOps/s. The range query workload scans short ranges with an average of 100 keys. For the sequential read workload, we sequentially read all the KV pairs in increasing order of key values on the entire KV store (which is created by a random write workload). For the random read, range query, and sequential read workloads, we first run a random write workload to create the database, and then wait until the compaction process is finished on the database before performing their operations.

From the results, we can observe the following:

- For random write operations, SLM-DB provides around 2 times higher throughput than LevelDB on average over all the value sizes. This is achieved by significantly reducing the amount of data written to disk for compaction. Note that in our experiments, the overhead of inserting keys to the B+-tree is small and also, the insertion to B+-tree is performed by a background thread. Therefore, the insertion overhead has no effect on the write performance of SLM-DB.
- For random read operations, SLM-DB shows similar or better performance than LevelDB depending on the value size. As discussed in Section 2.2, the locating overhead of LevelDB is not so high when the value size is 1KB. Thus, the read latency of SLM-DB is only 7% better for 1KB values. As the value size increases, the performance difference between SLM-DB and LevelDB increases due to the efficient search of the KV pair using the B+-tree index in SLM-DB. However, when the value size becomes as large as 64KB, the time spent to read the data block from disk becomes long relative to that with smaller value sizes. Thus, the performance difference between SLM-DB and LevelDB drops to 25%.
- For short range query operations, LevelDB with full sequentiality of KVs in each level can sequentially read KV pairs in a given range, having better performance for 1KB and 4KB value sizes. In case of a 1KB value size, a 4KB data block contains 4 KV pairs on average. Therefore, when one block is read from disk, the block is cached in memory and then, LevelDB benefits from cache hits on scanning the following three keys without incurring any disk read. However, in order to position a starting key, a range query operation requires a random read operation, for which SLM-DB provides high performance. Also, it takes a relatively long time to read a data block for a large value size. Thus, even with less sequentiality, SLM-DB shows comparable performance for range queries. Note that when the scan range becomes longer, the performance of SLM-DB generally improves. For example, we ran additional experiments of the range query workload with an average of 1,000 key ranges for the 4KB value size. In this case, SLM-DB throughput was 57.7% higher than that of LevelDB.
- For the sequential read workload to scan all KV pairs,

Table 2: db_bench latency of SLM-DB in microseconds/Op

Value size	1KB	4KB	16KB	64KB
Random Write	25.14	77.44	262.41	1065.68
Random Read	323.56	338.25	406.94	851.98
Range Query	14.68	25.15	54.41	307.71
Sequential Read	4.74	19.35	80.74	182.28

SLM-DB achieves better performance than LevelDB, except for the 1KB value size.

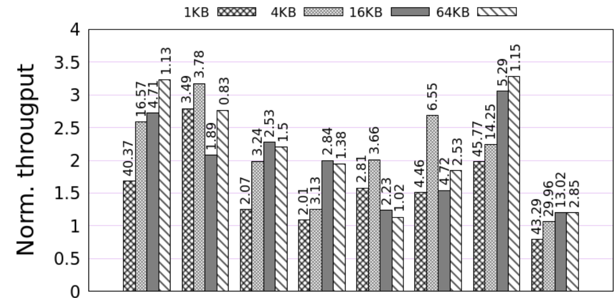
While running the random read, range query, and sequential read workloads, LevelDB and SLM-DB perform additional compaction operations. We measure the total amount of disk write of LevelDB and SLM-DB from the creation of a database to the end of each workload. By selectively compacting SSTables, the total amount of disk write of SLM-DB is only 39% of that of LevelDB on average for the random read, range query, and sequential read workloads. Note that for LevelDB with db_bench workloads, the amount of write for WAL is 14% of its total amount of write on average.

Recall that SLM-DB adds an SSTable to the compaction candidate list for garbage collection only when more than a certain percentage of KV pairs stored in the SSTable are obsolete, and it performs selective compaction for SSTables with poor sequentiality. We analyze the space amplification of SLM-DB for the random write workload in db_bench. Over all the value sizes, the size of the database on disk for SLM-DB is up to 13% larger than that for LevelDB. Finally, we show the operation latency performance of SLM-DB in Table 2.

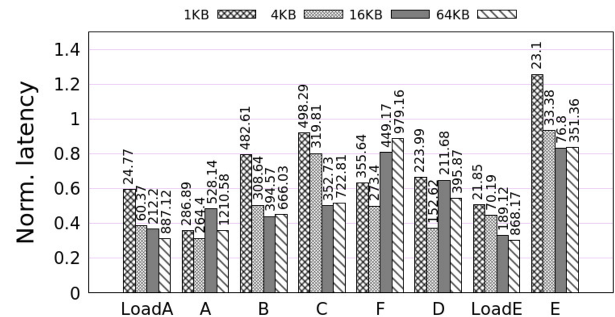
6.4 Results with YCSB

YCSB consists of six workloads that capture different real world scenarios [17]. To run the YCSB workloads, we modify db_bench to run YCSB workload traces for various value sizes (similar to [35]).

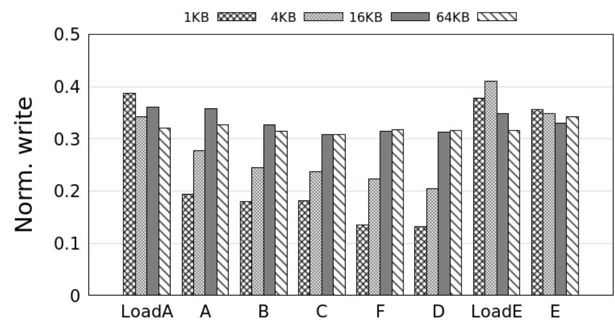
Figures 7(a), 7(b) and 7(c) show the operation throughput, latency, and the total amount of write with SLM-DB, normalized to those with LevelDB over the six YCSB workloads [17]. In Figures 7(a) and 7(b), the numbers presented on top of the bars are operation throughput in KOps/s and operation latency in microseconds/Op of SLM-DB, respectively. For each workload, the cumulative amount of write is measured when the workload finishes. For the results, we load the database for workload A by inserting KVs, and continuously run workload A, workload B, workload C, workload F, and workload D in order. We then delete the database, and reload the database to run workload E. Workload A performs 50% reads and 50% updates, Workload B performs 95% reads and 5% updates, Workload C performs 100% reads, and Workload F performs 50% reads and 50% read-modify-writes. For these workloads, Zipfian distribution is used. Workload D performs 95% reads for the latest keys



(a) Throughput



(b) Latency



(c) Total amount of write

Figure 7: YCSB performance of SLM-DB normalized to LevelDB with the same setting.

and 5% inserts. Workload E performs 95% range query and 5% inserts with Zipfian distribution.

In Figure 7(a), the throughputs of SLM-DB are higher than those of LevelDB for all the workloads over varying value sizes, except for workload E with a 1KB value size. For 4∼64KB value sizes, the performance of SLM-DB for workload E (i.e., short range queries) is 15.6% better than that of LevelDB on average due to the fast point query required for each range query and the selective compaction mechanism that provides some degree of sequentiality for KV pairs stored on disks. For workload A, which is composed of 50% reads and 50% updates, updating a value for a key is performed only when the key already exists in the database. Thus, this update is the "insert if exists" operation. For this operation, SLM-DB efficiently checks the existence

of a key through a B+-tree search. On the other hand, checking the existence of a key is an expensive operation in LevelDB. If the key does not exist, LevelDB needs to search the key at every level. Therefore, for workload A, SLM-DB achieves 2.7 times higher throughput than LevelDB does on average.

As shown in Figure 7(c), the total amount of write in SLM-DB is much smaller than that of LevelDB in all the workloads. In particular, with a 1KB value size, SLM-DB only writes 13% of the data that LevelDB writes to disk while executing up to workload D. Note that for LevelDB with YCSB workloads, the amount of write for WAL is 11% of its total amount of write on average.

6.5 Other Performance Factors

In the previous discussion, we mainly focused on how SLM-DB would perform for target workloads that we envision for typical KV stores. Also, there were parameter and scheme choices that were part of the SLM-DB design. Due to various limitations, we were not able to provide a complete set of discussion on these matters. In this section, we attempt to provide a sketch of some of these matters.

Effects of varying live-key ratios In the above experiments, the live-key ratio is set to 0.7. As the ratio increases, SLM-DB will perform garbage collection more aggressively. We run experiments of the random write and range query workloads of db_bench with a 1KB value size over varying live-key ratios of 0.6, 0.7, and 0.8. With ratio=0.7, the range query latency decreases by around 8%, while write latency increases by 17% due to more compaction compared to ratio=0.6. With ratio=0.8, the range query latency remains the same as that with ratio=0.7. However, with ratio=0.8, write performance is severely degraded (i.e., two times slower than ratio=0.6) because the live-key ratio selection scheme adds too many files to the candidate list, making SLM-DB stall for compaction. Compared to ratio=0.6, with ratio=0.7 and ratio=0.8, the database sizes (i.e., space amplification) decrease by 1.59% and 3.05%, whereas the total amounts of disk write increase by 7.5% and 12.09%, respectively.

Effects of compaction candidate selection schemes The selection schemes based on the leaf node scans and sequentiality degree per range query can improve the sequentiality of KVs stored on disks. Using YCSB Workload E, which is composed of a large number of range query operations, with a 1KB value size, we analyze the performance effects of these schemes by disabling them in turn. First, when we disable both schemes, the latency result becomes more than 10 times longer than with both schemes enabled. Next, we disable the sequentiality degree per range query, which is only activated by a range query operation, while keeping the leaf node scans for selection. The result is that there is a range query latency increase of around 50%. Finally, we flip the two selection schemes and disable the leaf node scans and

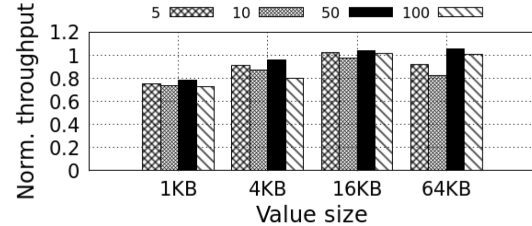


Figure 8: Range query performance of SLM-DB over various key ranges normalized to LevelDB with the same setting.

enable the sequentiality degree per range query scheme. In this case, the result is around a 15% performance degradation. This implies that selection based on the leaf node scans will play an important role for real world workloads that are composed of a mix of "point queries, updates and occasional scans" as described in the study by Sears and Ramakrishnan [36].

Short range query Figure 8 shows the range query performances of db_bench with SLM-DB over various key ranges, 5, 10, 50, and 100, normalized to those of LevelDB. For small key ranges such as 5 and 10, the performance trend over different value sizes is similar to that of the random read workload shown in Figure 6(b) as the range query operation depends on random read operations needed to locate the starting key.

Smaller value sizes We evaluate the performance of SLM-DB for a database with a 128 byte value size for random write, random read, and range query workloads in db_bench. Note that with this setting, the total number of write operations becomes so large that, for the range query workload, we choose to execute for only 1% of the write operations due to time and resource limitations. For these experiments, we find that write performance of SLM-DB is 36.22% lower than that of LevelDB. The reason behind this is PM write latency, where it is set to 500ns. With DRAM write latency, in fact, write performance of SLM-DB becomes 24.39% higher, while with 300ns PM write latency, it is only 6.4% lower than LevelDB. With small value sizes, we see the effect of PM write latency on performance. Even so, note that SLM-DB provides reasonable write performance with strong durability of data, considering that the performance of LevelDB with `fsync` enabled for WAL is more than 100 times lower than that with `fsync` disabled for WAL. For random read operations, SLM-DB improves performance by 10.75% compared to LevelDB. For range query operations, with the key range sizes of 50 and 100, performances of SLM-DB are 17.48% and 10.6% lower, respectively, than those of LevelDB. However, with the key range sizes of 5 and 10, SLM-DB becomes more than 3 times slower than LevelDB as LevelDB takes advantage of the cache hits brought about by the high sequentiality of KVs stored on disk.

LevelDB with additional B+-tree index We implement a version of LevelDB that has an additional B+-tree index stored in PM as SLM-DB. This version utilizes the B+-tree

index for random read operations and the B+-tree iterators for range query operations. We evaluate the performance of LevelDB with the B+-tree index over various value sizes using db_bench as in Section 6.3. For the random write workload, the performance of LevelDB with the B+-tree is almost the same as that of LevelDB. However, for the random read workload, the performance is almost the same as that of SLM-DB. For the range query workload, it shows 6.64% higher performance than LevelDB on average over all value sizes, as it leverages not only the full sequentiality of KVs in each level, but also the B+-tree iterator instead of multiple iterators of LevelDB (each of which iterates each level in an LSM-tree).

Effects of PM bandwidth In the above experiments, it is assumed that PM bandwidth is the same as that of DRAM (i.e., 8GB/s). We run experiments of SLM-DB with a 1KB value size over various PM bandwidths, 2GB/s, 5GB/s and 8GB/s. Note that for these experiments, we used a different machine with two Intel Xeon Octa-core E5-2620V4 processors (2.1Ghz) to decrease the memory bandwidth by thermal throttling.

Write performance of SLM-DB is affected not only by MemTable insertion of a KV pair but also by background compaction, which can stall write operations. For compaction, the performance degradation caused by lower PM bandwidths for B+-tree insertion is around 5%. However, the effect of PM bandwidth on compaction performance is negligible as file creation is the performance dominating factor. For MemTable insertion, performance with 2GB/s is degraded by 9% compared to that with 8GB/s. Therefore, the final write performance with 2GB/s is around 6% lower than that with 8GB/s. With 5GB/s, the performance of MemTable insertion is not degraded, showing final write performance similar to that with 8GB/s. We also find that with value sizes of 4, 16, and 64KB, write performance of SLM-DB is not affected by PM bandwidth.

For the random read workload, the time to search MemTable and Immutable MemTable and to query the B+-tree is very small comprising just 0.27% of the total time of a read operation for all PM bandwidths used in the experiments. Thus, the effect of PM bandwidth on read performance of SLM-DB is negligible. Note that in our experiments, PM and DRAM read latency is assumed to be the same. However, we believe this has only minor implications on the final read latency as the time to read data from PM during a read operation is very small compared to the time to read a block from disk.

Larger DBs We evaluate the performance of LevelDB and SLM-DB for a database that is created by inserting 20GB data over various value sizes using db_bench. Note that we use 4GB DRAM for the experiments. The write and read throughputs of SLM-DB are 2.48 and 1.34 times higher, respectively, than those of LevelDB (on average), while the range query throughput is 10% lower. Recall that for the database with 8GB data insertion (in Section 6.3), SLM-DB showed 1.96 and 1.45 times higher write and read throughput, respectively, and 11% lower range query throughput than LevelDB.

7 Discussion

Persistent memory cost SLM-DB utilizes extra PM for MemTable, Immutable MemTable, the B+-tree index, and compaction log. For MemTable and Immutable MemTable, its size is user configurable and constant. On the other hand, PM consumed for the B+-tree index depends on the value size, as for a fixed sized database, with a smaller value size, the number of records increases, which results in a larger B+-tree index. Finally, the compaction log size is very small relative to the size of the B+-tree index.

In our experiments of the database with 8GB data insertion, the total amount of PM needed for MemTable and Immutable Memtable is 128MB. For the B+-tree, a total of 26 bytes are used per key in the leaf nodes: (integer type) 8 bytes for the key, 8 bytes for the pointer, and 10 bytes for the location information. Thus, the total amount of PM needed for the B+-tree is dominantly determined by the leaf nodes. In particular, with the 1KB and 64KB value size, SLM-DB uses around 700MB and 150MB of PM, respectively. The cost of PM is expected to be cheaper than DRAM [20], and so we will be able to achieve high performance KV stores with a reasonably small extra cost for PM in many cases.

Parallelism SLM-DB is currently implemented as a modification of LevelDB. Thus, LevelDB and consequently, SLM-DB as well, has some limitations in efficiently handling multi-threaded workloads. In particular, for writes, there is a writer queue that limits parallel write operations, and for reads, a global lock needs to be acquired during the operations to access the shared SSTable metadata [11]. However, by design, SLM-DB can easily be extended for exploiting parallelism; a concurrent skiplist may replace our current implementation, lock-free search feature of FAST and FAIR B-tree for read operations may be exploited, and multi-threaded compaction can be supported [8]. We leave improving parallelism of SLM-DB as future work.

8 Related Work

KV stores utilizing PM have been investigated [4, 6, 23, 41]. HiKV assumes a hybrid memory system of DRAM and PM for a KV store, where data is persisted to only PM, eliminating the use of disks [41]. HiKV maintains a persistent hash index in PM to process read and write operations efficiently and also has a B+-tree index in DRAM to support range query operations. Thus, it needs to rebuild the B+-tree index when the system fails. Unlike HiKV, our work considers a system in which PM coexists with HDDs or SSDs sim-

ilar to NoveLSM [23]. Similarly to HiKV, it is possible for SLM-DB to have the B-tree index in DRAM and the hash table index in PM. However, for a hybrid system that has both PM and disks, the final read latency is not strongly affected by index query performance as disk performance dominates, unless the database is fully cached. Thus, in such a hybrid setting, using a hash table rather than a B-tree index does make a meaningful performance difference, even while paying for the additional overhead of keeping two index structures. pmemkv [6] is a key-value store based on a hybrid system of DRAM and PM, which has inner nodes of a B+-tree in DRAM and stores leaf nodes of the B+-tree in PM.

NoveLSM [23] and NVMRocks [4] redesign an LSM-tree based KV store for PM. NoveLSM proposes to have an immutable PM MemTable between the DRAM MemTable and the disk component, reducing serialization and deserialization costs. Additionally, there is a mutable PM MemTable, which is used along with the DRAM MemTable to reduce stall caused by compaction. Since the DRAM MemTable with WAL and the mutable PM MemTable are used together, the commit log for the DRAM MemTable and versions of keys need to be carefully maintained to provide consistency. When a large mutable PM MemTable is used, heavy writes are buffered in MemTable and flush operations from Immutable MemTable to disk will occur less frequently. However, to handle a database with a size larger than the total size of DRAM/PM MemTables and Immutable MemTables, KVs in Immutable MemTable will eventually need to be flushed to disk. In NVMRocks, the MemTable is persisted in PM, eliminating the logging cost like SLM-DB, and PM is also used as a cache to improve read throughput [4]. In both NoveLSM and NVMRocks, PM is also used to store SSTables. However, in our work, we propose a new structure of employing a persistent B+-tree index for fast query and a single level disk component of SSTable files with selective compaction, while leveraging the memory component similar to an LSM-tree. In general, SLM-DB can be extended to utilize a large PM MemTable [23], multiple Immutable MemTables [8], and a PM cache [4], orthogonally improving the performance of the KV store.

Optimization techniques to enhance the performance of an LSM-tree structure for conventional systems based on DRAM and disks have been extensively studied [10, 11, 30, 35, 37, 38, 40]. WiscKey provides optimized techniques for SSDs by separating keys and values [30]. In WiscKey, sorted keys are maintained in an LSM-tree while values are stored in a separate value log without hierarchical levels similar to SLM-DB, reducing I/O amplification. Since decoupling keys and values hurts the performance of range queries, it utilizes the parallel random reads of SSDs to efficiently prefetch the values. HashKV similarly separates keys and values stored on SSDs as WiscKey does [14]. It optimizes garbage collection by grouping KVs based on a hash function for update-intensive workloads. LOCS looks into improving the performance of an LSM-tree based KV store by leveraging open-channel SSDs [38].

VT-tree [37] proposes stitching optimization that avoids rewriting already sorted data in an LSM-tree, while maintaining the sequentiality of KVs sufficiently to provide efficient range query performance similar to SLM-DB. However, VT-tree still needs to maintain KV pairs in multiple levels, and does not focus on improving read performance. LSM-trie focuses on reducing write amplification, especially for large scale KV stores with small value sizes by using a trie structure [40]. However, LSM-trie does not support range query operations as it is based on a hash function. PebblesDB proposes the Fragmented Log-Structured Merge Trees, which fragments KVs into smaller files, reducing write amplification in the same level [35]. FloDB introduces a small in-memory buffer on top of MemTable, which optimizes the memory component of an LSM-tree structure and supports skewed read-write workloads effectively [11]. TRIAD also focuses on skewed workloads by keeping hot keys in memory without flushing to disk [10]. The fractal index tree is investigated to reduce I/O amplification for B+-tree based systems [13].

There have been several studies to provide optimal persistent data structures such as a radix tree [29], a hashing scheme [43], and a B+-tree [16, 22, 34, 42] in PM. They propose write optimal techniques while providing consistency of the data structures with 8-byte failure atomic writes in PM.

9 Concluding Remarks

In this paper, we presented the Single-Level Merge DB (SLM-DB) that takes advantage of both the B+-tree index and the LSM-tree approach by leveraging PM. SLM-DB utilizes a persistent B+-tree index, the PM resident MemTable, and a single level disk component of SSTable files with selective compaction. Our extensive experimental study demonstrates that SLM-DB provides high read and write throughput as well as comparable range query performance, compared to LevelDB, while achieving low write amplification and near-optimal read amplification.

Acknowledgments

We would like to thank our shepherd Ajay Gulati and the anonymous reviewers for their invaluable comments. This work was partly supported by Institute for Information & Communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. 2015-0-00590, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development) and National Research Foundation of Korea (NRF) funded by the Korea government(MSIT) (NRF-2018R1A2B6006107 and NRF-2016M3C4A7952634). Young-ri Choi is the corresponding author.

References

[1] Intel and micron produce breakthrough memory technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[2] Kyoto Cabinet: a straightforward implementation of DBM. http://fallabs.com/kyotocabinet/.

[3] Leveldb. https://github.com/google/leveldb.

[4] NVMRocks: RocksDB on Non-Volatile Memory Systems. http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/.

[5] pmem.io Persistent Memory Programming. https://pmem.io/.

[6] pmemkv. https://github.com/pmem/pmemkv.

[7] Recovery and Fault-Tolerance for Persistent Memory Pools Using Persistent Memory Development Kit (PMDK). https://software.intel.com/en-us/articles/recovery-and-fault-tolerance-for-persistent-memory-pools-using-persistent-memory.

[8] RocksDB. https://rocksdb.org/.

[9] YCSB on RocksDB. https://github.com/brianfrankcooper/YCSB/tree/master/rocksdb.

[10] Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., and Konka, P. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)* (2017).

[11] Balmau, O., Guerraoui, R., Trigonakis, V., and Zablotchi, I. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)* (2017).

[12] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[13] Bender, M. A., Farach-Colton, M., Fineman, J. T., Fogel, Y. R., Kuszmaul, B. C., and Nelson, J. Cache-oblivious Streaming B-trees. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (2007).

[14] Chan, H. H. W., Li, Y., Lee, P. P. C., and Xu, Y. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)* (2018).

[15] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006).

[16] Chen, S., and Jin, Q. Persistent B+-trees in Non-volatile Main Memory. *Proceedings of the VLDB Endowment 8*, 7 (Feb. 2015), 786–797.

[17] 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).

[18] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2007).

[19] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System Software for Persistent Memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)* (2014).

[20] Eisenman, A., Gardner, D., AbdelRahman, I., Axboe, J., Dong, S., Hazelwood, K., Petersen, C., Cidon, A., and Katti, S. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the 13th EuroSys Conference (EuroSys)* (2018).

[21] Huai, Y. Spin-Transfer Torque MRAM (STT-MRAM) : Challenges and Prospects. *AAPPS bulletin 18*, 6 (Dec. 2008), 33–40.

[22] Hwang, D., Kim, W.-H., Won, Y., and Nam, B. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 16th Usenix Conference on File and Storage Technologies (FAST)* (2018).

[23] Kannan, S., Bhat, N., Gavrilovska, A., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)* (2018).

[24] Kannan, S., Gavrilovska, A., Gupta, V., and Schwan, K. HeteroOS - OS design for heterogeneous memory management in datacenter. In *Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017).

[25] Kim, H., Seshadri, S., Dickey, C. L., and Chiu, L. Phase Change Memory in Enterprise Storage Systems: Silver Bullet or Snake Oil? *ACM SIGOPS Operating Systems Review 48*, 1 (May 2014), 82–89.

[26] Kim, W.-H., Kim, J., Baek, W., Nam, B., and Won, Y. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[27] Küay, E., Kandemir, M., Sivasubramaniam, A., and Mutlu, O. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2013).

[28] Lakshman, A., and Malik, P. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review 44*, 2 (Apr. 2010), 35–40.

[29] Lee, S. K., Lim, K. H., Song, H., Nam, B., and Noh, S. H. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST)* (2017).

[30] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)* (2016).

[31] Marmol, L., Sundararaman, S., Talagala, N., and Rangaswami, R. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)* (2015).

[32] Mei, F., Cao, Q., Jiang, H., and Tintri, L. T. LSM-tree Managed Storage for Large-scale Key-value Store. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)* (2017).

[33] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. The Log-structured Merge-tree (LSM-tree). *Acta Informatica 33*, 4 (June 1996), 351–385.

[34] Oukid, I., Lasperas, J., Nica, A., Willhalm, T., and Lehner, W. 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)* (2016).

[35] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (2017).

[36] SEARS, R., AND RAMAKRISHNAN, R. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2012).

[37] SHETTY, P. J., SPILLANE, R. P., MALPANI, R. R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building Workload-Independent Storage with VT-Trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)* (2013).

[38] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)* (2014).

[39] WONG, H. . P., RAOUX, S., KIM, S., LIANG, J., REIFENBERG, J. P., RAJENDRAN, B., ASHEGHI, M., AND GOODSON, K. E. Phase Change Memory. *Proceedings of the IEEE 98*, 12 (Dec 2010), 2201–2227.

[40] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based Ultra-large Key-value Store for Small Data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)* (2015).

[41] XIA, F., JIANG, D., XIONG, J., AND SUN, N. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)* (2017).

[42] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)* (2015).

[43] ZUO, P., AND HUA, Y. A Write-Friendly and Cache-Optimized Hashing Scheme for Non-Volatile Memory Systems. *IEEE Transactions on Parallel and Distributed Systems 29*, 5 (May 2018), 985–998.

Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks

Shengan Zheng[†*] Morteza Hoseinzadeh[§] Steven Swanson[§]
[†]*Shanghai Jiao Tong University* [§]*University of California, San Diego*

Abstract

Emerging fast, byte-addressable Non-Volatile Main Memory (NVMM) provides huge increases in storage performance compared to traditional disks. We present Ziggurat, a tiered file system that combines NVMM and slow disks to create a storage system with near-NVMM performance and large capacity. Ziggurat steers incoming writes to NVMM, DRAM, or disk depending on application access patterns, write size, and the likelihood that the application will stall until the write completes. Ziggurat profiles the application's access stream online to predict the behavior of individual writes. In the background, Ziggurat estimates the "temperature" of file data, and migrates the cold file data from NVMM to disks. To fully utilize disk bandwidth, Ziggurat coalesces data blocks into large, sequential writes. Experimental results show that with a small amount of NVMM and a large SSD, Ziggurat achieves up to 38.9× and 46.5× throughput improvement compared with EXT4 and XFS running on an SSD alone, respectively. As the amount of NVMM grows, Ziggurat's performance improves until it matches the performance of an NVMM-only file system.

1 Introduction

Emerging fast, byte-addressable persistent memories, such as battery-backed NVDIMMs [25] and 3D-XPoint [24], promise to dramatically increase the performance of storage systems. These non-volatile memory technologies offer vastly higher throughput and lower latency compared with traditional block-based storage devices.

Researchers have proposed several file systems [8, 10, 11, 29, 32] on NVMM. These file systems leverage the direct access (DAX) feature of persistent memory to bypass the page cache layer and provide user applications with direct access to file data.

The high performance of persistent memory comes at a high cost. The average price per byte of persistent memory is higher than SSD, and SSDs and hard drives scale to much larger capacities than NVMM. So, workloads that are cost-sensitive or require larger capacities than NVMM can provide would benefit from a storage system that can leverage the strengths of both technologies: NVMM for speed and disks for capacity.

Tiering is a solution to this dilemma. Tiered file systems manage a hierarchy of heterogeneous storage devices and place data in the storage device that is a good match for the data's performance requirements and the application's future access patterns.

Using NVMM poses new challenges to the data placement policy of tiered file systems. Existing tiered storage systems (such as FlashStore [9] and Nitro [23]) are based on disks (SSDs or HDDs) that provide the same block-based interface, and while SSDs are faster than hard disks, both achieve better performance with larger, sequential writes and neither can approach the latency of DRAM for reads or writes.

NVMM supports small (e.g., 8-byte) writes and offers DRAM-like latency for reads and write latency within a small factor of DRAM's. This makes the decision of where to place data and metadata more complex: The system must decide where to initially place write data (DRAM or NVMM), how to divide NVMM between metadata, freshly written data, and data that the application is likely to read.

The first challenge is how to fully exploit the high bandwidth and low latency of NVMM. Using NVMM introduces a much more efficient way to persist data than disk-based storage systems. File systems can persist synchronous writes simply by writing them to NVMM, which not only bypasses the page cache layer but also removes the high latency of disk accesses from the critical path. Nevertheless, a DRAM page cache still has higher throughput and lower latency than NVMM, which makes it competitive to perform asynchronous writes to the disk tiers.

The second challenge is how to reconcile NVMM's random access performance with the sequential accesses that disks and SSDs favor. In a tiered file system with NVMM and disks, bandwidth and latency are no longer the only

*This work was done while visiting University of California, San Diego.

differences between different storage tiers. Compared with disks, the gap between sequential and random performance of NVMM is much smaller, which makes it capable of absorbing random writes. Simultaneously, the file system should leverage NVMM to maximize the sequentiality of writes and reads to and from disk.

We propose Ziggurat, a tiered file system that spans NVMM and disks. Ziggurat exploits the benefits of NVMM through intelligent data placement during file writes and data migration. Ziggurat includes two placement predictors that analyze the file write sequences and predict whether the incoming writes are both large and stable, and whether updates to the file are likely to be synchronous. Ziggurat then steers the incoming writes to the most suitable tier based on the prediction: writes to synchronously-updated files go to the NVMM tier to minimize the synchronization overhead. Small, random writes also go to the NVMM tier to fully avoid random writes to disk. The remaining large sequential writes to asynchronously-updated files go to disk.

We implement an efficient migration mechanism in Ziggurat to make room in NVMM for incoming file writes and accelerate reads to frequently accessed data. We first profile the temperature of file data and select the coldest file data blocks to migrate. During migration, Ziggurat coalesces adjacent data blocks and migrates them in large chunks to disk. Ziggurat also adjusts the migration policy according to the application access patterns.

The contributions of this paper include:

- We describe a synchronicity predictor to efficiently predict whether an application is likely to block waiting a write to complete.
- We describe a write size predictor to predict whether the writes to a file are both large and stable.
- We describe a migration mechanism that utilizes the characteristics of different storage devices to perform efficient migrations.
- We design an adaptive migration policy that can fit different access patterns of user applications.
- We implement and evaluate Ziggurat to demonstrate the effectiveness of the predictors and the migration mechanism.

We evaluate Ziggurat using a collection of micro- and macro-benchmarks. We find that Ziggurat is able to obtain near-NVMM performance on many workloads even with little NVMM. With a small amount of NVMM and a large SSD, Ziggurat achieves up to 38.9× and 46.5× throughput improvement compared with EXT4 and XFS running on SSD alone, respectively. As the amount of NVMM grows, Ziggurat's performance improves until it nearly matches the performance of an NVMM-only file system.

The remainder of the paper is organized as follows. Section 2 describes a variety of storage technologies and the NOVA file system. Section 3 presents a design overview of the Ziggurat file system. We discuss the placement policy and the migration mechanism of Ziggurat in Section 4 and Section 5, respectively. Section 6 evaluates Ziggurat, and Section 7 shows some related work. Finally, we present our conclusions in Section 8.

Technology	Latency		Sequential Bandwidth	
	Read	Write	Read	Write
DRAM	0.1μs	0.1μs	25GB/s	25GB/s
NVMM	0.2μs	0.5μs	10GB/s	5GB/s
Optane SSD	10μs	10μs	2.5GB/s	2GB/s
NVMe SSD	120μs	30μs	2GB/s	500MB/s
SATA SSD	80μs	85μs	500MB/s	500MB/s
Hard disk	10ms	10ms	100MB/s	100MB/s

Table 1: **Performance comparison among different storage media.** DRAM, NVMM and hard disk numbers are estimated based on [5, 19, 35]. SSD numbers are extracted from Intel's website.

2 Background

Ziggurat targets emerging non-volatile memory technologies and conventional block-based storage devices (e.g., SSDs or HDDs). This section provides background on NVMM and disks, and the NOVA file system that Ziggurat is based on.

2.1 Storage Technologies

Emerging non-volatile main memory (NVMM), solid-state drive (SSD) and hard disk drive (HDD) technologies have their unique latency, bandwidth, capacity, and characteristics. Table 1 shows the performance comparison of different storage devices.

Non-volatile memory provides byte-addressability, persistence and direct access via the CPU's memory controller. Battery-backed NVDIMMs [25, 26] have been available for some time. Battery-free non-volatile memory technologies include phase change memory (PCM) [22, 28], memristors [31, 33], and spin-torque transfer RAM (STT-RAM) [7, 20]. Intel and Micron's 3D-XPoint [24] will soon be available. All of these technologies offer both longer latency and higher density than DRAM. 3D-XPoint has also appeared in Optane SSDs [17], enabling SSDs that are much faster than their flash-based counterparts.

2.2 The NOVA File System

Ziggurat is implemented based on NOVA [32], an NVMM file system designed to maximize performance on hybrid memory systems while providing strong consistency guarantees. Below, we discuss the file structure and scalability aspects of NOVA's design that are most relevant to Ziggurat.

NOVA maintains a separate log for each inode. NOVA also maintains radix trees in DRAM that map file offsets to NVMM locations. The relationship between the inode, its log, and its data pages is illustrated in Figure 2a. For file writes, NOVA creates *write entries* (the log entries for data updates) in the inode log. Each write entry holds a pointer to the newly written data pages, as well as its modification time (`mtime`). After NOVA creates a write entry, it updates the tail of the inode log in NVMM, along with the in-DRAM radix tree.

NOVA uses per-cpu allocators for NVMM space and per-cpu journals for managing complex metadata updates. This enables parallel block allocation and avoids contention in journaling. In addition, NOVA has per-CPU inode tables to ensure good scalability.

3 Ziggurat Design Overview

Ziggurat is a tiered file system that spans across NVMM and disks (hard or solid state). We design Ziggurat to fully utilize the strengths of NVMM and disks and to offer high file performance for a wide range of access patterns.

Three design principles drive the decisions we made in designing Ziggurat. First, Ziggurat should be *fast-first*. It should use disks to expand the capacity of NVMM rather than using NVMM to improve the performance of disks as some previous systems [14, 15] have done. Second, Ziggurat strives to be *frugal* by placing and moving data to avoid wasting scarce resources (e.g., NVMM capacity or disk bandwidth). Third, Ziggurat should be *predictive* by dynamically learning the access patterns of a given workload and adapting its data placement decisions to match.

These principles influence all aspects of Ziggurat's design. For instance, being fast-first means, in the common case, file writes go to NVMM. However, Ziggurat will make an exception if it predicts that steering a particular write in NVMM would not help application performance (e.g., if the write is large and asynchronous).

Alternatively, if the writes are small and synchronous (e.g., to a log file), Ziggurat will send them to NVMM initially, detect when the log entries have "cooled", and then aggregate those many small writes into larger, sequential writes to disk.

Ziggurat uses two mechanisms to implement these design principles. The first is a placement policy driven by a pair of predictors that measure and analyze past file access behavior to make predictions about future behavior. The second is an efficient migration mechanism that moves data between tiers to optimize NVMM performance and disk bandwidth. The migration system relies on a simple but effective mechanism to identify cold data to move from NVMM to disk.

We describe Ziggurat in the context of a simple two-tiered system comprising NVMM and an SSD, but Ziggurat can use any block device as the "lower" tier. Ziggurat can also handle more than one block device tier by migrating data blocks across different tiers.

3.1 Design Decisions

We made the following design decisions in Ziggurat to achieve our goals.

Send writes to the most suitable tier Although NVMM is the fastest tier in Ziggurat, file writes should not always go to NVMM. NVMM is best-suited for small updates (since small writes to disk are slow) and synchronous writes (since NVMM has higher bandwidth and lower latency). However, for larger asynchronous writes, targeting disk is faster, since Ziggurat can buffer the data in DRAM more quickly than it can write to NVMM, and the write to disk can occur in the background. Ziggurat uses its *synchronicity predictor* to analyze the sequence of writes to each file and predict whether future accesses are likely to be synchronous (i.e., whether the application will call `fsync` in the near future).

Only migrate cold data in cold files During migration, Ziggurat targets the cold portions of cold files. Hot files and hot data in unevenly-accessed files remain in the faster tier. When the usage of the fast tier is above a threshold, Ziggurat selects files with the earliest average modification time to migrate (Section 5.1). Within each file, Ziggurat migrates blocks that are older than average. Unless the whole file is cold (i.e., its modification time is not recent), in which case we migrate the whole file.

High NVMM space utilization Ziggurat fully utilizes NVMM space to improve performance. Ziggurat uses NVMM to absorb synchronous writes. Ziggurat uses a dynamic migration threshold for NVMM based on the read-write pattern of applications, so it makes the most of NVMM to handle file reads and writes efficiently. We also implement reverse migration (Section 5.2) to migrate data from disk to NVMM when running read-dominated workloads.

Migrate file data in groups In order to maximize the write bandwidth of disks, Ziggurat performs migration to disks as sequentially as possible. The placement policy ensures that most small, random writes go to NVMM. However, migrating these small write entries to disks directly will suffer from the poor random access performance of disks. In order to make migration efficient, Ziggurat coalesces adjacent file data into large chunks for migration to exploit sequential disk bandwidth (Section 5.3).

High scalability Ziggurat extends NOVA's per-cpu storage space allocators to include all the storage tiers. It also uses per-cpu migration and page cache writeback threads to improve scalability.

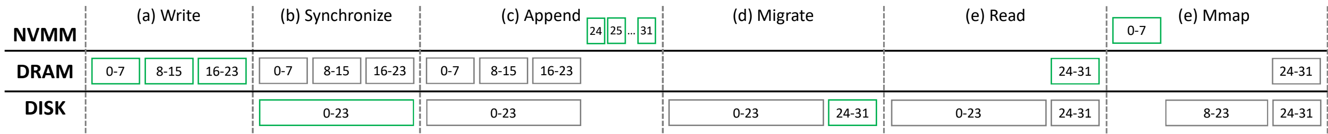


Figure 1: **File operations.** Ziggurat utilizes different storage tiers to handle I/O requests such as write, synchronize, append, migrate, read and mmap efficiently.

3.2 File Operations

Figure 1 illustrates how Ziggurat handles operations on files (write, synchronize, append, migrate, read, and mmap) that span multiple tiers.

Write The application initializes the first 24 blocks of the file with three sequential writes in (a). Ziggurat first checks the results from the synchronicity predictor and the write size predictor (Section 4) to decide which tier should receive the new data. In the example, the three writes are large and Ziggurat predicts that the accesses are asynchronous, so Ziggurat steers these writes to disk. It writes the data to the page cache in DRAM and then asynchronously writes them to disk.

Synchronize The application calls `fsync` in (b). Ziggurat traverses the write log entries of the file, and writes back the dirty data pages in the DRAM page cache. The write-back threads merge all adjacent dirty data pages to perform large sequential writes to disk. If the file data were in NVMM, `fsync` would be a no-op.

Append After the `fsync`, the application performs eight synchronous writes to add eight blocks to the end of the file in (c). The placement predictor recognizes the pattern of small synchronous writes and Ziggurat steers the writes to NVMM.

Migrate When the file becomes cold in (d), Ziggurat evicts the first 24 data pages from DRAM and migrates last eight data blocks from NVMM to disk using group migration (Section 5.3).

Read The user application reads the last eight data blocks in (e). Ziggurat fetches them from disk to DRAM page cache.

Memory map The user application finally issues a `mmap` request to the head of the file in (f). Ziggurat uses reverse migration to bring the data into NVMM and then maps the pages into the application's address space.

4 Placement Policy

Ziggurat steers synchronous or small writes to NVMM, but it steers asynchronous, large writes to disk, because writing to the DRAM page cache is faster than writing to NVMM, and Ziggurat can write to disk in the background. It uses two predictors to distinguish these two types of writes.

Synchronicity predictor The synchronicity predictor predicts whether the application is likely to call `fsync` on the file in the near future. The synchronicity predictor counts the number of data blocks written to the file between two calls to `fsync`. If the number is less than a threshold (e.g., 1024 in our experiments), the predictor classifies it as a synchronously-updated file. The predictor treats writes to files opened with `O_SYNC` as synchronous as well.

Write size predictor The write size predictor not only ensures that a write is large enough to effectively exploit disk bandwidth but also that the future writes within the same address range are also likely to be large. The second condition is critical. For example, if the application initializes a file with large I/Os, and then performs many small I/Os, these small new write entries will read and invalidate discrete blocks, increasing fragmentation and leading to many random disk accesses to service future reads.

Ziggurat's write size predictor keeps a counter in each write entry to indicate whether the write size is both large and stable. When Ziggurat rewrites an old write entry, it first checks whether the write size is big enough to cover at least half the area taken up by the original log entry. If so, Ziggurat transfers the counter value of the old write entry to the new one and increases it by one. Otherwise, it resets the counter to zero. If the number is larger than four (a tunable parameter), Ziggurat classifies the write as "large". Writes that are both large and asynchronous go to disk.

5 Migration Mechanism

The purpose of migration is to make room in NVMM for incoming file writes, as well as speeding up reads to frequently accessed data. We use *basic migration* to migrate data from disk to NVMM to fully utilize NVMM space when running read-dominated workloads. We use *group migration* to migrate data from NVMM to disk by coalescing adjacent data blocks to achieve high migration efficiency and free up space for future writes. Ziggurat can achieve near-NVMM performance for most accesses as long as the migration mechanism is efficient enough.

In this section, we first describe how Ziggurat identifies good targets for migration. Then, we illustrate how it migrates data efficiently to maximize the bandwidth of the disk

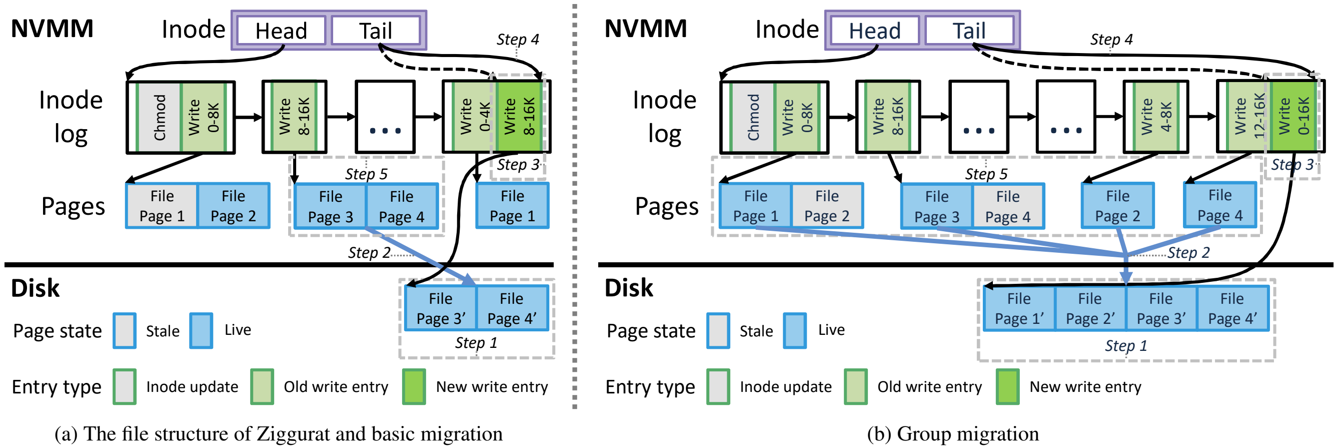


Figure 2: **Migration mechanism of Ziggurat.** Ziggurat migrates file data between tiers using its basic migration and group migration mechanisms. The blue arrows indicate data movement, while the black ones indicate pointers.

with basic migration and group migration. Finally, we show how to migrate file logs efficiently.

5.1 Migration Profiler

Ziggurat uses a migration profiler to identify cold data to migrate from NVMM to disk.

Implementation. Ziggurat first identifies the cold files to migrate. Ziggurat profiles the temperature of each file by maintaining *cold lists*, the per-cpu lists of files on each storage tier, sorted by the average modification time(`amtime`) computed across all the blocks in the file. The per-cpu cold lists correspond to per-cpu migration threads which migrate files from one tier to another. Ziggurat updates the cold list whenever it modifies a file. To identify the coldest blocks within an cold file, Ziggurat tracks the mtime for each block in the file.

To migrate data, Ziggurat pops the coldest file from a cold list. If the `mtime` of the popped file is not recent (more than 30 seconds ago), then Ziggurat treats the whole file as cold and migrates all of it. Otherwise, the modification time of the file's block will vary, and Ziggurat migrates the write entries with `mtime` earlier than the `amtime` of the file. Hence, the cold part of the file is migrated to a lower tier, and the hot part of the file stays in the original tier.

Deciding when to migrate. Most existing tiered storage systems (such as [6, 21]) use a fixed utilization threshold to decide when to migrate data to lower tiers. However, a higher threshold is not suitable for write-dominated workloads, since the empty space in persistent memory will be devoured by intensive file writes. In this case, the file writes have to either stall before the migration threads clean up enough space in NVMM, or write to disk. On the other hand, a lower threshold is not desirable for read-dominated workloads, since reads have to load more blocks from disks instead of NVMM. We implement a dynamic threshold for NVMM in Ziggurat based on the overall read-write ratio of the file system. The migration threshold rises from 50% to 90% as the read-write ratio of the system changes.

5.2 Basic Migration

The goal of basic migration is to migrate the coldest data in Ziggurat to disk. When the usage of the upper tier is above the threshold, a per-cpu migration thread migrates the coldest data in a cold file to disk. The migration process repeats until the usage of the upper tier is below the threshold again.

The granularity of migration is a write entry. During migration, we traverse the in-DRAM radix tree to locate every valid write entry in the file and migrate the write entries with `mtime` earlier than the `amtime` of the file.

Figure 2a illustrates the basic procedures of how Ziggurat migrates a write entry from NVMM to disk. The first step is to allocate continuous space on disk to hold the migrated data. Ziggurat copies the data from NVMM to disk. Then, it appends a new write entry to the inode log with the new location of the migrated data blocks. After that, it updates the log tail in NVMM and the radix tree in DRAM. Finally, Ziggurat frees the old blocks of NVMM.

To improve scalability, Ziggurat uses locks in the granularity of a write entry instead of an entire file. Ziggurat locks write entries during migration but other parts of the file remain available for reading. Migration does not block file writes. If any foreground file I/O request tries to acquire the inode lock, the migration thread will stop migrating the current file, and release the lock.

If a write entry migrates to a disk when the DRAM page cache usage is low (i.e., below 50%), Ziggurat will make a copy of the pages in the DRAM page cache in order to

accelerate future reads. Writes will benefit from this as well, since unaligned writes have to read the partial blocks from their neighbor write entries to fill the data blocks.

Ziggurat implements reverse migration, which migrates file data from disks to NVMM, using basic migration. Write entries are migrated successively without grouping since NVMM can handle sequential and random writes efficiently. File `mmap` uses reverse migration to enable direct access to persistent data. Reverse migration also optimizes the performance of read-dominated workloads when NVMM usage is low since the performance depends on the size of memory. If Ziggurat can only migrate data from a faster tier to a slower one, then the precious available space of NVMM will stay idle when running read-dominated workloads. Meanwhile, the data on disks contend for a limited DRAM. Reverse migration makes full use of NVMM in such a scenario.

5.3 Group Migration

Group migration avoids fine-grain migration to improve efficiency and maximize sequential bandwidth to disks. Ziggurat tends to fill NVMM with small writes due to its data placement policy. Migrating them from NVMM to disk with basic migration is inefficient because it will incur the high random access latency of disks.

Group migration coalesces small write entries in NVMM into large sequential ones to disk. There are four benefits: (1) It merges small random writes into large sequential writes, which improves the migration efficiency. (2) If the migrated data is read again, loading continuous blocks is much faster than loading scattered blocks around the disk. (3) By merging write entries, the log itself becomes smaller, reducing metadata access overheads. (4) It moderates disk fragmentation caused by log-structured writes by mimicking garbage collection.

As illustrated in Figure 2b, the steps of group migration are similar to migrating a write entry. In step 1, we allocate large chunks of data blocks in the lower tier. In step 2, we copy multiple pages to the lower tier with a single sequential write. After that, we append the log entry, and update the inode log tail, which commits the group migration. The stale pages and logs are freed afterward. Ideally, the *group migration size* (the granularity of group migration) should be set close to the future I/O size, so that applications can fetch file data with one sequential read from disk. In addition, it should not exceed the CPU cache size in order to maximize the performance of loading the write entries from disks.

5.4 File Log Migration

Ziggurat migrates file logs in addition to data when NVMM utilization is too high, freeing up space for hot data and metadata. Ziggurat periodically scans the cold lists, and initiates log migration on cold files. Figure 3 illustrates how log migration is performed. Ziggurat copies live log entries from NVMM into the page cache. The log entries are compacted into new log pages during coping. Then, it writes the new log pages back to disk, and updates the inode metadata cache in DRAM to point to the new log. After that, Ziggurat atomically replaces the old log with the new one and reclaims the old log.

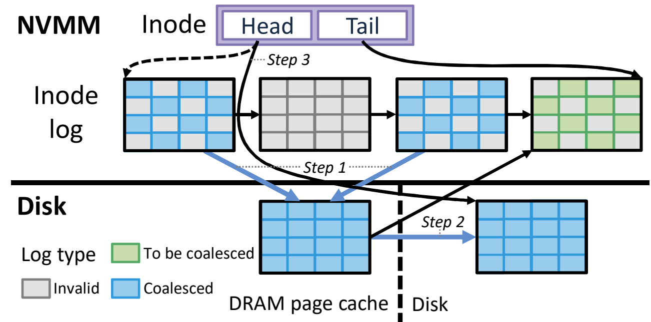


Figure 3: **Log migration in Ziggurat.** Ziggurat compacts logs as it moves them from NVMM to disk.

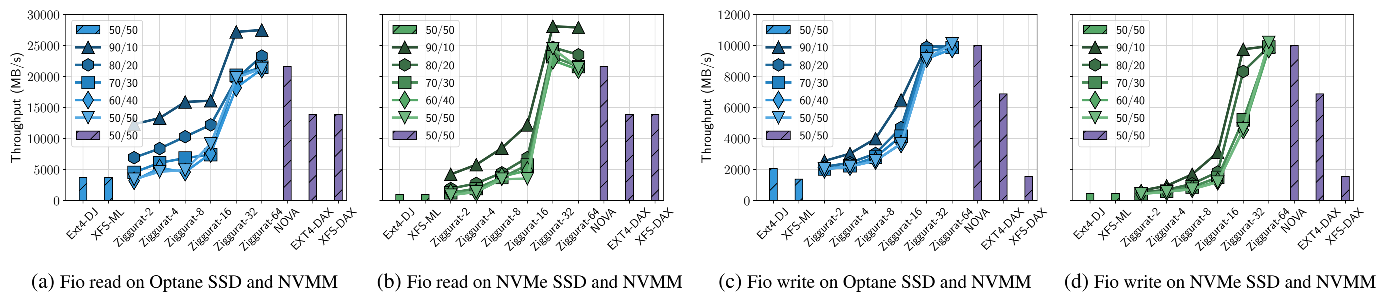
6 Evaluation

6.1 Experimental Setup

Ziggurat is implemented on Linux 4.13. We used NOVA as the code base to build Ziggurat, and added around 8.6k lines of code. To evaluate the performance of Ziggurat, we run micro-benchmarks and macro-benchmarks on a dual-socket Intel Xeon E5 server. Each processor runs at 2.2GHz, has 10 physical cores, and is equipped with 25 MB of L3 cache and 128 GB of DRAM. The server also has a 400 GB Intel DC P3600 NVMe SSD and a 187 GB Intel DC P4800X Optane SSD.

As persistent memory devices are not yet available, we emulate the latency and bandwidth of NVMM with the NUMA effect on DRAM. There are two NUMA nodes in our platform. During the experiments, the entire address space of NUMA node 1 is used for NVMM emulation. All applications are pinned to run on the processors and memory of NUMA node 0. Table 2 shows the DRAM latency of our experimental platform by Intel Memory Latency Checker [16].

We compare Ziggurat with different types of file systems. For NVMM-based file systems, we compare Ziggurat with NOVA [32], Strata [21] (NVMM only) and the DAX-based file systems on Linux: EXT4-DAX and XFS-DAX. For disk-based file systems, we compare Ziggurat with EXT4 in the data journaling mode (-DJ) and XFS in the metadata logging mode (-ML). Both EXT4-DJ and XFS-ML provide data atomicity, like Ziggurat. For EXT4-DJ, the journals are kept in a 2 GB journaling block device (JBD) on NVMM. For XFS-ML, the metadata logging device is 2 GB of NVMM.



(a) Fio read on Optane SSD and NVMM (b) Fio read on NVMe SSD and NVMM (c) Fio write on Optane SSD and NVMM (d) Fio write on NVMe SSD and NVMM

Figure 4: **Fio performance.** Each workload performs 4 KB reads/writes to a hybrid file system backed by NVMM and SSD (EXT4-DJ, XFS-ML and Ziggurat) or an NVMM-only file system (NOVA, EXT4-DAX and XFS-DAX).

Node	0	1
0	76.6	133.7
1	134.2	75.5

(a) NUMA latency (ns)

Node	0	1
0	52213.9	25505.9
1	25487.3	52111.8

(b) NUMA bandwidth (MB/s)

Table 2: **NUMA latency and bandwidth of our platform.** We use the increased latency and reduced bandwidth of the remote NUMA node to emulate the lower performance of NVMM compared to DRAM.

We limit the capacity of the DRAM page cache to 10 GB.

For tiered file systems, we only do the comparison among Ziggurat with different configurations. To the best of our knowledge, Strata is the only currently available tiered file system that spans across NVMM and disks. However, the publicly available version of Strata only supports a few applications and has trouble running workloads with dataset sizes larger than NVMM size as well as multi-threaded applications.

We vary the NVMM capacity available to Ziggurat to show how performance changes with different storage configurations. The dataset size of each workload is smaller than 64 GB. The variation starts with Ziggurat-2 (i.e., Ziggurat with 2 GB of NVMM). In this case, most of the data must reside on disk forcing Ziggurat to frequently migrate data to accommodate incoming writes. Ziggurat-2 is also an interesting comparison point for EXT4-DJ and XFS-ML, since those configurations take different approaches to using a small amount of NVMM to improve file system performance. The variation ends with Ziggurat-64 (i.e., Ziggurat with 64 GB of NVMM). The group migration size is set to 16 MB. We run each workload three times and report the average across these runs.

6.2 Microbenchmarks

We demonstrate the relationship between access locality and the read/write throughput of Ziggurat with Fio [1]. Fio can

Locality	90/10	80/20	70/30	60/40	50/50
Parameter θ	1.04	0.88	0.71	0.44	0

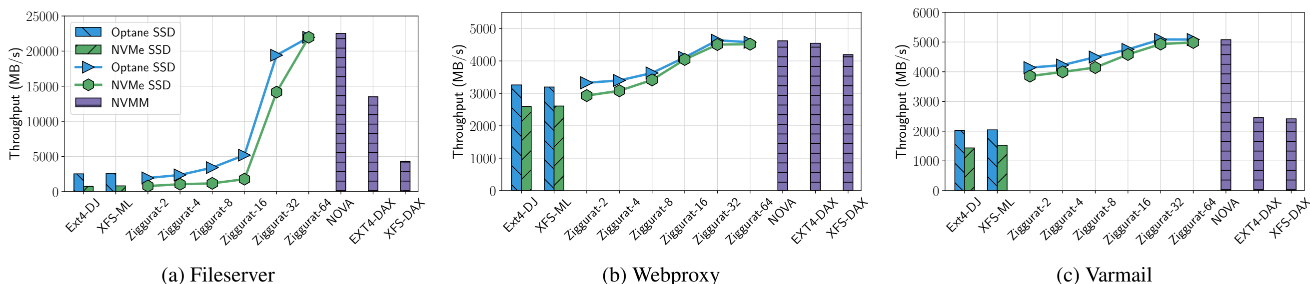
Table 3: **Zipf Parameters.** We vary the Zipf parameter, θ, to control the amount of locality in the access stream.

issue random read/write requests according to Zipfian distribution. We vary the Zipf parameter θ to adjust the locality of random accesses. We present the results with a range of localities range from 90/10 (90% of accesses go to 10% of data) to 50/50 (Table 3). We initialize the files with 2 MB writes and the total dataset is 32 GB. We use 20 threads for the experiments, each thread performs 4 KB I/Os to a private file, and all writes are synchronous.

Figure 4 shows the results for Ziggurat, EXT4-DJ, and XFS-ML on Optane SSD and NVMe SSD, as well as NOVA, EXT4-DAX, and XFS-DAX on NVMM. The gaps between the throughputs from Optane SSD and NVMe SSD in both graphs are large because Optane SSD's read/write bandwidth is much higher than the NVMe SSD's. The throughput of Ziggurat-64 is close to NOVA for the 50/50 locality, the performance gap between Ziggurat-64 and NOVA is within 2%. This is because when all the data fits in NVMM, Ziggurat is as fast as NOVA. The throughput of Ziggurat-2 is within 5% of EXT4-DJ and XFS-ML.

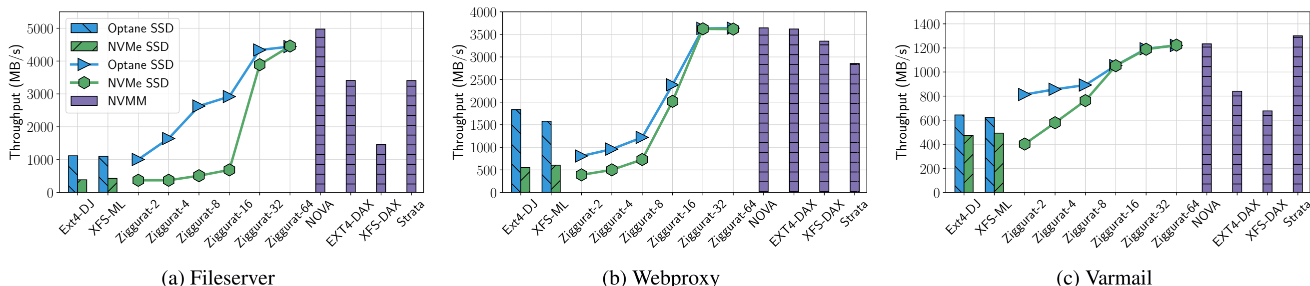
In Figure 4a and Figure 4b, the random read performance of Ziggurat grows with increased locality. The major overhead of reads comes from fetching cold data blocks from disk to DRAM page cache. There is a dramatic performance increase in 90/10, due to CPU caching and the high locality of the workload.

In Figure 4c and Figure 4d, the difference between the random write performance of Ziggurat with different amounts of locality is small. Since all the writes are synchronous 4 KB aligned writes, Ziggurat steers these writes to NVMM. If NVMM is full, Ziggurat writes the new data blocks to the DRAM page cache and then flushes them to disk synchronously. Since the access pattern is random, the migration threads cannot easily merge the discrete data blocks to



(a) Fileserver (b) Webproxy (c) Varmail

Figure 5: **Filebench performance (multi-threaded).** Each workload runs with 20 threads so as to fully show the scalability of the file systems. The performance gaps between Optane SSD and NVMe SSD are smaller than the single-threaded ones.



(a) Fileserver (b) Webproxy (c) Varmail

Figure 6: **Filebench performance (single-threaded).** For small amounts of NVMM, Ziggurat is no slower than conventional file systems running on disk. With large amount of NVMM, performance nearly matches that of NVMM-only file systems.

perform group migration in large sequential writes to disk. Therefore, the migration efficiency is limited by the random write bandwidth of disks, which leads to accumulated cold data blocks in NVMM. Increasing NVMM size, increasing locality, or reducing work set size can all help alleviate this problem.

We also measure the disk throughput of the random write workloads on Ziggurat-2 to show how Ziggurat fully utilizes disk bandwidth to achieve maximum performance. Although it is hard to merge the discrete data blocks to perform group migration, the per-CPU migration threads make full use of the concurrency of disks to achieve high migration efficiency. The average disk write bandwidth of Ziggurat-2 is 1917 MB/s and 438 MB/s for Optane SSD and NVMe SSD, respectively. These values are very close to the bandwidth limit numbers in Table 1.

6.3 Macrobenchmarks

We select three Filebench workloads: fileserver, webproxy, and varmail to evaluate the overall performance of Ziggurat. Table 4 summarizes the characteristics of these workloads.

Figure 5 shows the multi-threaded Filebench throughput on our five comparison file systems and several Ziggurat configurations. In general, we observe that the throughput of Ziggurat-64 is close to NOVA, the performance gap between Ziggurat-64 and NOVA is within 3%. Ziggurat gradually bridges the gap between disk-based file systems and NVMM-based file systems by increasing the NVMM size.

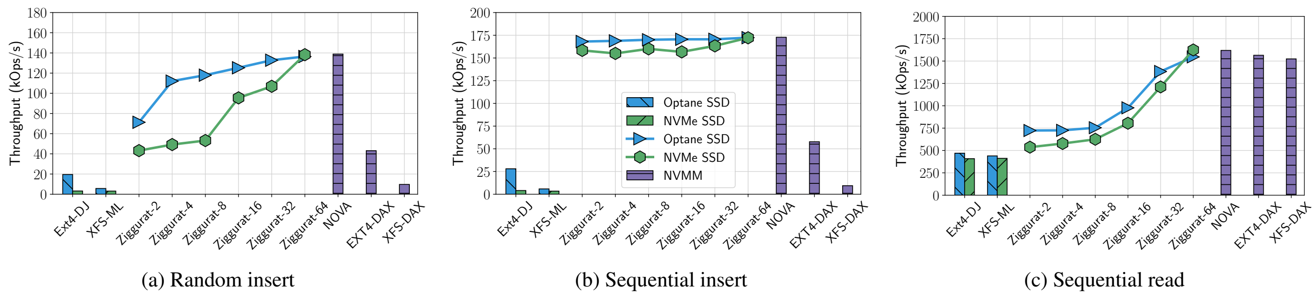
Workload	Average file size	# of files	I/O size (R/W)	Threads	R/W ratio
Fileserver	2MB	16K	16KB/16KB	20/1	1:2
Webproxy	2MB	16K	1MB/16KB	20/1	5:1
Varmail	2MB	16K	1MB/16KB	20/1	1:1

Table 4: **Filebench workload characteristics.** These workloads have different read/write ratios and access patterns.

Fileserver emulates the I/O activity of a simple file server, which consists of creates, deletes, appends, reads and writes. In the fileserver workload, Ziggurat-2 has similar throughput to EXT4-DJ and XFS-ML. The performance increases significantly when the NVMM size is larger than 32 GB since most of the data reside in memory. Ziggurat-64 outperforms EXT4-DAX and XFS-DAX by 2.6× and 5.1×.

Webproxy is a read-intensive workload, which involves appends and repeated reads to files. Therefore, all the file systems achieve high throughputs by utilizing the page cache.

Varmail emulates an email server with frequent synchronous writes. Ziggurat-2 outperforms EXT4-DJ and XFS-ML by 2.1× (Optane SSD) and 2.6× (NVMe SSD)



(a) Random insert (b) Sequential insert (c) Sequential read

Figure 7: **Rocksdb performance.** Ziggurat shows good performance for inserting file data with write-ahead logging, due to the clear distinction between hot and cold files and its migration mechanism.

on average. Varmail performs an `fsync` after every two appends. Ziggurat analyzes these synchronous appends and steers them to NVMM, eliminating the cost of most of the `fsyncs`.

Figure 6 illustrates the single-threaded Filebench throughputs. Strata achieves the best throughput in the varmail workload since its digestion skips many temporary durable writes which are superseded by subsequent writes. However, Ziggurat-64 outperforms Strata by 31% and 27% on the fileserver and webproxy workloads due to the inefficiency of reads in Strata.

6.4 Rocksdb

We illustrate the high performance of updating a key-value store with write-ahead logging (WAL) on Ziggurat with Rocksdb [13], a persistent key-value store based on log-structured merge trees (LSM-trees). Every update to RocksDB is written to two places: an in-memory data structure called memtable and a write-ahead log in the file system. When the size of the memtable reaches a threshold, RocksDB writes it back to disk and discards the log.

We select three Rocksdb workloads from db_bench: random insert (FillUniqueRandom), sequential insert (FillSeq), and sequential read (ReadSeq) to evaluate the key-value throughput and migration efficiency of Ziggurat. We set the writes to synchronous mode for a fair comparison. The database size is set to 32 GB.

Figure 7 measures the Rocksdb throughput. In the random insert workload, Ziggurat with 2 GB of NVMM achieves 8.6× and 13.2× better throughput than EXT4-DJ and XFS-ML, respectively. In the sequential insert workload, Ziggurat is able to maintain near-NVMM performance even when there are only 2 GB of NVMM. It achieves up to 38.9× and 46.5× throughput of EXT4-DJ and XFS-ML, respectively.

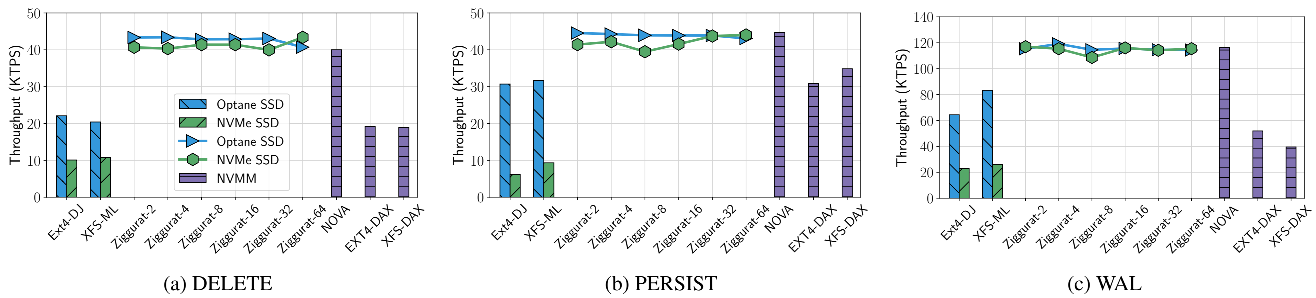
WAL is a good fit for Ziggurat. The reason is three-fold. First, since the workload updates WAL files much more frequently than the database files, the migration profiler can differentiate them easily. The frequently-updated WAL files remain in NVMM, whereas the rarely-updated database files are subject to migration.

Second, the database files are usually larger than the group migration size. Therefore, group migration can fully-utilize the high sequential bandwidth of disks. Moreover, since Rocksdb mostly updates the journal files instead of the large database files, the migration threads can merge the data blocks from the database files and perform sequential writes to disk without interruption. The high migration efficiency helps clean up NVMM space more quickly so that NVMM can absorb more synchronous writes, which in turn boosts the performance.

Third, the WAL files are updated frequently with synchronous and small updates. The synchronicity predictor can accurately identify the synchronous write pattern from the access stream of the WAL files, and the write size predictor can easily discover that the updates to these files are too small to be steered to disk. Therefore, Ziggurat steers the updates to NVMM so that it can eliminate the double copy overhead caused by synchronous writes to disks. Since the entire WAL files are hot, Ziggurat is able to maintain high performance as long as the size of NVMM is larger than the total size of the WAL files, which is only 128 MB in our experiments.

Comparing Figure 7a and Figure 7b, the difference between the results from random and sequential insert of Ziggurat is due to read-modify-writes for unaligned writes. In the random insert workload, the old data blocks of the database files are likely to be on disk, especially when the NVMM size is small. Thus, loading them from disks introduces large overhead. However, in the sequential insert workload, the old data blocks come from recent writes to the files which are likely to be in NVMM. Hence, Ziggurat achieves near-NVMM performance in the sequential insert workload.

In sequential read, Ziggurat-2 outperforms EXT-DJ and XFS-ML by 42.8% and 47.5%. With increasing NVMM size, the performance of Ziggurat gradually increases. The read throughputs of Ziggurat-64, NOVA, EXT4-DAX, and XFS-DAX are close (within 6%).



(a) DELETE (b) PERSIST (c) WAL

Figure 8: **SQLite performance.** Ziggurat maintains near-NVMM performance because the hot journal files are either short-lived or frequently updated, so Ziggurat keeps them in NVMM. Migrating the cold database file with group migration in the background imposes little overhead to foreground file operations.

6.5 SQLite

We analyze the performance of different logging mechanisms on Ziggurat by measuring SQLite [2], a popular lightweight relational database that supports both undo and redo logging. It hosts the entire database in a single file, with other auxiliary files for logging (rollback or write-ahead log). We use Mobibench [18] to test the performance of SQLite with three journaling modes: DELETE, PERSIST and WAL. DELETE and PERSIST are rollback journaling modes. The journal files are deleted at the end of each transaction in DELETE mode. The PERSIST mode foregoes the deletion and instead overwrites the journal header with zeros. The WAL mode uses write-ahead logging for rollback journaling. The database size is set to 32 GB in the experiments. The experimental results are presented in Figure 8.

For DELETE and PERSIST, the journal files are either short-lived or frequently updated. Therefore, they are classified as hot files by the migration profiler of Ziggurat. Hence, Ziggurat only migrates the cold parts of the database files, leaving the journal files in NVMM to absorb frequent updates. The performance gain comes from accurate profiling and high migration efficiency of Ziggurat. With an efficient migration mechanism, Ziggurat can clear up space in NVMM fast enough for in-coming small writes. As a result, Ziggurat maintains near-NVMM performance in all configurations. Compared with block-based file systems running on Optane SSD, Ziggurat achieves 2.0× and 1.4× speedup for DELETE and PERSIST on average, respectively. Furthermore, Ziggurat outperforms block-based file systems running on NVMe SSD by 3.9× and 5.6× for DELETE and PERSIST on average, respectively.

In WAL mode, there are three types of files: the main database files and two temporary files for each database: *WAL* and *SHM*. The *WAL* files are the write-ahead log files, which are hot during key-value insertions. The *SHM* files are the shared-memory files which are used as the index for the WAL files. They are accessed by SQLite via `mmap`.

Ziggurat's profiler keeps these hot files in NVMM. Meanwhile, the cold parts of the large database files migrate to disks in the background. This only introduces very small overhead to the foreground database operations. Therefore, Ziggurat maintains near-NVMM performance even when there's only about 5% of data is actually in NVMM (Ziggurat-2), which outperforms block-based file systems by 1.61× and 4.78×, respectively. Ziggurat also achieves 2.22× and 2.92× higher performance compared with EXT4-DAX and XFS-DAX on average.

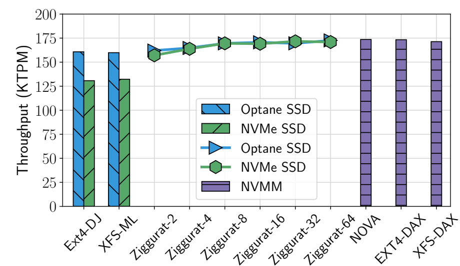
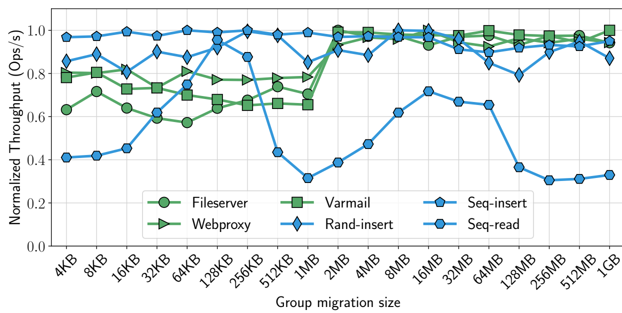


Figure 9: **MySQL performance.** Ziggurat manages to keep high throughput even with little NVMM.

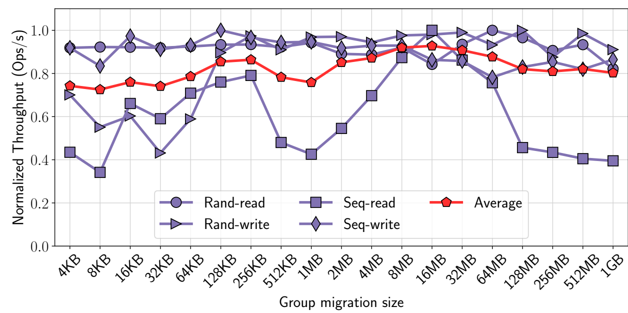
6.6 MySQL

We further evaluate the throughput of databases on Ziggurat with MySQL [27], another widely-used relational database. We measure the throughput of MySQL with TPC-C [3], a representative online transaction processing (OLTP) workload. We run the experiments with a data set size of 20 GB.

Figure 9 shows the MySQL throughput. The performance of Ziggurat is always close to or better than EXT-DJ and XFS-ML. On average, Ziggurat-2 outperforms disk-based file systems by 1% (Optane SSD) and 19% (NVMe SSD). During the transactions, Ziggurat steers most of the small updates to NVMM. Since the transactions need to be processed in DRAM, Ziggurat is capable of migrating the data blocks to disks in time, which leaves ample NVMM space to



(a) Filebench and Rocksdb



(b) Fio

Figure 10: **Performance impact of group migration size.** We use Filebench (green), Rocksdb (blue) and Fio (purple) as our benchmarks. The average throughput (red) peaks when the group migration size is set to 16 MB.

receive new writes. Thus, Ziggurat maintains near-NVMM performance even with little NVMM.

6.7 Parameter Tuning

We illustrate the impact of the parameter choices on performance by measuring the throughput of workloads from Fio, Filebench and Rocksdb with a range of thresholds. We run the workloads with Ziggurat-2 on NVMe SSD.

Group migration size We vary the group migration size from 4 KB to 1 GB. The normalized throughputs relative to the maximum performance are shown in Figure 10. In general, larger group migration size provides better performance for write-dominated workloads, such as Rand-write from Fio and Rand-insert from Rocksdb.

For read-dominated workloads, such as Seq-read from Fio and Seq-read from Rocksdb, the throughputs peak when the group migration size is set to 128 KB and 16 MB. This is because the maximum I/O size of our NVMe SSD is 128 KB and the CPU cache size of our experimental platform is 25 MB. Note that the group migration size is also the granularity of loading file data from disk to DRAM since we fetch file data in the granularity of write entry. On one hand, if the group migration size is too small, Ziggurat has to issue multiple requests to load the on-disk file data into DRAM, which hurts performance. On the other hand, if the group migration size is too large, then a small read request will fetch redundant data blocks from disk, which will waste I/O bandwidth and pollute CPU cache. As Figure 10b shows, the average throughputs of all ten workloads peak when the group migration size is set to 16 MB.

The throughputs of Filebench workloads are saturated when the group migration size reaches 2 MB because the average file size of the workloads is 2 MB. During the migration of the Filebench workloads, the data blocks of a file are coalesced into one write entry, which suits the access pattern of whole-file-reads.

Synchronous write size threshold We vary the synchronous write size threshold from 4 KB to 1 GB. The performance results are insensitive to the synchronous write size threshold throughout the experiments. The standard deviation is less than 3% of the average throughput. We further examine the accuracy of the synchronicity predictor given different synchronous write size thresholds. The predictor accurately predicts the presence or absence of an `fsync` in the near future 99% of the time. The lowest accuracy (97%) occurs when the synchronous write size is set between the average file size and the append size of Varmail. In this case, the first `fsync` contains the writes from file initialization and the first append, while the subsequent `fsyncs` only contain one append. In general, the synchronous write size threshold should be set a little larger than the average I/O size of the synchronous write operations from the workloads. In this case, the synchronicity predictor can not only identify synchronously updated files easily, but also effectively distinguish asynchronous, large writes from rest of the access stream.

Sequential write counter threshold We vary the sequential write counter threshold of Ziggurat from 1 to 64. We find that different sequential write counter thresholds have little impact on performance since the characteristics of our workloads are stable. Users should balance the trade-off between accuracy and prediction overhead when running workloads with unstable access patterns. A higher threshold number improves the accuracy of the sequential predictor, which can effectively avoid jitter in variable workloads. However, it also introduces additional prediction overhead for Ziggurat to produce correct prediction.

7 Related Work

The introduction of multiple storage technologies provides an opportunity of having a large uniform storage space over

a set of different media with varied characteristics. Applications may leverage the diversity of storage choices either directly (e.g. the persistent read cache of RocksDB), or by using NVMM-based file systems (e.g. NOVA, EXT4-DAX or XFS-DAX). In this section, we place Ziggurat's approach to this problem in context relative to other work in this area.

NVMM-Based File Systems. BPFS [8] is a storage class memory (SCM) file system, which is based on shadow-paging. It proposes short-circuit shadow paging to curtail the overheads of shadow-paging in regular cases. However, some I/O operations that involve a large portion of the file system tree (such as moving directories) still impose large overheads. Like BPFS, Ziggurat also exploits fine-grained copy-on-write in all I/O operations.

SCMFS [30] offers simplicity and performance gain by employing the virtual address space to enable continuous file addressing. SCMFS keeps the mapping information of the whole available space in a page table which may be scaled to several Gigabytes for large NVMM. This may result in a significant increase in the number of TLB misses. Although Ziggurat similarly maps all available storage devices into a unified virtual address space, it also performs migration from NVMM to block devices, and group page allocation which reduces TLB misses.

PMFS [11] is another NVMM-based file system which provides atomicity in metadata updates through journaling, but large size write operations are not atomic because it relies on small size in-place atomic updates. Unlike PMFS, Ziggurat's update mechanism is always through journaling with fine-grained copy-on-writes.

Dong *et al.* propose SoupFS [10], a simplified soft update implementation of an NVMM-based file system. They adjust the block-oriented directory organization to use hash tables to leverage the byte-addressability of NVMM. It also gains performance by taking out most synchronous flushes from the critical path. Ziggurat also exploits asynchronous flushes to clear the critical path for higher write throughput.

Tiering Systems. Hierarchical storage Management (HSM) systems date back decades to when disks and tapes were the only common massive storage technologies. There have been several commercial HSM solutions for block-based storage media such as disk drives. IBM Tivoli Storage Manager is one of the well-established HSM systems that transparently migrates rarely used or sufficiently aged files to a lower cost media. EMC DiskXtender is another HSM system with the ability of automatically migrating inactive data from the costly tier to a lower cost media. AutoTiering [34] is another example of a block-based storage management system. It uses a sampling mechanism to estimate the IOPS of running a virtual machine on other tiers. It calculates their performance scores based on the IOPS measurement and the migration costs, and sorts all possible movements accordingly. Once it reaches a threshold, it initiates a live migration.

Since the invention of NVDIMMs, many fine-grained tiering solutions have been introduced. Agarwal *et al.* propose Thermostat [4], a methodology for managing huge pages in two-tiered memory which transparently migrates cold pages to NVMM as the slow memory, and hot pages to DRAM as the fast memory. The downside of this approach is the performance degradation for those applications with uniform temperature across a large portion of the main memory. Conversely, Ziggurat's migration granularity is variable, so it does not hurt performance due to fixed-size migration as in Thermostat. Instead of huge pages, it coalesces adjacent dirty pages into larger chunks for migration to block devices.

X-Mem [12] is a set of software techniques that relies on an off-line profiling mechanism. The X-Mem profiler keeps track of every memory access and traces them to find out the best storage match for every data structure. X-Mem requires users to make several modifications to the source code. Additionally, unlike Ziggurat, the off-line profiling run should be launched for each application before the production run.

Strata [21] is a multi-tiered user-space file system that exploits NVMM as the high-performance tier, and SSD/HDD as the lower tiers. It uses the byte-addressability of NVMM to coalesce logs and migrate them to lower tiers to minimize write amplification. File data can only be allocated in NVMM in Strata, and they can be migrated only from a faster tier to a slower one. The profiling granularity of Strata is a page, which increases the bookkeeping overhead and wastes the locality information of file accesses.

8 Conclusion

We have implemented and described Ziggurat, a tiered file system that spans across NVMM and disks. We manage data placement by accurate and lightweight predictors to steer incoming file writes to the most suitable tier, as well as an efficient migration mechanism that utilizes the different characteristics of storage devices to achieve high migration efficiency. Ziggurat bridges the gap between disk-based storage and NVMM-based storage, and provides high performance and large capacity to applications.

Acknowledgments

We thank our shepherd Kimberly Keeton and anonymous reviewers for their insightful and helpful comments, which improve the paper. The first author is supported by the National Key Research and Development Program of China (No. 2018YFB1003302). A gift from Intel supported parts of this work.

References

[1] Fio: Flexible i/o tester. http://freecode.com/projects/fio.

[2] Sqlite. https://www.sqlite.org/.

[3] Tpc benchmark c. http://www.tpc.org/tpcc/.

[4] AGARWAL, N., AND WENISCH, T. F. Thermostat: Application-transparent page management for two-tiered main memory. *ACM SIGARCH Computer Architecture News 45*, 1 (2017), 631–644.

[5] ARULRAJ, J., PAVLO, A., AND DULLOOR, S. R. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 707–722.

[6] CANO, I., AIYAR, S., ARORA, V., BHATTACHARYYA, M., CHAGANTI, A., CHEAH, C., CHUN, B. N., GUPTA, K., KHOT, V., AND KRISHNAMURTHY, A. Curator: Self-managing storage for enterprise clusters. In *NSDI* (2017), pp. 51–66.

[7] CHEN, E., APALKOV, D., DIAO, Z., DRISKILL-SMITH, A., DRUIST, D., LOTTIS, D., NIKITIN, V., TANG, X., WATTS, S., AND WANG, S. Advances and future prospects of spin-transfer torque random access memory. *IEEE Transactions on Magnetics 46*, 6 (2010), 1873–1878.

[8] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 133–146.

[9] DEBNATH, B., SENGUPTA, S., AND LI, J. Flashstore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 1414–1425.

[10] DONG, M., AND CHEN, H. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA* (2017), pp. 719–731.

[11] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 15.

[12] DULLOOR, S. R., ROY, A., ZHAO, Z., SUNDARAM, N., SATISH, N., SANKARAN, R., JACKSON, J., AND SCHWAN, K. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 15.

[13] FACEBOOK. Rocksdb. http://rocksdb.org.

[14] FANG, R., HSIAO, H.-I., HE, B., MOHAN, C., AND WANG, Y. High performance database logging using storage class memory.

[15] HITZ, D., LAU, J., AND MALCOLM, M. A. File system design for an nfs file server appliance. In *USENIX winter* (1994), vol. 94.

[16] INTEL. Intel memory latency checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[17] INTEL. Intel optane technology, 2018. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[18] JEONG, S., LEE, K., HWANG, J., LEE, S., AND WON, Y. Androstep: Android storage performance analysis tool. In *Software Engineering (Workshops)* (2013), vol. 13, pp. 327–340.

[19] KANNAN, S., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., WANG, Y., XU, J., AND PALANI, G. Designing a true direct-access file system with devfs. In *16th USENIX Conference on File and Storage Technologies* (2018), p. 241.

[20] KAWAHARA, T. Scalable spin-transfer torque ram technology for normally-off computing. *IEEE Design & Test of Computers 28* (2010), 52–63.

[21] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 460–477.

[22] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News* (2009), vol. 37, ACM, pp. 2–13.

[23] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: A capacity-optimized ssd cache for primary storage. In *USENIX Annual Technical Conference* (2014), pp. 501–512.

[24] MICRON. 3d-xpoint technology, 2017. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[25] MICRON. Battery-backed nvdimms, 2017. https://www.micron.com/products/dram-modules/nvdimm#/.

[26] NARAYANAN, D., AND HODSON, O. Whole-system persistence. *ACM SIGARCH Computer Architecture News 40*, 1 (2012), 401–410.

[27] ORACLE. Mysql. https://www.mysql.com/.

[28] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News 37*, 3 (2009), 24–33.

[29] WILCOX, M. Add support for nv-dimms to ext4, 2017.

[30] WU, X., AND REDDY, A. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), ACM, p. 39.

[31] XU, C., DONG, X., JOUPPI, N. P., AND XIE, Y. Design implications of memristor-based rram cross-point structures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011* (2011), IEEE, pp. 1–6.

[32] XU, J., AND SWANSON, S. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST* (2016), pp. 323–338.

[33] YANG, J. J., STRUKOV, D. B., AND STEWART, D. R. Memristive devices for computing. *Nature nanotechnology 8*, 1 (2013), 13.

[34] YANG, Z., HOSEINZADEH, M., ANDREWS, A., MAYERS, C., EVANS, D. T., BOLT, R. T., BHIMANI, J., MI, N., AND SWANSON, S. Autotiering: automatic data placement manager in multi-tier all-flash datacenter. In *Performance Computing and Communications Conference (IPCCC), 2017 IEEE 36th International* (2017), IEEE, pp. 1–8.

[35] ZHANG, Y., YANG, J., MEMARIPOUR, A., AND SWANSON, S. Mojim: A reliable and highly-available non-volatile memory system. In *ACM SIGARCH Computer Architecture News* (2015), vol. 43, ACM, pp. 3–18.

Orion: A Distributed File System for Non-Volatile Main Memories and RDMA-Capable Networks

Jian Yang Joseph Izraelevitz Steven Swanson
UC San Diego
{*jianyang, jizraelevitz, swanson*}*@eng.ucsd.edu*

Abstract

High-performance, byte-addressable non-volatile main memories (NVMMs) force system designers to rethink tradeoffs throughout the system stack, often leading to dramatic changes in system architecture. Conventional distributed file systems are a prime example. When faster NVMM replaces block-based storage, the dramatic improvement in storage performance makes networking and software overhead a critical bottleneck.

In this paper, we present *Orion*, a distributed file system for NVMM-based storage. By taking a clean slate design and leveraging the characteristics of NVMM and high-speed, RDMA-based networking, Orion provides high-performance metadata and data access while maintaining the byte addressability of NVMM. Our evaluation shows Orion achieves performance comparable to local NVMM file systems and outperforms existing distributed file systems by a large margin.

1 Introduction

In a distributed file system designed for block-based devices, media performance is almost the sole determiner of performance on the data path. The glacial performance of disks (both hard and solid state) compared to the rest of the storage stack incentivizes complex optimizations (e.g., queuing, striping, and batching) around disk accesses. It also saves designers from needing to apply similarly aggressive optimizations to network efficiency, CPU utilization, and locality, while pushing them toward software architectures that are easy to develop and maintain, despite the (generally irrelevant) resulting software overheads.

The appearance of fast non-volatile memories (e.g., Intel's 3D XPoint DIMMs [28]) on the processor's memory bus will offer an abrupt and dramatic increase in storage system performance, providing performance characteristics comparable to DRAM and vastly faster than either hard drives or SSDs. These non-volatile main memories (NVMM) upend the traditional design constraints of distributed file systems.

For an NVMM-based distributed file system, media access performance is no longer the major determiner of performance. Instead, network performance, software overhead, and data placement all play central roles. Furthermore, since NVMM is byte-addressable, block-based interfaces are no longer a constraint. Consequently, old distributed file systems squander NVMM performance — the previously negligible inefficiencies quickly become the dominant source of delay.

This paper presents Orion, a distributed file system designed from the ground up for NVMM and Remote Direct Memory Access (RDMA) networks. While other distributed systems [41, 55] have integrated NVMMs, Orion is the first distributed file system to systematically optimize for NVMMs throughout its design. As a result, Orion diverges from block-based designs in novel ways.

Orion focuses on several areas where traditional distributed file systems fall short when naively adapted to NVMMs. We describe them below.

Use of RDMA Orion targets systems connected with an RDMA-capable network. It uses RDMA whenever possible to accelerate both metadata and data accesses. Some existing distributed storage systems use RDMA as a fast transport layer for data access [10, 18, 62, 63, 71] but do not integrate it deeply into their design. Other systems [41, 55] adapt RDMA more extensively but provide object storage with customized interfaces that are incompatible with file system features such as unrestricted directories and file extents, symbolic links and file attributes.

Orion is the first full-featured file system that integrates RDMA deeply into all aspects of its design. Aggressive use of RDMA means the CPU is not involved in many transfers, lowering CPU load and improving scalability for handling incoming requests. In particular, pairing RDMA with NVMMs allows nodes to directly access remote storage without any target-side software overheads.

Software Overhead Software overhead in distributed files system has not traditionally been a critical concern. As such, most distributed file systems have used two-layer designs that divide the network and storage layers into separate modules.

	Read Latency 512 B	Bandwidth GB/s Read	Write
DRAM	80 ns	60	30
NVMM	300 ns	8	2
RDMA NIC	3 μs	5 (40 Gbps)	
NVMe SSD	70 μs	3.2	1.3

Table 1: **Characteristics of memory and network devices** We measure the fisrt 3 lines on Intel Sandy Bridge-EP platform with a Mellanox ConnectX-4 RNIC and an Intel DC P3600 SSD. NVMM numbers are estimated based on assumptions made in [75].

Two-layer designs trade efficiency for ease of implementation. Designers can build a user-level daemon that stitches together off-the-shelf networking packages and a local file system into a distributed file system. While expedient, this approach results in duplicated metadata, excessive copying, unnecessary event handling, and places user-space protection barriers on the critical path.

Orion merges the network and storage functions into a single, kernel-resident layer optimized for RDMA and NVMM that handles data, metadata, and network access. This decision allows Orion to explore new mechanisms to simplify operations and scale performance.

Locality RDMA is fast, but it is still several times slower than local access to NVMMs (Table 1). Consequently, the location of stored data is a key performance concern for Orion. This concern is an important difference between Orion and traditional block-based designs that generally distinguish between client nodes and a pool of centralized storage nodes [18, 53]. Pooling makes sense for block devices, since access latency is determined by storage, rather than network latency, and a pool of storage nodes simplifies system administration. However, the speed of NVMMs makes a storage pool inefficient, so Orion optimizes for locality. To encourage local accesses, Orion migrates durable data to the client whenever possible and uses a novel delegated allocation scheme to efficiently manage free space.

Our evaluation shows that Orion outperforms existing distributed file systems by a large margin. Relative to local NVMM filesystems, it provides comparable application-level performance when running applications on a single client. For parallel workloads, Orion shows good scalability: performance on an 8-client cluster is between 4.1× and 7.9× higher than running on a single node.

The rest of the paper is organized as follows. We discuss the opportunities and challenges of building a distributed file system utilizing NVMM and RDMA in Section 2. Section 3 gives an overview of Orion's architecture. We describe the design decisions we made to implement high-performance metadata access and data access in Sections 4 and 5 respectively. Section 6 evaluates these mechanisms. We cover related work in Section 7 and conclude in Section 8.

2 Background and Motivation

Orion is a file system designed for distributed shared NVMM and RDMA. This section gives some background on NVMM and RDMA and highlights the opportunities these technologies provide. Then, it discusses the inefficiencies inherent in running existing distributed file systems on NVMM.

2.1 Non-Volatile Main Memory

NVMM is comprised of nonvolatile DIMMs (NVDIMMs) attached to a CPU's memory bus alongside traditional DRAM DIMMs. Battery-backed NVDIMM-N modules are commercially available from multiple vendors [46], and Intel's 3DX-Point memory [28] is expected to debut shortly. Other technologies such as spin-torque transfer RAM (STT-RAM) [45], ReRAM [27] are in active research and development.

NVMMs appear as contiguous, persistent ranges of physical memory addresses [52]. Instead of using block-based interface, file systems can issue load and store instructions to NVMMs directly. NVMM file systems provide this ability via direct access (or "DAX"), which allows read and write system calls to bypass the page cache.

Researchers and companies have developed several file systems designed specifically for NVMM [15, 21, 25, 73, 74]. Other developers have adapted existing file systems to NVMM by adding DAX support [14, 70]. In either case, the file system must account for the 8-byte atomicity guarantees that NVMMs provide (compared to sector atomicity for disks). They also must take care to ensure crash consistency by carefully ordering updates to NVMMs using cache flush and memory barrier instructions.

2.2 RDMA Networking

Orion leverages RDMA to provide low latency metadata and data accesses. RDMA allows a node to perform one-sided read/write operations from/to memory on a remote node in addition to two-sided send/recv operations. Both user- and kernel-level applications can directly issue remote DMA requests (called *verbs*) on pre-registered memory regions (MRs). One-sided requests bypass CPU on the remote host, while two-sided requests require the CPU to handle them.

An RDMA NIC (RNIC) is capable of handling MRs registered on both virtual and physical address ranges. For MRs on virtual addresses, the RDMA hardware needs to translate from virtual addresses to DMA addresses on incoming packets. RNICs use a hardware pin-down cache [65] to accelerate lookups. Orion uses physically addressed DMA MRs, which do not require address translation on the RNIC, avoiding

the possibility of pin-down cache misses on large NVMM regions.

Software initiates RDMA requests by posting work queue entries (WQE) onto a pair of send/recv queues (a queue pair or "QP"), and polling for their completion from the completion queue (CQ). On completing a request, the RNIC signals completion by posting a completion queue entry (CQE).

A send/recv operation requires both the sender and receiver to post requests to their respective send and receive queues that include the source and destination buffer addresses. For one-sided transfers, the receiver grants the sender access to a memory region through a shared, secret 32-bit "rkey." When the receiver RNIC processes an inbound one-sided request with a matching rkey, it issues DMAs directly to its local memory without notifying the host CPU.

Orion employs RDMA as a fast transport layer, and its design accounts for several idiosyncrasies of RDMA:

Inbound verbs are cheaper Inbound verbs, including recv and incoming one-sided read/write, incur lower overhead for the target, so a single node can handle many more inbound requests than it can initiate itself [59]. Orion's mechanisms for accessing data and synchronizing metadata across clients both exploit this asymmetry to improve scalability.

RDMA accesses are slower than local accesses RDMA accesses are fast but still slower than local accesses. By combining the data measured on DRAM and the methodology introduced in a previous study [75], we estimate the one-sided RDMA NVMM read latency to be $\sim 9\times$ higher than local NVMM read latency for 64 B accesses, and $\sim 20\times$ higher for 4 KB accesses.

RDMA favors short transfers RNICs implement most of the RDMA protocol in hardware. Compared to transfer protocols like TCP/IP, transfer size is more important to transfer latency for RDMA because sending smaller packets involves fewer PCIe transactions [35]. Also, modern RDMA hardware can inline small messages along with WQE headers, further reducing latency. To exploit these characteristics, Orion aggressively minimizes the size of the transfers it makes.

RDMA is not persistence-aware Current RDMA hardware does not guarantee persistence for one-sided RDMA writes to NVMM. Providing this guarantee generally requires an extra network round-trip or CPU involvement for cache flushes [22], though a proposed [60] RDMA "commit" verb would provide this capability. As this support is not yet available, Orion ensures persistence by CPU involvement (see Section 5.3).

3 Design Overview

Orion is a distributed file system built for the performance characteristics of NVMM and RDMA networking. NVMM's low latency and byte-addressability fundamentally alter the

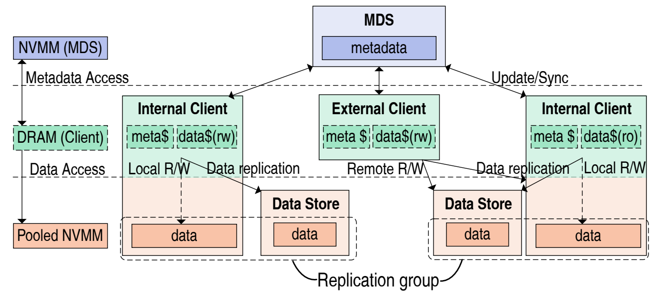


Figure 1: **Orion cluster organization** An Orion cluster consists of a metadata server, clients and data stores.

relationship among memory, storage, and network, motivating Orion to use a clean-slate approach to combine the file system and networking into a single layer. Orion achieves the following design goals:

- **Scalable performance with low software overhead:** Scalability and low-latency are essential for Orion to fully exploit the performance of NVMM. Orion achieves this goal by unifying file system functions and network operations and by accessing data structures on NVMM directly through RDMA.
- **Efficient network usage on metadata updates:** Orion caches file system data structures on clients. A client can apply file operations locally and only send the changes to the metadata server over the network.
- **Metadata and data consistency:** Orion uses a log-structured design to maintain file system consistency at low cost. Orion allows read parallelism but serializes updates for file system data structures across the cluster. It relies on atomically updated inode logs to guarantee metadata and data consistency and uses a new coordination scheme called *client arbitration* to resolve conflicts.
- **DAX support in a distributed file system:** DAX-style (direct load/store) access is a key benefit of NVMMs. Orion allows clients to access data in its local NVMM just as it could access a DAX-enabled local NVMM file system.
- **Repeated access become local access:** Orion exploits locality by migrating data to where writes occur and making data caching an integral part of the file system design. The log-structured design reduces the cost of maintaining cache coherence.
- **Reliability and data persistence:** Orion supports metadata and data replication for better reliability and availability. The replication protocol also guarantees data persistency.

The remainder of this section provides an overview of the Orion software stack, including its hardware and software organization. The following sections provide details of how Orion manages metadata (Section 4) and provides access to data (Section 5).

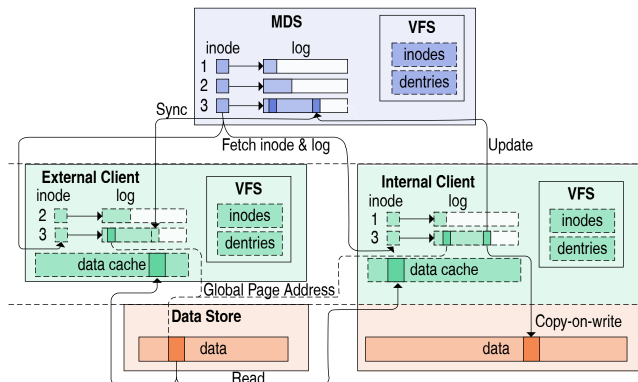


Figure 2: **Orion software organization** Orion exposes as a log-structured file system across MDS and clients. Clients maintain local copies of inode metadata and sync with the MDS, and access data at remote data stores or local NVMM directly.

3.1 Cluster Organization

An Orion cluster consists of a *metadata server (MDS)*, several *data stores (DSs)* organized in replication groups, and *clients* all connected via an RDMA network. Figure 1 shows the architecture of an Orion cluster and illustrates these roles.

The MDS manages metadata. It establishes an RDMA connection to each of the clients. Clients can propagate local changes to the MDS and retrieve updates made by other clients.

Orion allows clients to manage and access a global, shared pool of NVMMs. Data for a file can reside at a single DS or span multiple DSs. A client can access a remote DS using one-sided RDMA and its local NVMMs using load and store instructions.

Internal clients have local NVMM that Orion manages. Internal clients also act as a DSs for other clients. *External clients* do not have local NVMM, so they can access data on DSs but cannot store data themselves.

Orion supports replication of both metadata and data. The MDS can run as a high-availability pair consisting of a primary server and a mirror using Mojim [76]-style replication. Mojim provides low latency replication for NVMM by maintaining a single replica and only making updates at the primary.

Orion organizes DSs into replication groups, and the DSs in the group have identical data layouts. Orion uses broadcast replication for data.

3.2 Software Organization

Orion's software runs on the clients and the MDS. It exposes a normal POSIX interface and consists of kernel modules that manage file and metadata in NVMM and handle communication between the MDS and clients. Running in the kernel avoids the frequent context switches, copies, and kernel/user crossing that conventional two-layer distributed file systems designs require.

The file system in Orion inherits some design elements from NOVA [73, 74], a log-structured POSIX-compliant local NVMM file system. Orion adopts NOVA's highly-optimized mechanisms for managing file data and metadata in NVMM. Specifically, Orion's local file system layout, inode log data structure, and radix trees for indexing file data in DRAM are inherited from NOVA, with necessary changes to make metadata accessible and meaningful across nodes. Figure 2 shows the overall software organization of the Orion file system.

An Orion inode contains pointers to the head and tail of a metadata log stored in a linked list of NVMM pages. A log's entries record all modifications to the file and hold pointers to the file's data blocks. Orion uses the log to build virtual file system (VFS) inodes in DRAM along with indices that map file offsets to data blocks. The MDS contains the metadata structures of the whole file system including authoritative inodes and their logs. Each client maintains a local copy of each inode and its logs for the files it has opened.

Copying the logs to the clients simplifies and accelerates metadata management. A client can recover all metadata of a file by walking through the log. Also, clients can apply a log entry locally in response to a file system request and then propagate it to the MDS. A client can also tell whether an inode is up-to-date by comparing the local and remote log tail. An up-to-date log should be equivalent on both the client and the MDS, and this invariant is the basis for our metadata coherency protocol. Because MDS inode log entries are immutable except during garbage collection and logs are append-only, logs are amenable to direct copying via RDMA reads (see Section 4).

Orion distributes data across DSs (including the internal clients) and replicates the data within replication groups. To locate data among these nodes, Orion uses *global page addresses* (GPAs) to identify pages. Clients use a GPA to locate both the replication group and data for a page. For data reads, clients can read from any node within a replication group using the global address. For data updates, Orion performs a copy-on-write on the data block and appends a log entry reflecting the change in metadata (e.g., write offset, size, and the address to the new data block). For internal clients, the copy-on-write migrates the block into the local NVMM if space is available.

An Orion client also maintains a client-side data cache. The cache, combined with the copy-on-write mechanism, lets Orion exploit and enhance data locality. Rather than relying on the operating system's generic page cache, Orion manages DRAM as a customized cache that allows it to access cached pages using GPAs without a layer of indirection. This also simplifies cache coherence.

4 Metadata Management

Since metadata updates are often on an application's critical path, a distributed file system must handle metadata requests quickly. Orion's MDS manages all metadata updates and holds the authoritative, persistent copy of metadata. Clients cache metadata locally as they access and update files, and they must propagate changes to both the MDS and other clients to maintain coherence.

Below, we describe how Orion's metadata system meets both these performance and correctness goals using a combination of communication mechanisms, latency optimizations, and a novel arbitration scheme to avoid locking.

4.1 Metadata Communication

The MDS orchestrates metadata communication in Orion, and all authoritative metadata updates occur there. Clients do not exchange metadata. Instead, an Orion client communicates with the MDS to fetch file metadata, commit changes and apply changes committed by other clients.

Clients communicate with the MDS using three methods depending on the complexity of the operation they need to perform: (1) direct *RDMA reads*, (2) speculative and highly-optimized *log commits*, and (3) acknowledged *remote procedure calls* (RPCs).

These three methods span a range of options from simple/-lightweight (direct RDMA reads) to complex/heavyweight (RPC). We use RDMA reads from the MDS whenever possible because they do not require CPU intervention, maximizing MDS scalability.

Below, we describe each of these mechanisms in detail followed by an example. Then, we describe several additional optimizations Orion applies to make metadata updates more efficient.

RDMA reads Clients use one-sided RDMA reads to pull metadata from the MDS when needed, for instance, on file open. Orion uses wide pointers that contain a pointer to the client's local copy of the metadata as well as a GPA that points to the same data on the MDS. A client can walk through its local log by following the local pointers, or fetch the log pages from the MDS using the GPAs.

The clients can access the inode and log for a file using RDMA reads since NVMM is byte addressable. These accesses bypass the MDS CPU, which improves scalability.

Log commits Clients use log commits to update metadata for a file. The client first performs file operations locally by appending a log entry to the local copy of the inode log. Then it forwards the entry to the MDS and waits for completion.

Log commits use RDMA sends. Log entries usually fit in two cache lines, so the RDMA NIC can send them as inlined messages, further reducing latencies. Once it receives the acknowledgment for the send, the client updates its local log

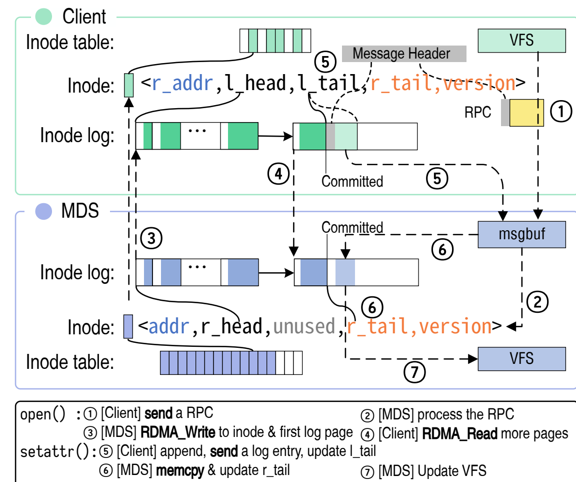


Figure 3: **Orion metadata communication** Orion maintains metadata structures such as inode logs on both MDS and clients. A client commit file system updates through Log Commits and RPCs.

tail, completing the operation. Orion allows multiple clients to commit log entries of a single inode without distributed locking using a mechanism called *client arbitration* that can resolve inconsistencies between inode logs on the clients (Section 4.3).

Remote procedure calls Orion uses synchronous remote procedure calls (RPCs) for metadata accesses that involve multiple inodes as well as operations that affect other clients (e.g., a file write with `O_APPEND` flag).

Orion RPCs use a send verb and an RDMA write. An RPC message contains an opcode along with metadata updates and/or log entries that the MDS needs to apply atomically. The MDS performs the procedure call and responds via one-sided RDMA write or message send depending on the opcode. The client blocks until the response arrives.

Example Figure 3 illustrates metadata communication. For `open()` (an RPC-based metadata update), the client allocates space for the inode and log, and issues an RPC ①. The MDS handles the RPC ② and responds by writing the inode along with the first log page using RDMA ③. The client uses RDMA to read more pages if needed and builds VFS data structures ④.

For a `setattr()` request (a log commit based metadata update), the client creates a local entry with the update and issues a log commit ⑤. It then updates its local tail pointer atomically after it has sent the log commit. Upon receiving the log entry, the MDS appends the log entry, updates the log tail ⑥, and updates the corresponding data structure in VFS ⑦.

RDMA Optimizations Orion avoids data copying within

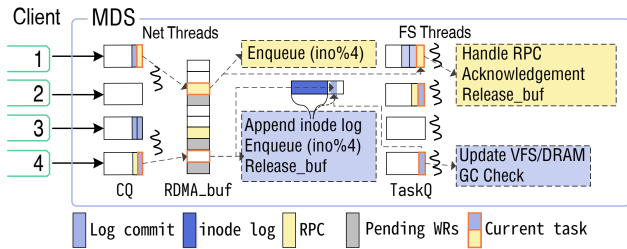


Figure 4: **MDS request handling** The MDS handles client requests in two stages: First, networking threads handle RDMA completion queue entries (CQEs) and dispatch them to file system threads. Next, file system threads handle RPCs and update the VFS.

a node whenever possible. Both client-initiated RDMA reads and MDS-initiated RDMA writes (e.g., in response to an RPC) target client file system data structures directly. Additionally, log entries in Orion contain extra space (shown as *message headers* in Figure 3) to accommodate headers used for networking. Aside from the DMA that the RNIC performs, the client copies metadata at most once (to avoid concurrent updates to the same inode) during a file operation.

Orion also uses *relative pointers* in file system data structures to leverage the linear addressing in kernel memory management. NVMM on a node appears as contiguous memory regions in both kernel virtual and physical address spaces. Orion can create either type of address by adding the relative pointer to the appropriate base address. Relative pointers are also meaningful across power failures.

4.2 Minimizing Commit Latency

The latency of request handling, especially for log commits, is critical for the I/O performance of the whole cluster. Orion uses dedicated threads to handle per-client receive queues as well as file system updates. Figure 4 shows the MDS request handling process.

For each client, the MDS registers a small (256 KB) portion of NVMM as a communication buffer. The MDS handles incoming requests in two stages: A *network thread* polls the RDMA completion queues (CQs) for work requests on pre-posted RDMA buffers and dispatches the requests to *file system threads*. As an optimization, the MDS prioritizes log commits by allowing network threads to append log entries directly. Then, a file system thread handles the requests by updating file system structures in DRAM for a log commit or serving the requests for an RPC. Each file system thread maintains a FIFO containing pointers to updated log entries or RDMA buffers holding RPC requests.

For a log commit, a network thread reads the inode number, appends the entry by issuing non-temporal moves and then atomically updates the tail pointer. At this point, other clients can read the committed entry and apply it to their local copy

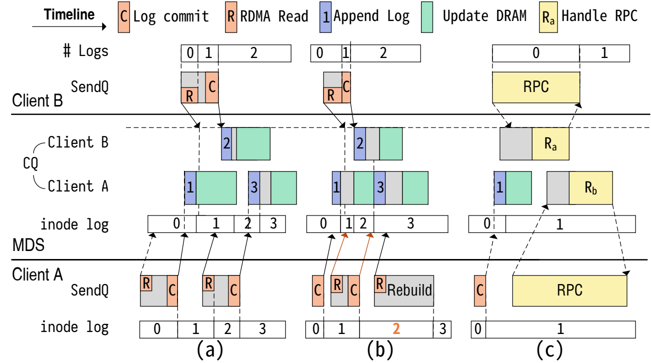


Figure 5: **Metadata consistency in Orion** The inode log on Client A is consistent after (a) updating the log entry committed by another client using RDMA reads, (c) issuing an RPC, and (b) rebuilding the log on conflicts.

of the inode log. The network thread then releases the recv buffer by posting a recv verb, allowing its reuse. Finally, it dispatches the task for updating in-DRAM data structures to a file system thread based on the inode number.

For RPCs, the network thread dispatches the request directly to a file system thread. Each thread processes requests to a subset of inodes to ensure better locality and less contention for locks. The file system threads use lightweight journals for RPCs involving inodes that belong to multiple file system threads.

File system threads perform garbage collection (GC) when the number of "dead" entries in a log becomes too large. Orion rebuilds the inode log by copying live entries to new log pages. It then updates the log pointers and increases the version number. Orion makes this update atomic by packing the version number and tail pointer into 64 bits. The thread frees stale log pages after a delay, allowing ongoing RDMA reads to complete. Currently we set the maximal size of file writes in a log entry to be 512 MB.

4.3 Client Arbitration

Orion allows multiple clients to commit log entries to a single inode at the same time using a mechanism called *client arbitration* rather than distributed locking. Client arbitration builds on the following observations:

1. Handling an inbound RDMA read is much cheaper than sending an outbound write. In our experiments, a single host can serve over 15 M inbound reads per second but only 1.9 M outbound writes per second.
2. For the MDS, CPU time is precious. Having the MDS initiate messages to maintain consistency will reduce Orion performance significantly.
3. Log append operations are lightweight: each one takes around just 500 CPU cycles.

A client commits a log entry by issuing a send verb and

polling for its completion. The MDS appends log commits based on arrival order and updates log tails atomically. A client can determine whether a local inode is up-to-date by comparing the log length of its local copy of the log and the authoritative copy at the MDS. Clients can check the length of an inode's log by retrieving its tail pointer with an RDMA read.

The client issues these reads in the background when handling an I/O request. If another client has modified the log, the client detects the mismatch and fetches the new log entries using additional RDMA reads and retries.

If the MDS has committed multiple log entries in a different order due to concurrent accesses, the client blocks the current request and finds the last log entry that is in sync with the MDS, it then fetches all following log entries from the MDS, rebuilds its in-DRAM structures, and re-executes the user request.

Figure 5 shows the three different cases of concurrent accesses to a single inode. In (a), the client A can append the log entry #2 from client B by extending its inode log. In (b), the client A misses the log entry #2 committed by client B, so it will rebuild the inode log on the next request. In (c), the MDS will execute concurrent RPCs to the same inode sequentially, and the client will see the updated log tail in the RPC acknowledgment.

A rebuild occurs when all of the following occur at the same time: (1) two or more clients access the same file at the same time and one of the accesses is log commit, (2) one client issues two log commits consecutively, and (3) the MDS accepts the log commit from another client after the client RDMA reads the inode tail but before the MDS accepts the second log commit.

In our experience this situation happens very rarely, because the "window of vulnerability" – the time required to perform a log append on the MDS – is short. That said, Orion lets applications identify files that are likely targets of intensive sharing via an `ioctl`. Orion uses RPCs for all updates to these inodes in order to avoid rebuilds.

5 Data Management

Orion pools NVMM spread across internal clients and data stores. A client can allocate and access data either locally (if the data are local) or remotely via one-sided RDMA. Clients use local caches and migration during copy-on-write operations to reduce the number of remote accesses.

5.1 Delegated Allocation

To avoid allocating data on the critical path, Orion uses a distributed, two-stage memory allocation scheme.

The MDS keeps a bitmap of all the pages Orion manages. Clients request large chunks of storage space from the MDS via an RPC. The client can then autonomously allocate space

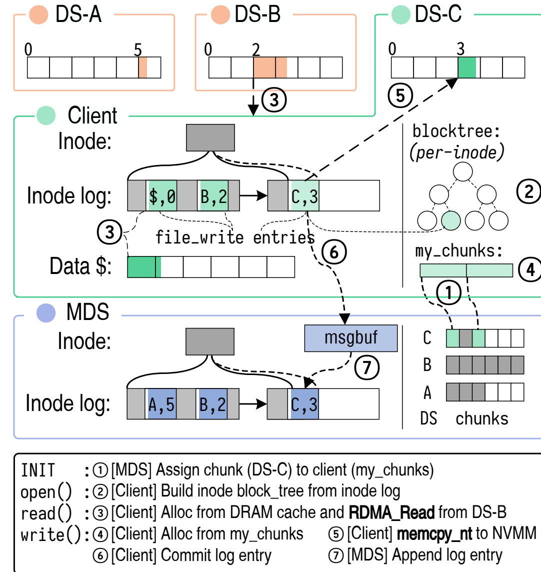


Figure 6: **Orion data communication** Orion allows clients manage and access data independently.

within those chunks. This design frees the MDS from managing fine-grain data blocks, and allows clients to allocate pages with low overhead.

The MDS allocates internal clients chunks of its local NVMM when possible since local writes are faster. As a result, most of their writes go to local NVMM.

5.2 Data Access

To read file data, a client either communicates with the DS using one-sided RDMA or accesses its local NVMM via DAX (if it is an internal client and the data is local). Remote reads use one-sided RDMA reads to retrieve existing file data and place it in local DRAM pages that serve as a cache for future reads.

Remote writes can also be one-sided because allocation occurs at the client. Once the transfer is complete, the client issues a log commit to the MDS.

Figure 6 demonstrates Orion's data access mechanisms. A client can request a block chunk from the MDS via an RPC ①. When the client opens a file, it builds a radix tree for fast lookup from file offsets to log entries ②. When handling a `read()` request, the client reads from the DS (DS-B) to its local DRAM and update the corresponding log entry ③. For a `write()` request, it allocates from its local chunk ④ and issues `memcpy_nt()` and `sfence` to ensure that the data reaches its local NVMM (DS-C) ⑤. Then a log entry containing information such as the GPA and size is committed to the MDS ⑥. Finally, the MDS appends the log entry ⑦.

5.3 Data Persistence

Orion always ensures that metadata is consistent, but, like many file systems, it can relax the consistency requirement on data based on user preferences and the availability of replication.

The essence of Orion's data consistency guarantee is the extent to which the MDS delays the log commit for a file update. For a weak consistency guarantee, an external client can forward a speculative log commit to the MDS before its remote file update has completed at a DS. This consistency level is comparable to the write-back mode in ext4 and can result in corrupted data pages but maintains metadata integrity. For strong data consistency that is comparable to NOVA and the data journaling mode in ext4, Orion can delay the log commit until after the file update is persistent at multiple DSs in the replication group.

Achieving strong consistency over RDMA is hard because RDMA hardware does not provide a standard mechanism to force writes into remote NVMM. For strongly consistent data updates, our algorithm is as follows.

A client that wishes to make a consistent file update uses copy-on-write to allocate new pages on all nodes in the appropriate replica group, then uses RDMA writes to update the pages. In parallel, the client issues a speculative log commit to the MDS for the update.

DSs within the replica group detect the RDMA writes to new pages using an RDMA trick: when clients use RDMA writes on the new pages, they include the page's global address as an *immediate value* that travels to the target in the RDMA packet header. This value appears in the target NIC's completion queue, so the DS can detect modifications to its pages. For each updated page, the DS forces the page into NVMM and sends an acknowledgment via a small RDMA write to the MDS, which processes the client's log commit once it reads a sufficient number of acknowledgments in its DRAM.

5.4 Fault Tolerance

The high performance and density of NVMM makes the cost of rebuilding a node much higher than recovering it. Consequently, Orion makes its best effort to recover the node after detecting an error. If the node can recover (e.g., after a power failure and most software bugs), it can rejoin the Orion cluster and recover to a consistent state quickly. For NVMM media errors, module failures, or data-corrupting bugs, Orion rebuilds the node using the data and metadata from other replicas. It uses relative pointers and global page addresses to ensure metadata in NVMM remain meaningful across power failures.

In the metadata subsystem, for MDS failures, Orion builds a Mojim-like [76] high-availability pair consisting of a primary MDS and a mirror. All metadata updates flow to the primary MDS, which propagates the changes to the mirror. When the primary fails, the mirror takes over and journals all the incoming requests while the primary recovers.

In the data subsystem, for DS failures, the DS journals the immediate values of incoming RDMA write requests in a circular buffer. A failed DS can recover by obtaining the pages committed during its downtime from a peer DS in the same replication group. When there are failed nodes in a replication group, the rest of the nodes work in the strong data consistency mode introduced in Section 5.3 to ensure successful recovery in the event of further failures.

6 Evaluation

In this section, we evaluate the performance of Orion by comparing it to existing distributed file systems as well as local file systems. We answer the following questions:

- How does Orion's one-layer design affect its performance compared to existing two-layer distributed file systems?
- How much overhead does managing distributed data and metadata add compared to running a local NVMM file system?
- How does configuring Orion for different levels of reliability affect performance?
- How scalable is Orion's MDS?

We describe the experimental setup and then evaluate Orion with micro- and macrobenchmarks. Then we measure the impact of data replication and the ability to scale over parallel workloads.

6.1 Experimental Setup

We run Orion on a cluster with 10 nodes configured to emulate persistent memory with DRAM. Each node has two quad-core Intel Xeon (Westmere-EP) CPUs with 48 GB of DRAM, with 32 GB configured as an emulated `pmem` device. Each node has an RDMA NIC (Mellanox ConnectX-2 40 Gbps HCA) running in Infiniband mode and connects to an Infiniband switch (QLogic 12300). We disabled the Direct Cache Access feature on DSs. To demonstrate the impact to co-located applications, we use a dedicated core for issuing and handling RDMA requests on each client.

We build our Orion prototype on the Linux 4.10 kernel with the RDMA verb kernel modules from Mellanox OFED [42]. The file system in Orion reuses code from NOVA but adds ~8K lines of code to support distributed functionalities and data structures. The networking module in Orion is built from scratch and comprises another ~8K lines of code.

We compare Orion with three distributed file systems Ceph [69], Gluster [19], and Octopus [41] running on the same RDMA network. We also compare Orion to ext4 mounted on a remote iSCSI target hosting a ramdisk using iSCSI Extension over RDMA (iSER) [12] (denoted by Ext4/iSER), which provides the client with private access to

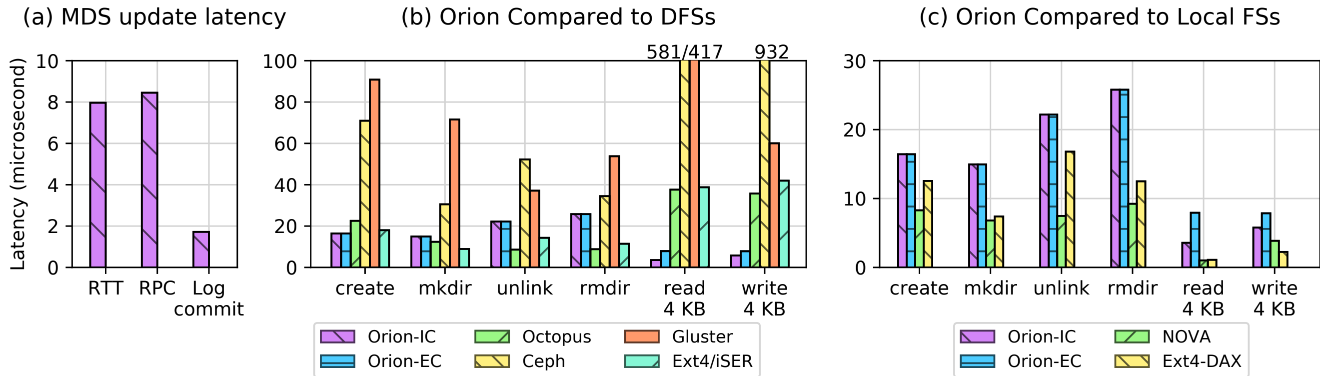


Figure 7: **Average latency of Orion metadata and data operations** Orion is built on low-latency communication primitive (a). These lead to basic file operation latencies that are better than existing remote-access storage system (b) and within a small factor of local NVMM file systems (c).

a remote block device. Finally, we compare our system with two local DAX file systems: NOVA [73, 74] and ext4 in DAX mode (ext4-DAX).

6.2 Microbenchmarks

We begin by measuring the networking latency of log commits and RPCs. Figure 7(a) shows the latency of a log commit and an RPC compared to the network round trip time (RTT) using two sends verbs. Our evaluation platform has a network round trip time of 7.96 μs. The latency of issuing an Orion RPC request and obtaining the response is 8.5 μs. Log commits have much lower latency since the client waits until receiving the acknowledgment of an RDMA send work request, which takes less than half of the network round trip time: they complete in less than 2 μs.

Figure 7(b) shows the metadata operation latency on Orion and other distributed file systems. We evaluated basic file system metadata operations such as `create`, `mkdir`, `unlink`, `rmdir` as well as reading and writing random 4 KB data using FIO [6]. Latencies for Ceph and Gluster are between 34% and 443% higher than Orion.

Octopus performs better than Orion on `mkdir`, `unlink` and `rmdir`, because Octopus uses a simplified file system model: it maintains all files and directories in a per-server hash table indexed by their full path names and it assigns a fixed number of file extents and directory entries to each file and directory. This simplification means it cannot handle large files or directories.

Ext4/iSER outperforms Orion on some metadata operations because it considers metadata updates complete once they enter the block queue. In contrast, NVMM-aware systems (such as Orion or Octopus) report the full latency for persistent metadata updates. The 4 KB read and write measurements in the figure give a better measure of I/O latency – Orion outperforms Ext4/iSER configuration by between 4.9× and 10.9×.

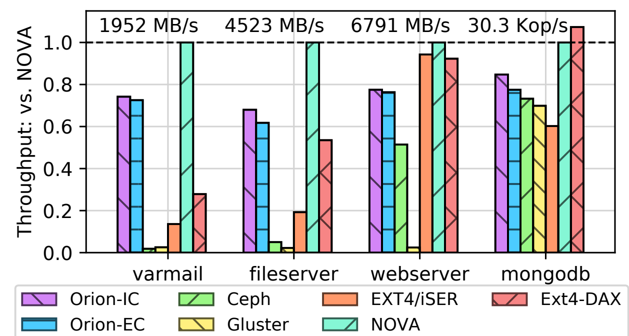


Figure 8: **Application performance on Orion** The graph is normalized to NOVA, and the annotations give NOVA's performance. For write-intensive workloads, Orion outperforms Ceph and Gluster by a wide margin.

For file reads and writes, Orion has the lowest latency among all the distributed file systems we tested. For internal clients (Orion-IC), Orion's 4 KB read latency is 3.6 μs and 4 KB write latency of 5.8 μs. For external clients (Orion-EC), the write latency is 7.9 μs and read latency is similar to internal clients because of client-side caching. For cache misses, read latency is 7.9 μs.

We compare Orion to NOVA and Ext4-DAX in Figure 7(c). For metadata operations, Orion sends an RPC to the MDS on the critical path, increasing latency by between 98% to 196% compared to NOVA and between 31% and 106% compared to Ext4-DAX. If we deduct the networking round trip latency, Orion increases software overheads by 41%.

6.3 Macrobenchmarks

We use three Filebench [64] workloads (varmail, fileserver and webserver) as well as MongoDB [4] running YCSB's [16]

Workload	# Threads	# Files	Avg. File Size	R/W Size	Append Size
varmail	8	30 K	16 KB	1 MB	16 KB
fileserver	8	10 K	128 KB	1 MB	16 KB
webserver	8	50 K	64 KB	1 MB	8 KB
mongodb	12	YCSB-A, RecordCount=1M, OpCount=10M			

Table 2: **Application workload characteristics** This table includes the configurations for three filebench workloads and the properties of YCSB-A.

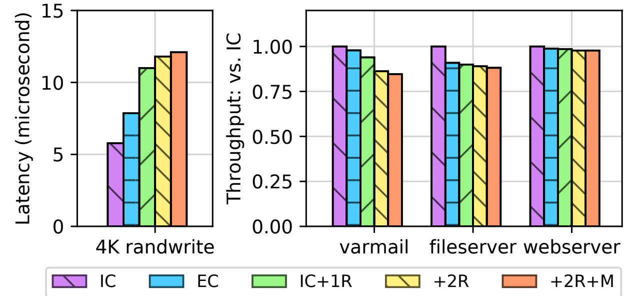


Figure 9: **Orion data replication performance** Updating a remote replica adds significantly to random write latency, but the impact on overall benchmark performance is small.

Workload A (50% read/50% update) to evaluate Orion. Table 2 describes the workload characteristics. We could not run these workloads on Octopus because it limits the directory entries and the number of file extents, and it ran out of memory when we increased those limits to meet the workloads' requirements.

Figure 8 shows the performance of Orion internal and external clients along with other file systems. For filebench workloads, Orion outperforms Gluster and Ceph by a large margin (up to 40×). We observe that the high synchronization cost in Ceph and Gluster makes them only suitable for workloads with high queue depths, which are less likely on NVMM because media access latency is low. For MongoDB, Orion outperforms other distributed file systems by a smaller margin because of the less intensive I/O activities.

Although Ext4/iSER does not support sharing, file system synchronization (e.g., `fsync()`) is expensive because it flushes the block queue over RDMA. Orion outperforms Ext4/iSER in most workloads, especially for those that require frequent synchronization, such as varmail (with 4.5× higher throughput). For webserver, a read-intensive workload, Ext4/iSER performs better than local Ext4-DAX and Orion because it uses the buffer cache to hold most of the data and does not flush writes to storage.

Orion achieves an average of 73% of NOVA's throughput. It also outperforms Ext4-DAX on metadata and I/O intensive workloads such as varmail and filebench. For Webserver, a read-intensive workload, Orion is slower because it needs to communicate with the MDS.

The performance gap between external clients and internal clients is small in our experiments, especially for write requests. This is because our hardware does not support the optimized cache flush instructions that Intel plans to add in the near future [51]. Internal clients persist local writes using `clflush` or non-temporal memory copy with fences; both of which are expensive [76].

6.4 Metadata and Data Replication

Figure 9 shows the performance impact of metadata and data replication. We compare the performance of a single internal client (IC), a single external client (EC), an internal client with one and two replicas (IC+1R, +2R), and an internal client with two replicas and MDS replication (+2R+M). For a 4 KB write, it takes an internal client 12.1 μs to complete with our strongest reliability scheme (+2R+M), which is 2.1× longer than internal client and 1.5× longer than an external client. For filebench workloads, overall performance decreases by between 2.3% and 15.4%.

6.5 MDS Scalability

Orion uses a single MDS with a read-only mirror to avoid the overhead of synchronizing metadata updates across multiple nodes. However, using a single MDS raises scalability concerns. In this section, we run an MDS paired with 8 internal clients to evaluate the system under heavy metadata traffic.

We measure MDS performance scalability by stressing it with different types of requests: client initiated inbound RDMA reads, log commits, and RPCs. Figure 10 measures throughput for the MDS handling concurrent requests from different numbers of clients. For inbound RDMA reads (a), each client posts RDMA reads for an 8-byte field, simulating reading the log tail pointers of inodes. In (b) the client sends 64-byte log commits spread across 10,000 inodes. In (c) the clients send 64-byte RPCs and the MDS responds with 32-byte acknowledgments. Each RPC targets one of the 10,000 inodes. Finally, in (d) we use FIO to perform 4 KB random writes from each client to private file.

Inbound RDMA reads have the best performance and scale well: with eight clients, the MDS performs 13.8 M RDMA reads per second – 7.2× the single-client performance. For log commits, peak throughput is 2.5 M operations per second with eight clients – 4.1× the performance for a single client. Log commit scalability is lower because the MDS must perform the log append in software. The MDS can perform 772 K RPCs per second with seven clients (6.2× more than a single). Adding an eighth does not improve performance due to contention among threads polling CQEs and threads handling RPCs. The FIO write test shows good scalability – 7.9× improvement with eight threads. Orion matches NOVA performance with two clients and out-performs NOVA by

Figure 10: **Orion metadata scalability for MDS metadata operations and FIO 4K randwrite** Orion exhibits good scalability with rising node counts for inbound 8 B RDMA reads (a), 64 B log commits (b), RPCs (c), and random writes (d).

4.1× on eight clients.

Orion is expected to have good scaling under these conditions. Similar to other RDMA based studies, Orion is suitable to be deployed on networks with high bisectional bandwidth and predictable end-to-end latency, such as rack-scale computers [17, 39]. In these scenarios, the single MDS design is not a bottleneck in terms of NVMM storage, CPU utilization, or networking utilization. Orion metadata consumes less than 3% space compared to actual file data in our experiments. Additionally, metadata communication is written in tight routines running on dedicated cores, where most of the messages fit within two cache lines. Previous works [7,40] show similar designs can achieve high throughput with a single server.

In contrast, several existing distributed file systems [8, 19, 30, 69] target data-center scale applications, and use mechanisms designed for these conditions. In general, Orion's design is orthogonal to the mechanisms used in these systems, such as client side hashing [19] and partitioning [69], which could be integrated into Orion as future work. On the other hand, we expect there may be other scalability issues such as RDMA connection management and RNIC resource contention that need to be addressed to allow further scaling for Orion. We leave this exploration as future work.

7 Related work

Orion combines ideas from NVMM file systems, distributed file systems, distributed shared memory, user level file systems with trusted services, and recent work on how to best utilize RDMA. Below, we place Orion in context relative to key related work in each of these areas.

NVMM file systems Emerging NVMM technologies have inspired researchers to build a menagerie NVMM-specific file systems. Orion extends many ideas and implementation details from NOVA [73,74] to the distributed domain, especially in how Orion stores and updates metadata. Orion also relies on key insights developed in earlier systems [15,21,25,68,70,72].

Distributed file systems There are two common ways to provide distributed file accesses: the first is to deploy a Clustered File System (CFS) [8,11,20,26,31,54] running on block devices exposed via a storage area network (SAN) protocol like iSCSI [53], Fiber Channel or NVMe Over Fabrics [18]. They use RDMA to accelerate the data path [12, 18, 61] and they can accelerate data transfers using zero-copy techniques while preserving the block-based interface.

The second is to build a Distributed File System (DFS) [8, 9, 19, 30, 37, 38, 47, 57, 57, 69] that uses local file systems running on a set of servers to create a single, shared file system image. They consist of servers acting in dedicated roles and communicating using customized protocols to store metadata and data. Some distributed file systems use RDMA as a drop-in replacement of existing networking protocols [10,18,63,71] while preserving the local file system logic.

Their diversity reflects the many competing design goals they target. They vary in the interfaces they provide, the consistency guarantees they make, and the applications and deployment scenarios they target. However, these systems target hard drives and SSDs and include optimizations such as queuing striping and DRAM caching. Orion adds to this diversity by rethinking how a full-featured distributed file system can fully exploit the characteristics of NVMM and RDMA.

Octopus [41] is a distributed file system built for RDMA and NVMM. Compared to Orion, its design has several limitations. First, Octopus assumes a simplifed file system model and uses a static hash table to organize file system metadata and data, which preventing it from running complex workloads. Second, Octopus uses client-side partitioning. This design restricts access locality: as the number of peers increases, common file system tasks such as traversing a directory become expensive. Orion migrates data to local NVMM to improve locality. Finally, Octopus does not provide provisions for replication of either data or metadata, so it cannot tolerate node failures.

Trusted file system services Another research trend is to decouple file system control plane and data plane, and build userspace file systems [36, 48, 68] with trusted services to reduce the number of syscalls. Orion MDS plays a similar role as the trusted service. However, Orion heavily leverages

kernel file system mechanisms, as well as the linear addressing of kernel virtual addresses and DMA addresses. In order to support DAX accesses, extending a uerspace file system to a distributed setting must deal with issues such as large page tables, sharing and protection across processes and page faults, which are all not RDMA friendly.

Distributed shared memory There has been extensive research on distributed shared memory (DSM) systems [44, 49, 50], and several of them have considered the problem of distributed, shared persistent memory [29, 55, 56]. DSM systems expose a simpler interface than a file system, so the designers have made more aggressive optimizations in many cases. However, that makes adapting existing software to use them more challenging.

Hotpot [55] is a distributed shared persistent memory system that allows applications to commit fine-grained objects on memory mapped NVMM files. It is built on a customized interface, requiring application modification. Hotpot uses a multi-stage commit protocol for consistency, while Orion uses client-side updates to ensure file system consistency.

Mojim [76] provides fine-grained replication on NVMM, and Orion uses this technique to implement metadata replication.

RDMA-optimized applications Many existing works explore how RDMA can accelerate data center applications, such as key-value stores [23, 33, 43], distributed transaction systems [13, 24, 34], distributed memory allocators [5, 23, 66, 67] and RPC implementations [35, 58]. There are several projects using RDMA protocols targeting to accelerate existing distributed storages [3, 32] or work as a middle layer [1, 2]. Orion differs from these systems in that it handles network requests directly within file systems routines, and uses RDMA to fully exploit NVMM's byte-addressability.

8 Conclusion

We have described and implemented Orion, a file system for distributed NVMM and RDMA networks. By combining file system functions and network operations into a single layer, Orion provides low latency metadata accesses and allows clients to access their local NVMMs directly while accepting remote accesses. Our evaluation shows that Orion outperforms existing NVMM file systems by a wide margin, and it scales well over multiple clients on parallel workloads.

Acknowledgments

The authors would like to thank our shepherd Florentina Popovici, and the anonymous reviewers for their insightful comments and suggestions. We thank members of Non-Volatile Systems Lab for their input. The work described in this paper is supported by a gift from Huawei.

References

[1] Accelio - Open-Source IO, Message, and RPC Acceleration Library. https://github.com/accelio/accelio.

[2] Alluxio - Open Source Memory Speed Virtual Distributed Storage. https://www.alluxio.org/.

[3] Crail: A fast multi-tiered distributed direct access file system. https://github.com/zrlio/crail.

[4] MongoDB Community. https://www.mongodb.com/community.

[5] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, Boston, MA, 2018.

[6] Jens Axboe. Flexible I/O Tester. https://github.com/axboe/fio.

[7] Mahesh Balakrishnan, Dahlia Malkhi, John D Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. Corfu: A distributed shared log. *ACM Transactions on Computer Systems (TOCS)*, 31(4):10, 2013.

[8] Peter J Braam. The Lustre storage architecture, 2004.

[9] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[10] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. NFS over RDMA. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, pages 196–208. ACM, 2003.

[11] Philip H Carns, Walter B Ligon III, Robert B Ross, Rajeev Thakur, et al. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th annual Linux showcase and conference*, pages 391–430, 2000.

[12] Mallikarjun Chadalapaka, Hemal Shah, Uri Elzur, Patricia Thaler, and Michael Ko. A Study of iSCSI Extensions for RDMA (iSER). In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, NICELI '03, pages 209–219. ACM, 2003.

[13] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 26. ACM, 2016.

[14] Dave Chinner. xfs: DAX support. https://lwn.net/Articles/635514/. Accessed 2019-01-05.

[15] 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, pages 133–146, 2009.

[16] 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.

[17] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2c2: A network stack for rack-scale computers. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 551–564. ACM, 2015.

[18] Patrice Couvert. High speed IO processor for NVMe over fabric (NVMeoF). *Flash Memory Summit*, 2016.

[19] Alex Davies and Alessandro Orsaria. Scale out with GlusterFS. *Linux Journal*, 2013(235):1, 2013.

[20] Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent, Dave Noveck, Tom Talpey, and Mark Wittle. The Direct Access File System. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 175–188, Berkeley, CA, USA, 2003. USENIX Association.

[21] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA*, pages 719–731, 2017.

[22] Chet Douglas. RDMA with PMEM, Software mechanisms for enabling access to remote persistent memory. http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf. Accessed 2019-01-05.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[24] 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*, SOSP '15, pages 54–70. ACM, 2015.

[25] 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, pages 15:1–15:15. ACM, 2014.

[26] Chandramohan A Thekkath Edward K. Lee. Petal: Distributed Virtual Disks. *ASPLOS VII*, pages 1–9, 1996.

[27] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara, and G. Hush. A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 338–339, Feb 2014.

[28] Mike Ferron-Jones. A New Breakthrough in Persistent Memory Gets Its First Public Demo. https://itpeernetwork.intel.com/new-breakthrough-persistent-memory-first-public-demo/. Accessed 2019-01-05.

[29] João Garcia, Paulo Ferreira, and Paulo Guedes. The PerDiS FS: A transactional file system for a distributed persistent store. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 189–194. ACM, 1998.

[30] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43. ACM, 2003.

[31] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *ACM SIGMETRICS Performance Evaluation Review*, volume 25, pages 272–284. ACM, 1997.

[32] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K Panda. High performance design for HDFS with byte-addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing*, page 8. ACM, 2016.

[33] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.

[34] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *OSDI*, volume 16, pages 185–201, 2016.

[35] Anuj Kalia Michael Kaminsky and David G Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference*, page 437, 2016.

[36] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a true direct-access file system with DevFS. In *16th USENIX Conference on File and Storage Technologies*, page 241, 2018.

[37] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. 2000.

[38] Edward K. Lee, Chandramohan A. Thekkath, and Timothy Mann. Frangipani: A Scalable Distributed File System. In *Proceedings of the sixteenth ACM symposium on Operating systems principles - SOSP '97*, volume 31, pages 224–237. ACM Press, 1997.

[39] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony IT Rowstron, Hugh Williams, and Xiaohan Zhao. Xfabric: A reconfigurable in-rack network for rack-scale computers. In *NSDI*, volume 16, pages 15–29, 2016.

[40] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 476–488. ACM, 2015.

[41] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, 2017.

[42] Mellanox. Mellanox OFED for Linux User Manual, 2017.

[43] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[44] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[45] H. Noguchi, K. Ikegami, K. Kushida, K. Abe, S. Itai, S. Takaya, N. Shimomura, J. Ito, A. Kawasumi, H. Hara, and S. Fujita. A 3.3ns-access-time 71.2uW/MHz 1Mb embedded STT-MRAM using physically eliminated read-disturb scheme and normally-off memory architecture. In *Solid-State Circuits Conference (ISSCC), 2015 IEEE International*, pages 1–3, Feb 2015.

[46] Colby Parkinson. NVDIMM-N: Where are we now? https://www.micron.com/about/blogs/2017/august/nvdimm-n-where-are-we-now. Accessed 2019-01-05.

[47] Swapnil Patil and Garth A Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *FAST*, volume 11, pages 13–13, 2011.

[48] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):11, 2016.

[49] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. A survey of distributed shared memory systems. In *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 1, pages 74–84. IEEE, 1995.

[50] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):63–71, 1996.

[51] Andy Rudoff. Processor Support for NVM Programming. http://www.snia.org/sites/default/files/AndyRudoff_Processor_Support_NVM.pdf. Accessed 2019-01-05.

[52] Arthur Sainio. NVDIMM: Changes are Here So Whats Next. *In-Memory Computing Summit*, 2016.

[53] Julian Satran, Kalman Meth, C Sapuntzakis, M Chadalapaka, and E Zeidner. Internet small computer systems interface (iSCSI), 2004.

[54] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 231–244, 2002.

[55] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337. ACM, 2017.

[56] Marc Shapiro and Paulo Ferreira. Larchant-RDOSS: a distributed shared persistent memory and its garbage collector. In *International Workshop on Distributed Algorithms*, pages 198–214. Springer, 1995.

[57] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[58] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. DaRPC: Data center RPC. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[59] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is Faster Than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 1–15. ACM, 2017.

[60] Talpey and Pinkerton. RDMA Durable Write Commit. https://tools.ietf.org/html/draft-talpey-rdma-commit-00. Accessed 2019-01-05.

[61] T Talpey and G Kamer. High Performance File Serving With SMB3 and RDMA via SMB Direct. In *Storage Developers Conference*, 2012.

[62] Haodong Tang, Jian Zhang, and Fred Zhang. Accelerating Ceph with RDMA and NVMeoF. In *14th Annual OpenFabrics Alliance (OFA) Workshop*, 2018.

[63] Wittawat Tantisiriroj, Seung Woo Son, Swapnil Patil, Samuel J Lang, Garth Gibson, and Robert B Ross. On the duality of data-intensive file system design: reconciling HDFS and PVFS. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 67. ACM, 2011.

[64] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41, 2016.

[65] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, pages 308–314. IEEE, 1998.

[66] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Clemens Lutz, Martin Schmatz, and Thomas R Gross. Rstore: A direct-access DRAM-based data store. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 674–685. IEEE, 2015.

[67] Shin-Yeh Tsai and Yiying Zhang. LITE: Kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 306–324. ACM, 2017.

[68] 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*, page 14. ACM, 2014.

[69] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[70] Matthew Wilcox. Add support for NV-DIMMs to Ext4. https://lwn.net/Articles/613384/. Accessed 2019-01-05.

[71] J. Wu, P. Wyckoff, and Dhabaleswar Panda. PVFS over InfiniBand: Design and performance evaluation. In *2003 International Conference on Parallel Processing, 2003. Proceedings.*, pages 125–132. IEEE, 2003.

[72] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11. ACM, 2011.

[73] 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.

[74] 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 *26th Symposium on Operating Systems Principles (SOSP '17)*, pages 478–496, 2017.

[75] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2015.

[76] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18. ACM, 2015.

INSTalytics: Cluster Filesystem Co-design for Big-data Analytics

Muthian Sivathanu[⋆], Midhul Vuppalapati[⋆], Bhargav S. Gulavani[⋆], Kaushik Rajan[⋆],
Jyoti Leeka[⋆], Jayashree Mohan[†], Piyus Kedia[⋄*]

[⋆]*Microsoft Research India,* [†]*Univ. of Texas Austin,* [⋄]*IIIT Delhi*

Abstract

We present the design, implementation, and evaluation of INSTalytics a co-designed stack of a cluster file system and the compute layer, for efficient big data analytics in large-scale data centers. INSTalytics amplifies the well-known benefits of data partitioning in analytics systems; instead of traditional partitioning on one dimension, INSTalytics enables data to be simultaneously partitioned on four different dimensions at the same storage cost, enabling a larger fraction of queries to benefit from partition filtering and joins without network shuffle.

To achieve this, INSTalytics uses compute-awareness to customize the 3-way replication that the cluster file system employs for availability. A new heterogeneous replication layout enables INSTalytics to preserve the same recovery cost and availability as traditional replication. INSTalytics also uses compute-awareness to expose a new sliced-read *API that improves performance of joins by enabling multiple compute nodes to read slices of a data block efficiently via co-ordinated request scheduling and selective caching at the storage nodes.*

We have built a prototype implementation of INSTalytics in a production analytics stack, and show that recovery performance and availability is similar to physical replication, while providing significant improvements in query performance, suggesting a new approach to designing cloud-scale big-data analytics systems.

1 Introduction

All the powers in the universe are already ours. It is we who put our hands before our eyes and cry that it is dark.

- Swami Vivekananda

Large-scale cluster file systems [10, 22, 17] are designed to deal with server and disk failures as a common case. To ensure high availability despite failures in the data center, they employ *redundancy* to recover data of a failed node from data in other available nodes [11]. One common redundancy mechanism that cluster file systems use for compute-intensive workloads is to keep multiple (typically three) copies of the data on different servers. While redundancy improves availability, it comes with significant storage and write amplification overheads, typically viewed as the *cost* to be paid for availability.

An increasingly important workload in such large-scale cluster file systems is big data analytics processing [6, 31, 2]. Unlike a transaction processing workload, analytics queries are typically scan-intensive, as they are interested in millions or even billions of records. A popular technique employed by analytics systems for efficient query execution, is partitioning of data [31] where the input files are *sorted* or *partitioned* on a particular column, such that records with a specific range of column values are physically clustered within the file. With partitioned layout, a query that is only interested in a particular range of column values (say 1%) can use metadata to only scan the relevant partitions of the file, instead of scanning the entire file (potentially tens or hundreds of terabytes). Similarly, with partitioned layout, join queries can avoid the cost of expensive network shuffle [29].

However, as partitioning is tied to the physical layout of bytes within a file, data is partitioned only on a single dimension; as a result, it only benefits queries that perform a filter or join on the column of partitioning, while other queries are still forced to incur the full cost of a scan or network shuffle.

In this paper, we present *INSTalytics* (INtelligent STore powered Analytics), a system that drives significant efficiency improvements in performing large-scale big data analytics, by amplifying the well-known benefits of partitioning. In particular, *INSTalytics* allows data to be partitioned simultaneously along *four* different dimensions, instead of a single dimension today, thus allowing a large fraction of queries to achieve the benefit of efficient partition filtering and efficient joins without network shuffle. The key approach that enables such improvements in *INSTalytics* is making the distributed file system *compute-aware*; by customizing the 3-way replication that the file system already does for availability, *INSTalytics* achieves such heterogeneous partitioning without incurring additional storage or write amplification cost.

The obvious challenge with such *logical replication* is ensuring the same availability and recovery performance as physical replication; a naive layout would require scanning the entire file in the other partitioning order, to recover a single failed block. *INSTalytics* uses a novel layout technique based on

*Jayashree and Piyus worked on this while at Microsoft Research

super-extents and *intra-extent circular buckets* to achieve recovery that is as efficient as physical replication. It also ensures the same availability and fault isolation guarantees as physical replication under realistic failure scenarios. The layout techniques in *INSTalytics* also enable an additional fourth partitioning dimension, in addition to the three logical copies.

The file system in *INSTalytics* also enables efficient execution of join queries, by supporting a new *sliced-read* API that uses compute-awareness to co-ordinate scheduling of requests across multiple compute nodes accessing the same storage extent, and selectively caches only the slices of the extent that are expected to be accessed, instead of caching the entire extent.

We have implemented *INSTalytics* in a production distributed file system stack, and evaluate it on a cluster of 500 machines with suitable modifications to the compute layers. We demonstrate that the cost of maintaining multiple partitioning dimensions at the storage nodes is negligible in terms of recovery performance and availability, while significantly benefiting query performance. We show through microbenchmarks and real-world queries from a production workload that *INSTalytics* enables significant improvements upto an order of magnitude, in the efficiency of analytics processing.

The key contributions of the paper are as follows.

- We propose and evaluate novel layout techniques for enabling four simultaneous partitioning/sorting dimensions of the same file without additional cost, while preserving the availability and recovery properties of the present 3-way storage replication;
- We characterize a real-world analytics workload in a production cluster to evaluate the benefit of having multiple partitioning strategies;
- We demonstrate with a prototype implementation that storage co-design with compute can be implemented practically in a real distributed file system with minimal changes to the stack, illustrating the pragmatism of the approach for a data center;
- We show that compute-awareness with heterogenous layout and co-ordinated scheduling at the file system significantly improves the performance of filter and join queries.

The rest of the paper is structured as follows. In § 2, we provide a background of big data analytics processing. In § 3, we characterizate a production analytics workload. We present logical replication in § 4, discuss its availability implications in § 5, and describe optimizations for joins in § 6. We present the implementation of *INSTalytics* in § 7, and evaluate it in § 8. We present related work in § 9, and conclude in § 10.

2 Background

In this section, we describe the general architecture of analytics frameworks, and the costs of big data query processing.

Cluster architecture: Big data analytics infrastructure typically comprises of a compute layer such as MapReduce [6] or Spark [28], and a distributed file system such as GFS [10] or HDFS [22]. Both these components run on several thousands of machines, and are designed to tolerate machine failures given the large scale. The distributed file system is typically a decentralized architecture [14, 26, 10], where a centralized "master" manages metadata while thousands of storage nodes manage the actual blocks of data, also referred to as chunks or extents (we use the term "extent' in the rest of the paper). An extent is typically between 64MB and 256MB in size. For availability under machine failures, each storage extent is replicated multiple times (typically thrice). The compute layer can run either on the same machines as the storage nodes (*i.e.*, co-located), or a different set of machines (*i.e.*, disaggregated). In the co-located model, the compute layer has the option of scheduling computation for locality between the data and compute, say at a rack-level, for better aggregate data bandwidth.

Cost of analytics queries: Analytics queries often process millions or billions of records, as they perform aggregation or filtering on hundreds of terabytes. As a result, they are scan-intensive on the disks - using index lookups would result in random disk reads on millions of records. Hence disk I/O is a key cost of query processing given the large data sizes.

An important ingredient of most big data workloads is *joins* of multiple files on a common field. To perform a join, all records that have the same join key value from both files need to be brought to one machine, thus requiring a *shuffle* of data across thousands of servers; each server sends a *partition* of the key space to a designated worker responsible for that partition. Such all-to-all *network shuffle* typically involves an additional disk write of the shuffled data to an intermediate file and subsequent disk read. Further, it places load on the data center switch hierarchy across multiple racks.

Optimizations: There are two common optimizations that reduce the cost of analytics processing: partitioning, and co-location. With partitioning, the data file stored on disk is sorted or partitioned by a particular dimension or column. If a query filters on that same column, it can avoid the cost of a full scan by performing *partition elimination*. With co-location, joins can execute without incurring network shuffle. If two files A and B are likely to be joined, they can both be partitioned on the join column/dimension in a consistent manner, so that their partition boundaries align. In addition, if the respective partitions are also placed in a rack-affinitized manner, the join can avoid cross-rack network shuffle, as it would be a per-partition join. Further, the partitioned join also avoids the intermediate I/O, because the respective partitions can perform the join of small buckets in memory.

3 Workload Analysis

We analyzed one week's worth of queries on a production cluster at *Microsoft* consisting of tens of thousands of servers. We

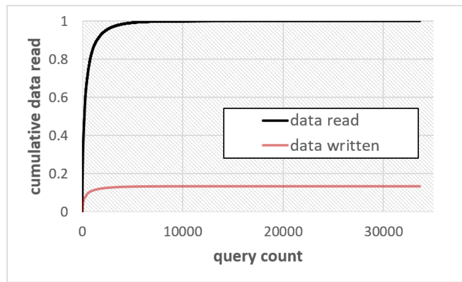


Figure 1: Data filtering in production queries

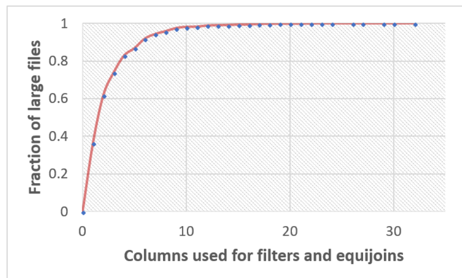


Figure 2: Number of partition dimensions needed.

share below our key findings.

3.1 Query Characteristics

Data filtering: The amount of data read in the first stage of the query execution vis-a-vis the amount of data written back at the end of the first stage indicates the degree of filtering that happens on the input. In Figure 1, we show a CDF of the data read and data written by unique query scripts (aggregating across repetitions) along the X-axis, sorted by the most expensive jobs. As the graph shows, on average, there is a 7x reduction in data sizes past the first stage. This reduction is due to both column-level and row-level filtering; while column-stores [15] help with the former, we primarily focus on row-level filtering which is complementary.

Importance of join queries: Besides filtering, joins of multiple files are an important part of big data analytics. In our production workload, we find that 37% of all queries contain a join, and 85% of those perform a join at the start of the query.

3.2 Number of dimensions needed

Today's systems limit the benefit of partitioning to just one dimension. To understand what fraction of queries would benefit from a higher number of dimensions, we analyzed each input file referenced in any job during a week, and we extracted all columns/dimensions that were ever used in a filter or join clause in any query that accessed the file. We plot a CDF of the fraction of files that were only accessed on K columns (K varying along X axis). As Figure 2 shows, with one partitioning dimension, we cover only about 33% of files, which illustrates the limited utility of today's partitioning. However, with 4 partitioning dimensions, the coverage grows significantly to about 83%. Thus, for most files, having them partitioned in 4 dimensions would enable efficient execution for all queries on that file, as they would benefit from partitioning or co-location.

Supporting multiple dimensions: Today, the user can partition data across multiple dimensions by storing multiple copies. However this comes with a storage cost: to support 4 dimensions, the user incurs a 4x space overhead, and worse, a 4x cost in write bandwidth to keep the copies up to date. Interestingly, in our discussions with teams that perform big-data analytics within *Microsoft*, we found examples where large product groups actually maintain multiple (usually 2) copies (partitioned on different columns) just to reduce query latency, despite the excessive cost of storing multiple copies. Many teams stated that more partition dimensions would enable new kinds of analytics, but the cost of supporting more dimensions today is too high.

4 Logical Replication

The key functionality in *INSTalytics* is to enable a much larger fraction of analytics queries to benefit from partitioning and co-location, but without paying additional storage or write cost. It achieves this by co-designing the storage layer with the analytics engine, and changing the *physical* replication (usually three copies that are byte-wise identical) that distributed file systems employ for availability, into *logical* replication, where each copy is partitioned along a different column. Logical replication provides the benefit of three simultaneous partitioning columns at the same storage and write cost, thus improving query coverage significantly, as shown in Section 3. As we describe later, our layout actually enables *four* different partitioning columns at the same cost.

The principle of logical replication is straight-forward, but the challenge lies in the details of the layout. There are two conflicting requirements: first, the layout should help query performance by enabling partition filtering and collocated joins in multiple dimensions; second, recovery performance and availability should not be affected.

In the rest of the section we describe several variants of logical replication, building towards a more complete solution that ensures good query performance at a recovery cost and availability similar to physical replication. For each variant, we describe its recovery cost and potential query benefits. We use the term *dimension* to refer to a column used for partitioning; each logical replica would pertain to a different dimension. We use the term *intra-extent bucketing* to refer to partitioning of rows within an extent; and we use the term *extent-aligned partitioning* to refer to partitioning of rows across multiple extents where partition boundaries are aligned with extent boundaries. When the context is clear, we simply use the terms *bucketing* and *partitioning* respectively for the above.

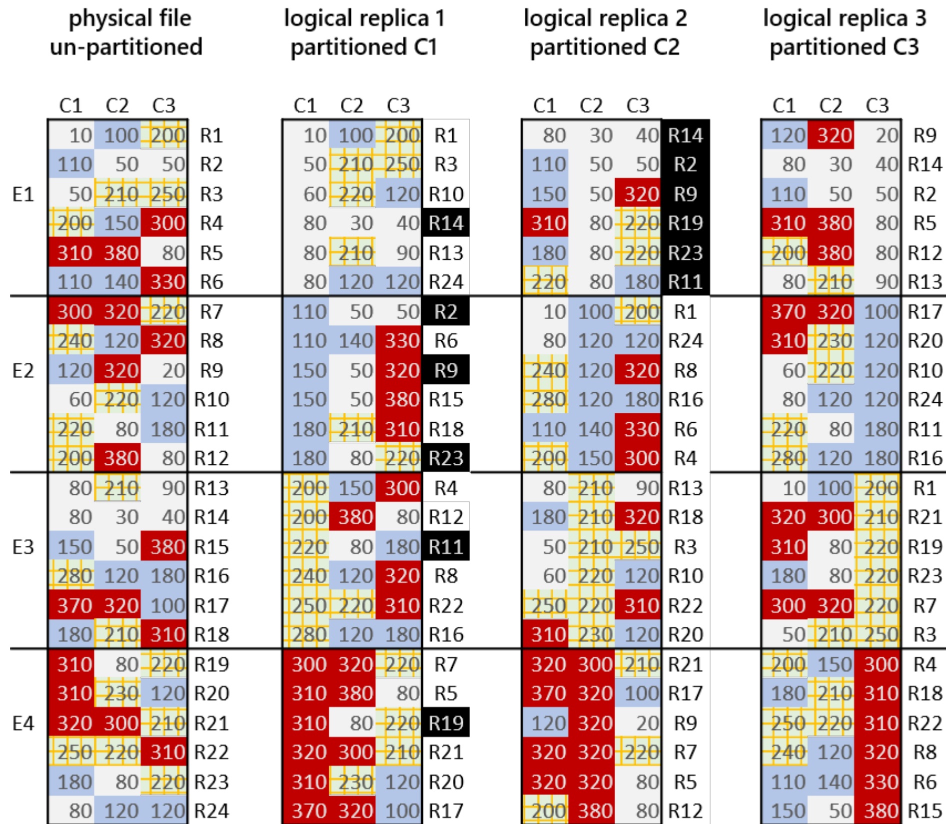


Figure 3: **Naive logical replication.** The figure shows the physical layout of the input file and the layout with naive logical replication. The file has 4 extents each consisting of 6 rows, 3 columns per row. Each logical replica is range partitioned on a different column. The first replica is partitioned on the first column (*e.g.*, E1 has values from 0-99, E2 from 100-199 and so on). Recovering E1 from replica 2 requires reading all extents from replica 1.

4.1 Naive layouts for logical replication

There are two simple layouts for logical replication, neither meeting the above constraints. The first approach is to perform logical replication at a file-granularity. In this layout the three copies of the file are each partitioned by a different dimension, and stored seperately in the file system with replication turned off. This layout is ideal for query performance as it is identical to keeping three different copies of the file. Unfortunately this layout is a non-starter in terms of recovery; as Figure 3 shows, there is *inter-dimensional diffusion* of information; the records in a particular storage extent in one dimension will be diffused across nearly all extents of the file in the other dimension. Thus, recovering a 200MB extent would require reading an entire say 10TB file in another dimension, whereas with physical replication, only 200MB is read from another replica.

The second sub-optimal approach is to perform logical replication at an intra-extent level. Here, one would simply use *intra-extent bucketing* to partition the records *within* a storage extent along different dimensions in each replica. This simplifies recovery as there is a one-to-one correspondence with the other replicas of the extent. This approach helps partly with filter queries as the file system can use metadata to read only the relevant buckets within an extent, but is not as efficient as the previous layout as clients would touch all extents instead of a subset of extents. The bigger problem though is that joins or aggregation queries cannot benefit at all, because co-location/local shuffle is impossible. We discuss more about the shortcomings of this approach in handling joins in Section 6.

4.2 Super-Extents

INSTalytics bridges the conflicting requirements of query performance and recovery cost, by introducing the notion of a *super-extent*. A super-extent is a fixed number (typically 100) of contiguous extents in the file in the original order the file was written (see Figure 4). Partitioning of records happens *only* within the confines of a super-extent and happens in an extent aligned manner. As shown in the figure this ensures that the inter-dimensional diffusion is now limited to only within a super-extent; all records in an extent in one dimension are hence guaranteed to be present somewhere within the corre-

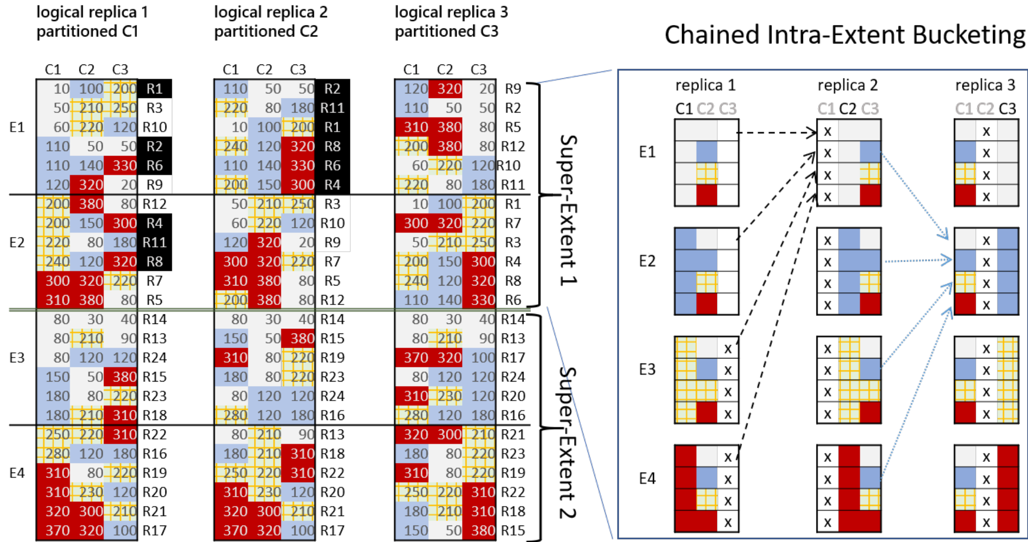


Figure 4: **Super-extents and intra extent bucketing.** The file to be replicated is divided into super extents and logical replication is performed within super-extents. Any extent (like E1 of replica 2 as highlighted) can be recovered by only reading extents of the same super extent in another replica (E1 and E2 from replica 1). Figure on the right shows a more detailed view of a super-extent (with 4 extents per super-extent instead of 2 for clarity). Data within an extent is partitioned on a different column and this helps reduce recovery cost further as recovering an extent only involves reading one bucket from each extent of the other replica. Recovering E4 of replica 2 requires only reading slices from replica 1 whose C2 boxes are red.

sponding super-extent (*i.e.*, 100 extents) of the other partitioning dimension. All 3 copies of the super-extent have exactly the same information, just arranged differently. The number of extents within a super-extent is configurable; in that sense, the super-extent layout can be treated as striking a tunable balance between the two extremes *i.e.*, global partitioning and intra-extent partitioning.

Super-extents reduce recovery cost because in order to recover a 200MB extent, the store has to read "only" 100 extents of that super-extent from another dimension. This is better than reading the entire say 10TB file with the naive apporoach, but the cost of recovery is still significantly higher (100x) than physical replication. We improve on this below. From a query perspective, the super-extent based layout limits the granularity of partitioning to the number of extents per super extent. This limits the potential savings for filtering and co-location. For example, consider a very selective filter query that matches only 1/1000th of the records. With the super extent layout, the query would still have to read one extent in each super-extent. Hence the maximum speedup because of partition elimination (with 100 extents per super-extent) is 100x, whereas the file-level global partitioning could support 1000 partitions and provide a larger benefit. However, the 100x speed up is significant enough that the tradeoff is a net win. Conceptually, the super extent based layout does a partial partitioning of the file, as every key in the partition dimension could be present in multiple extents, one per super extent. The globally partitioned layout would have keys more densely packed within fewer extents.

Fourth partition dimension. Finally, super-extents benefit query execution in a way that file-level global partitioning does not. As we do not alter the native ordering of file extents across super-extent boundaries, we get a fourth dimension. If the user already partitioned the file on a particular column, say timestamp, we preserve the coarse partitions by timestamp across super-extents, so a query filtering on timestamp can eliminate entire super-extents. Thus, we get 4 partition dimensions for no additional storage cost compared to today[1].

4.3 Intra-extent Chained Buckets

While super-extents reduce recovery cost to 100 extents instead of the whole file, the 100x cost is still a non-starter. Recovery needs to be low-latency to avoid losing multiple copies, and needs to be low on resource usage so that the cluster can manage recovery load during massive failures such as rack failures. Intra-extent chained buckets is the mechanism we use to make logical recovery as efficient as physical recovery.

The key idea behind intra-extent chained buckets, is to use bucketing *within* an extent for recovery instead of query performance. The records within an extent are bucketed on a dimension that is different from the partition dimension used within the super-extent. Let us assume that C_1, C_2, and C_3 are the three columns/dimensions chosen for logical replica-

[1] This fourth dimension is not equally powerful as the first three because while it provides partition elimintation for filters, it does not provide colocation for joins

tion. Given 100 extents per super-extent, the key-space of C_1 would be partitioned into 100 ranges, so the ith extent in every super-extent of dimension C_1 would contain records whose value for column C_1 fall in the ith range.

Figure 4 (right) illustrates how intra-extent chained buckets work. Let us focus on the first extent E1 of a super-extent in dimension C_1. The records within that extent are further bucketed by dimension C_2. So the 200MB extent is now comprised of 100 buckets each roughly of size 2MB. The first intra-extent bucket of E1 contains only records whose value of column C_2 falls in the first partition of dimension C_2, and so on. Similarly the extents of dimension C_2 have an intra-extent bucketing by column C_3, and the extents of dimension C_3 are intra-extent bucketed by column C_1.

With the above layout, recovery of an extent does not require reading the entire super-extent from another dimension. In Figure 4, to recover the last extent of replica 2, the store needs to read only the last bucket from extents E1 to E4 of replica 1, instead of reading the full content of those 4 extents.

Thus, in a superextent of 100 extents, instead of reading 100 x 200MB to recover a 200MB extent, we now only read 2MB each from 100 other extents, *i.e.*, read 200MB to recover a 200MB extent, essentially the same cost as physical replication. By reading 100 chunks of size 2MB each, we potentially increase the number of disk seeks in the cluster. However, given the 2MB size, the seek cost gets amortized with transfer time, so the cost is similar especially since physical recovery also does the read in chunks. As we show in Section 8, the aggregate disk load is very similar to physical replication. From a networking perspective, the bandwidth usage is similar except we have a parallel and more distributed load on the network.

Thus, with super-extents and intra-extent chained buckets, we achieve our twin goals of getting the benefit of partitioning and co-location for more queries, while simultaneously keeping recovery cost the same as physical replication.

4.4 Making storage record-aware

In order to perform logical replication, the file system needs to rearrange records across extents. However, the interface to the file system is only in terms of opaque blocks. Clients perform a *read block* or *write block* on the store, and the internal layout of the block is only known to the client. For example, the file could be an unstructured log file or a structured file with internal metadata. One could bridge this semantic gap by changing the storage API to be record-level, but it is impractical as it invovles changes to the entire software stack , and curtails the freedom of higher layers to use diverse formats.

To bridge this tension, we introduce the notion of *format adapters* in *INSTalytics*. The adapter is simply an encoder and decoder that translates back and forth between an opaque extent, and records within that extent. Each format would have its own adapter registered with the store, and only registered formats are supported for logical replication. This is pragmatic in cloud-scale data centers where the same entity controls both the compute stack and the storage stack and hence there is coordination when formats evolve.

A key challenge with the adapter framework is dealing with formats that disperse metadata. For example, in one of our widely-used internal formats, there are multiple levels of pointers across the entire file.there is a *footer* at the end of the file that points to multiple chunks of data, which in turn point to pages. For a storage node that needs to decode a single extent for logical replication, interpreting that extent requires information from several other extents (likely on other storage nodes), making the adapter inefficient. We therefore require that each extent is self-describing in terms of metadata. For the above format, we made small changes to the writer to terminate chunks at extent boundaries and duplicate the footer information within a chunk. Given the large size of the extent, the additional metadata cost is negligible (less than 0.1%).

4.5 Creating logical replicas

Logical replication requires application hints on the dimensions to use for logical replication, the file format, *etc.*. Also, not all files benefit from logical replication, as it is a function of the query workload, and the read/write ratio. Hence, files start off being physically replicated, and an explicit API from the compute layer converts the file to being logically replicated. Logical replication happens at the granularity of super-extents; the file system picks 100 extents, shuffles the data within those 100 (3-way replicated) extents and writes out 300 new extents, 100 in each dimension. The work done during logical replication is a disk read and a disk write. Failure-handling during logical replication is straight-forward: reads use the 3-way replicated copies for failover, and writes failover to a different available storage node that meets the fault-isolation constraints. The logical replication is not *in-place*. Until logical replication for a super-extent completes, the physically replicated copies are available, which simplifies failure retry. As the application that generates data knows whether to logically replicate, it could place those files on SSD, so that the extra write and read are much cheaper; because it's a transient state until logical replication, the SSD space required is quite small.

4.6 Handling Data Skew

To benefit from partitioning, the different partitions along a given dimension must be roughly balanced. However, in practice, because of data skew along some dimensions [4], some partitions may have more data than others. To handle this, *INSTalytics* allows for variable sized extents within a super-extent, so that data skew is only a performance issue, not a correctness issue. Given the broader implications of data skew for query performance, users already pre-process the data to

ensure that the partitioning dimensions are roughly balanced (by using techniques such as prefixing popular keys with random salt values, calibrating range boundaries based on a distribution analysis on the data, *etc.*). As the user specifies the dimensions for logical replication, as well as the range boundaries, *INSTalytics* benefits from such techniques as well. In future, we would like to build custom support for skew handling within *INSTalytics* as a more generic fallback, by performing the distribution analysis as part of creating logical replicas, to re-calibrate partition boundaries when the user-data is skewed.

5 Availability with logical replication

While the super-extent based layout ensures the same recovery cost as physical replication, it incurs a hit in availability. With physical replication, for an extent to become unavailable, all three machines holding the three copies of the extent must be unavailable. If p is the independent probability that a specific machine in the cluster goes down within say a 5-minute window, the probability of unavailability in physical replication for a given extent is p^3. But with logical replication, an extent becomes unavailable if the machine with that extent is down, and additionally *any one* of the 100 machines in each of the other two dimensions containing replicas of the super-extent, are down, making the probability of unavailability $p \times 100p \times 100p = 10^4.p^3$. In this reasoning, we only consider independent machine failures, as correlated failures are handled below with fault-isolated placement.

5.1 Parity extents

To handle this gap in availability, we introduce an additional level of redudundancy in the layout. Within a super-extent replica which comprises of 100 extents, we add *parity extents* with simple XOR parity for every group of 10 extents, *i.e.*, a total of 10 parity extents per super-extent replica. Now, each parity group can tolerate one failure, which means for unavailability, there has to be a double failure in the extent's parity group, and in addition, at least one parity group in each of the other two dimensions must have a double failure. The probability thus becomes $p.10p \times 10.\binom{11}{2}.p^2 \times 10.\binom{11}{2}.p^2 = 3.02 \times 10^6.p^6$. Solving this for p, as long as $(p < 0.7\%)$, the availability would be better than physical replication.[2]. . In rare cases of clusters where the probability is higher, we could use double-parity [5] in a larger group of 20 extents, so that each group of 20 extents can tolerate two failures. The probability of unavailability now becomes $p.\binom{21}{2}.p^2 \times 5.\binom{22}{3}.p^3 \times 5.\binom{22}{3}.p^3 = 12.45 \times 10^9.p^9$, so the cut-off point becomes $p < 2.1\%$. Note that another knob to control availability is the size of a super-extent: with super-extents comprising 10 extents instead of 100, the single parity itself can handle a significant failure probability of $p < 3.2\%$.

The machine failure probability p above refers to the failure probability within a small time window, *i.e.*, the time it takes to recover a failed extent. This is much lower than the average % of machines that are offline in a cluster at any given time, because the latter includes long dead machines, whose data would have been recovered on other machines anyway. As we show in Section 8, this failure probability of random independent machines (excluding correlated failures) in large clusters is less than 0.2%, so single parity is often sufficient, and hence this is what we have currently implemented.

Recovering a parity extent requires 10x disk and network I/O compared to a regular extent, because it has to perform an XOR of 10 corresponding blocks. As 10% of logically replicated extents are parity extents, this would double the cost of recovery (10% x 10x). We therefore store two physically replicated copies of each parity block, so that during recovery, most of the failed parity blocks can be recovered with a raw copy, and we incur the 10x disk cost only for a tiny fraction. This is a knob for the cluster administrator - whether to incur the additional 10% space cost, or the 2x performance cost during recovery; in our experience, recovery cost is more critical and hence the default is 2-way replication of parity blocks.

5.2 Fault-isolated placement

Replication is aimed at ensuring availability during machine failures. As failures can be correlated, *e.g.*, a rack power switch can take down all machines in that rack, file systems perform fault-isolated placement. For example, the placement would ensure that the 3 replicas of an extent are placed in 3 different failure domains, to aovid simultaneously losing multiple copies of the block. With logical replication, each extent does not have a corresponding replica, thus requiring a different strategy for fault-isolation. The way *INSTalytics* performs this fault isolation is to reason at the super-extent level, because all replicas of a given super-extent contain exactly the same information. We thus place each replica of the super-extent in a disjoint set of failure domains relative to any other replica of the same super-extent, thus ensuring the same fault-isolation properties as physical replication.

6 Efficient Processing of Join Queries

The multi-dimensional partitioning in *INSTalytics* is designed to improve performance of join queries in addition to filter queries. In this section, we first describe the *localized shuffle* that is enabled when files are joined on one of the dimensions of logical replication. We then introduce a new compute-aware API that the file system provides, to further optimize joins by

[2] This formula is an approximation (for ease of understanding) that works for small values of p. The accurate (and more complex) formula is $p.(1-(1-p)^{10})(1-(1-p)^{100}(1+10p)^{10})^2$ which would be greater than p^3 (physical replication) as long as $p < 0.725\%$

completely eliminating network shuffle.

6.1 Localized Shuffle

Joins on a logically replicated dimension can perform *localized shuffle*: partition *i* of the first file only needs to be joined with partition *i* of the second file, instead of a global shuffle across all partitions. Localized shuffle has two benefits. First, it significantly lowers (by 100x) the fan-in of the "reduce" phase, eliminating additional intermediate "aggregation" stages that big-data processing systems introduce just to reduce the fan-in factor for avoiding small disk I/Os and for fault-tolerance [29]. Elimination of intermediate aggregation reduces the number of passes of writes and reads of the data from disk. Second, it enables network-affinitized shuffle. If all extents for a partition are placed within a single rack of machines, local shuffle avoids the shared aggregate switches in the data center, and can thus be significantly more efficient.

The placement by the file system ensures that extents pertaining to the same partition across all super-extents in a given dimension are placed within a single rack of machines. The file system supports an interface to specify during logical replication the logical replica of another file to co-affinitize with.

6.2 Sliced Reads

Localized shuffle avoids additional aggregation phases, but still requires one write and read of intermediate data. If the individual partitions were small enough, the join of each individual partition can happen in parallel in memory, avoiding this disk cost. However, as super-extents limit the number of partitions to 100, joins of large files (*e.g.*, 10 TB) will be limited by parallelism and memory needs at compute nodes.

To address this limitation, *INSTalytics* introduces a new file system API called a *sliced-read*, which allows a client to read a small semantic-slice of an extent that belongs to a sub-range of a partition, further sub-dividing the partition into 100 (*slice-factor*) buckets. For instance, if say the first (out of 100) partition represents the range 0-10k, a *sliced-read* can ask for only records in the range 9k-9.1k, thus providing the equivalent benefit of having 10,000 partitions on the original file while the super-extent remains at 100 partitions. Each compute node would now read one bucket from each super-extent. The ability to perform per-partition joins enables efficient sliced reads, as it allows join execution to happen in *stages*, few partitions at a time (*e.g.*, 1-10 out of 100). We have modified the job scheduler to schedule compute nodes in stages.

However, with intra-extent bucketing (§ 4.3), the bucketing within an extent is by a different dimension, whereas *sliced-read* reqiures the bucketing within an extent to be on the same dimension. Hence, in order to return slices, the storage node must locally re-order the records within the extent. As multiple compute nodes will read different slices of the same extent, a naive implementation that reads the entire extent, repartitions it and returns only the relevant slice, would result in excessive disk I/O (*e.g.*, 100x more disk reads for a slice-factor of 100). In-memory caching of the re-ordered extent data at the storage nodes can help, but incurs a memory cost proportional to the working set (the number of extents being actively processed).

To bridge this gap, the storage node performs co-ordinated lazy request scheduling, as it is aware of the pattern of requests during a join through a *sliced-read*. In particular, it knows that *slice-factor* compute nodes would be reading from the same extent, so it queues requests until a threshold number of requests (*e.g.*, 90%) for a particular extent arrives. It then reads the extent from disk, re-arranges the records by the right dimension and services the requests, and caches the chunks pertaining to the stragglers, *i.e.*, the remaining 10%. The cache usage reduces further by a factor of 10-100 with staging (above), *i.e.*, to less than 0.1%-1% of the input size. Thus, by exploiting compute-awareness to perform co-ordinated request scheduling and selective caching, *sliced-read* enables join execution without incurring any intermediate write to disk.

Discussion: Both localized shuffle and sliced reads for efficient joins require the cross-extent partitioning that super-extents provide, and do not work with a naive approach of simply bucketing within an extent, as all extents will have data from all partitions in that model. The small fan-in that super-extent partitioning enables, is crucial to the feasibility of co-ordinated scheduling.

7 Implementation

We have implemented *INSTalytics* in the codebase of a production analytics stack that handles exabytes of data. A key constraint is that to be practically deployable, the changes needed to be *surgical* and isolated without changing existing reasoning about recovery and failures. We describe in this section key aspects of the implementation.

7.1 System architecture

The file system in our analytics stack comprises of a Paxos-replicated *Master* that holds the metadata *in memory*, and Storage Nodes that store extents identified by *GUIDs*. The master maintains a mapping from a *file ID* to a list of extents in the file, and an offset-to-extent map. The master also tracks *extent metadata*, tracking their size, the list of replicas, *i.e.*, the storage nodes on which those instances are placed, *etc.*. A key constraint is to avoid increase in the in-memory metadata.

7.2 Changes to Filesystem Master

With logical replication, the extent is no longer a homogenous entity. For example, the sizes of the replicas in each dimension may be different due to skews in data distribution. To handle this, we hide the actual sizes of the extent instances from the master; the master continues to track extents and offsets in the physical space. The extent table at the master continues to track three instances per extent; we map them

to the same bucket index across the three logical dimensions (*e.g.*, instances of extent 0 of the stream would track the 0th bucket of the three dimensions for first super-extent). We add a super-extent table to the stream metadata, tracking extent indexes that delimit super-extents; the total increase in metadata is $< 1\%$.

The extent placement logic is changed to handle super-extent aware fault isolation, and the recovery path is also changed minimally. Instead of asking a new storage node to copy extent data from a single source, the master sends a *list* of 300 sources for entire super-extent; the storage node talks to them to perform recovery.

7.3 Changes to Storage Nodes

The storage nodes handle most of the work for creation and recovery of logical extents. For each super-extent, the master sends a `logically_replicate` request to an orchestrator storage node, specifying the 300 physically replicated source instances, and 300 new destination storage nodes. The orchestrator sends a `create_logical_extent` request to each destination, on receipt of which the destination sends read requests to 100 source nodes to read one bucket's worth of data from each. It then assembles all that data into one extent by invoking the encoder of the format-specific adapter. The storage nodes that receive the bucketed read request invoke the *decoder* of the format-specific adapter to convert the extent into records, apply bucketing on the column values to return just the bucket the destination node is interested in; a decoded cache helps reuse this work across destinations. To amortize this cost across queries, logical replication is performed only on files that have a high read-write ratio. The storage nodes also track logical-extent-specific metadata in SSD that contain pointers to intra-extent sub-buckets, and additional rowID information to reconstruct data exactly as they were originally written. The recovery flow works similar to creating a logical extent, and handles parity recovery as well.

7.4 Changes to Compute Layers

Because of the adapter-based design at the storage nodes and the store's ability to reconstruct data with byte-level fidelity, compute layers continue to access storage through a block interface. The changes mostly have to deal with how the multiple dimensions are exposed to layers such as the query optimizer; the QO treats them as multiple clustered indexes. With our changes, the query optimizer can handle simple filters and joins by automatically picking the correct partition dimension; full integration into the query optimizer to handle complex queries is beyond the scope of this paper. The store provides a `filtered_read` API which returns only the subset of extents that match a given filter condition. Also, the store provides direct access to a specific dimension or specific bucket with file-name mangling (*e.g.*, filename[0]), for ease of integration. The client library of the store initiates *recovery-on-demand*; instead of trying a different replica for failover, it triggers an online recovery. As described in Section 4, this cost is identical to reading from another physical replica.

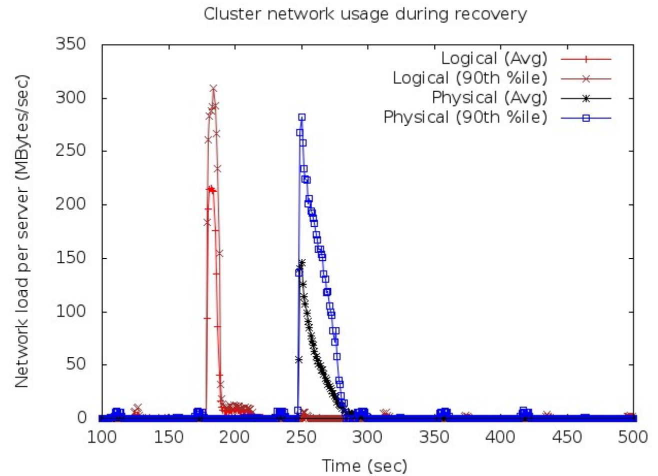


Figure 5: Cluster network load during recovery

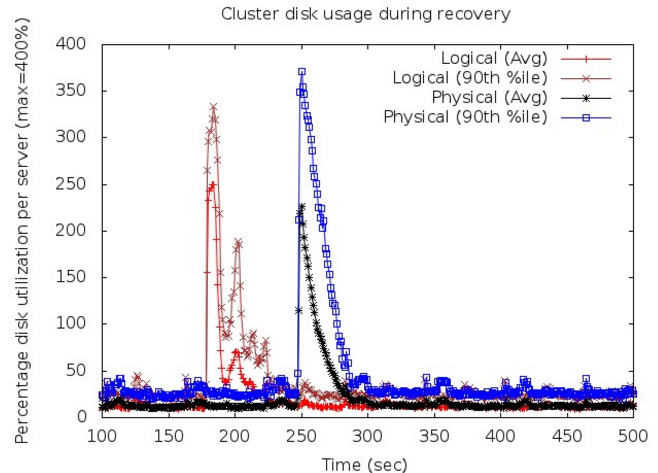


Figure 6: Cluster disk load during recovery

8 Evaluation

We evaluate *INSTalytics* in a cluster of 500 servers (20 racks of 25 servers each). Each server is a 2.4 GHz Xeon with 24 cores and 128 GB of RAM, 4 HDDs and 4 SSDs. 450 out of the 500 servers are configured as storage (and compute) nodes, and 5 as store master. We answer three questions in the evaluation:

- What is the recovery cost of the *INSTalytics* layout?
- What are the availability characteristics of *INSTalytics*?
- How much do benchmarks and real queries benefit ?

For our evaluation, unless otherwise stated, we use a configuration with 100 extents per superextent with an average extent size of 256MB.

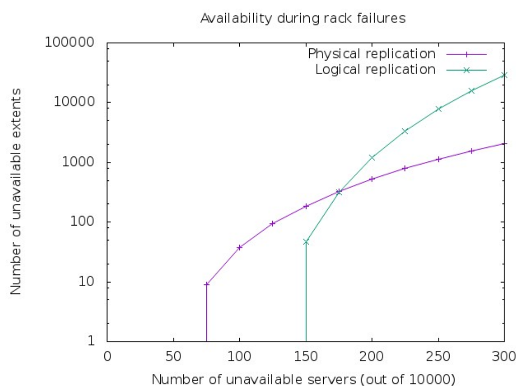


Figure 7: Storage availability during rack failures

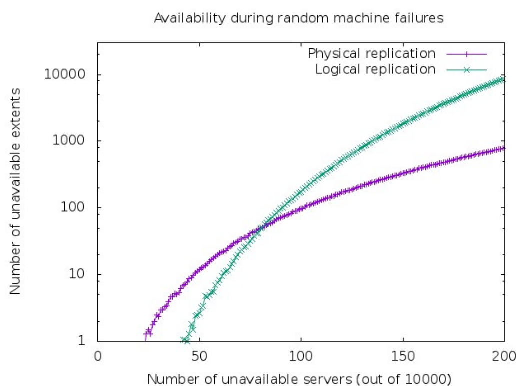


Figure 8: Storage availability during machine failures

8.1 Recovery performance

For this experiment, we take down an entire (random) rack of machines out of the 20 racks, thus triggering recovery of the extent replicas on those 25 machines. We then measure the load caused by recovery on the rest of the machines in the cluster. For a fair comparison, we turn off all throttling of recovery traffic in both physical and logical replication. During the physical/logical replication experiment, we ensure that all extents recovered belong to physically/logically replicated files respectively. The number of extents recovered is similar in the two experiments, about 7500 extents ($\tilde{1}$.5TB). We measure the network and disk utilization of all the live machines in the cluster, and plot average and 90th percentiles. Although the physical and logical recovery are separate experiments, we overlay them in the same graph with offset timelines.

Figure 5 shows the outbound network traffic on all servers in the cluster during recovery. The logical recovery is more bursty because of its ability to read sub-buckets from 100 other extents. The more interesting graph is Figure 6 that shows the disk utilization of all other servers; because each server has 4 disks, the maximum disk utilization is 400%. As can be seen, the width of the two spikes are similar, which shows that recovery in both physical and logical complete with similar latency. The metric of disk utilization, together with the overall time for recovery, captures the actual work done by the disks; for instance, any variance in disk queue lengths caused by extra load on the disks due to logical recovery, would have manifested in a higher width of the spike in the graph. The spike in the physical experiment is slightly higher in the 90th percentile, likely because it copies the entire extent from one source while logical replication is able to even out the load across several source extents. The key takeaway from the disk utilization graph is that the disk load caused by reading intra-extent chained buckets from 100 storage nodes, is as efficient as copying the entire extent from a single node with physical replication. The logical graph has a second smaller spike in utilization corresponding to the lag between reads and writes (all sub-buckets need to be read and assembled before the write can happen). For both disk and network, we summed up the aggregate across all servers during recovery and they are within 10% of physical recovery.

8.2 Availability

In this section, we compare the availability of logical and physical replication under two failure scenarios: isolated machine failures and co-ordinated rack-level failures. Because the size of our test cluster is too small to run these tests, we ran a simulation of a 10K machine cluster with our layout policy. Because of fault-isolated placement of buckets across dimensions, we tolerate up to 5 rack failures without losing any data (with parity, unavailability needs two failures in each dimension). Figure 7 shows the availability during pod failures. As can be seen, there is a crossover point until which logical replication with parity provides better availability than physical replication, and it gets worse after that. The cross-over point for isolated machine failures is shown in Figure 8 and occurs at 80 machines, *i.e.*, 0.8%. We also ran a simulation of double-parity layout; the cross-over points for isolated and correlated failures occur at 265 and 375 failures respectively.

To calibrate what realistic failure scenarios occur in practice, we analyzed the storage logs of a large cluster of tens of thousands of machines over a year to identify *dynamic failures*; failures which caused the master to trigger recovery (thus omitting long-dead machines etc.). We found that isolated machine failures are rare, typically affecting less than 0.2% of machines. There were 55 spikes of failures affecting more machines, but in all but 2 of those spikes, they were concentrated on one top-level failure domain, *i.e.*, a set of racks that share an aggregator switch. *INSTalytics* places the three dimensions of a file across multiple top-level failure domains, so is immune to these correlated failures, as it affects only one of three dimensions. The remaining 2 spikes had random failures of 0.8%, excluding failures from the dominant failure domain.

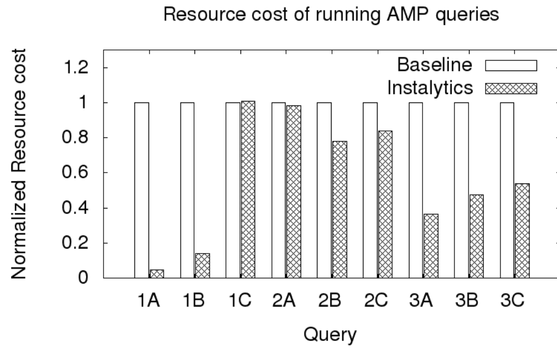


Figure 9: **Resource cost of AMP queries.**

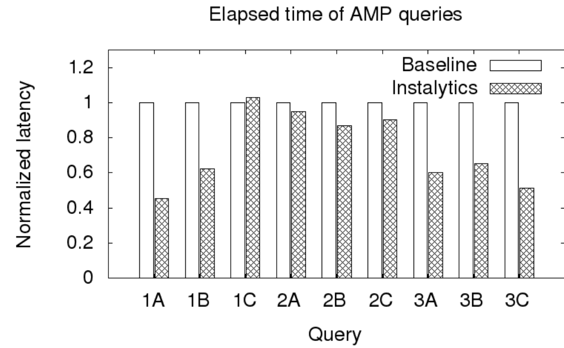


Figure 10: **Latency of AMP queries.**

8.3 Query performance

We evaluate the query performance benefits of *INSTalytics* on the AMP big-data benchmark [1, 16] and on a slice of queries from a production big data workload. We use two metrics: the query latency, and the "resource cost", *i.e.*, the number of machine hours spent on the query. All *sliced-read* joins use a request threshold of 90% and 10 stages. The baseline is a production analytics stack handling exabytes of data.

8.3.1 AMP Benchmark

The AMP benchmark has 4 query types: scan, aggregation, join, and external script query; we omit the last query, as our runtime does not have python support. As the emphasis is on large-scale workloads, we use a scale factor of 500 during data generation. We logically replicate the `uservisits` file (12TB) on 3 dimensons: url, visitDate, and IP and the `rankings` file (600GB) on 2: url and pagerank. Figures 9 and 10 show the benefit from *INSTalytics* in terms of cost and latency respectively. Queries 1A, 1B which perform heavy filtering on pagerank column of `rankings` benefit significantly from partitioning; the latency benefit is lower than the cost benefit because of the fixed startup latency. Query 3 does a join of the two files on the url column, after filtering `uservisits` on visitDate. Since both files are partitioned on url, 3C gets significant benefits while performing the join using sliced reads. 3A, 3B perform heavly filtering before the join and hence benefit from partitioning on the visitDate column. Queries 2A to 2C perform a group-by on a prefix of IP in `uservisits`, and get benefits of better local aggregation enabled by partitioning on IP. In summary, today one can pick one column out of three and get wins for a subset of queries, but lose benefits for other queries; *INSTalytics* simultaneously benefits all queries by supporting multiple dimensions.

Table 1 focuses on just the join within Q3 (excluding the aggregation that happens after the join). As can be seen, even the simple localized shuffle without sliced reads provides reasonable benefits. Further, we find that it reduces the amount of cross-rack network traffic by 93.4% compared to baseline. Sliced reads add to the benefit, providing nearly a 3x win for the join. To explore sensitivity to co-ordinated request scheduling, we show two configurations. In the "10% cache" configuration, the storage nodes wait for 90% of requests to arrive before serving the request; for the remaining 10% slices, the storage node caches the data. The "5%" configuration waits for 95% of requests. There is a tradeoff between cache usage and query performance; while the 5% configuration uses half the cache, it has a slightly higher query cost.

Configuration	Cost (hrs)	Latency (mins)
Baseline	125	11.8
Localized Shuffle	85	8.4
Sliced Reads (10% cache)	40.5	3.8
Sliced Reads (5% cache)	43	4.1

Table 1: **Performance of the join in Q3 of AMP benchmark**

8.3.2 Production queries

We also evaluate *INSTalytics* on a set of production queries from a slice of a telemetry analytics workload. The workload has 6 queries, all operating on an input file of size about 34 TB, sometimes joining with other files that are smaller. Table 2 shows the relative performance of *INSTalytics* on these queries. *INSTalytics* benefit queries Q1-Q4. Q5 and Q6 filter away very little data (<1%) and do not perform joins, so there are no gains to be had. Q1 and Q2 benefit from our join

Description	Q1	Q2	Q3	Q4	Q5	Q6
$Baseline_{cost}$	251	414	22	20	398	242
$INSTalytics_{cost}$	59	206	0.3	1.1	403	239
$Baseline_{latency}$	39	66	23	4	50	20
$INSTalytics_{latency}$	7	21	1.4	2.3	51	20

Table 2: **Performance of production queries.** *Cost numbers are in compute hours, latency numbers are in minutes.*

improvements, they both join on one of the dimensions. Q1 performs a simple 2 way join followed by a group-by. We see a huge (>4x) improvement in both cost and latency as the join dominates the query performance (the output of the join is just a few GB). Q2 is more complex, it performs a 3 way join followed by multiple aggregations and produces multiple large outputs (3TB+). *INSTalytics* improves the join by about 3x and does as well as the baseline in the rest of the query. Q3 & Q4 benefit heavily from filters on the other two dimensions processing 34TB of data at interactive latencies.

As seen from our evaluation, simultaneous partitioning on multiple dimensions enables significant improvements in both cost and latency. As discussed in §3, 83% of large files in our production workload require only 4 partition dimensions for full query coverage, and hence can benefit from *INSTalytics*. The workload shown in §3 pertains to a shared cluster that runs a wide variety of queries from several diverse product groups across *Microsoft* so we believe the above finding applies broadly. As we saw in Figure 2, some files need more than 4 dimensions. Simply creating two different copies of the file would cover 8 dimensions, as each copy can use logical replication on a different set of 4 dimensions.

9 Related Work

Co-design The philosophy of cross-layer systems design for improved efficiency has been explored in data center networks [13], operating systems [3] and disk systems [23]. Like [13], *INSTalytics* exploits the scale and economics of cloud data centers to perform cross-layer co-design of big data analytics and distributed storage.

Logical replication The broad concept of logical redundancy has also been explored; the Pilot file system [20] employed logical redundancy of metadata to manage file system consistency, by making file pages self-describing. The technique that *INSTalytics* uses to make extents self-describing for format adapters is similar to the self-describing pages in Pilot. Fractured mirrors [18] leverages the two disks in a RAID-1 mirror to store one copy in row-major and the other in column-major order to improve query performance, but it does not handle recovery of one copy from the other. Another system that exploits the idea of logical replication to speed-up big-data analytics is HAIL [8]; HAIL is perhaps the closest related system to *INSTalytics*; it employs a simpler form of logical replication where they only reorder records within a single storage block; as detailed in Sections 4 and 6, such a layout provides only a fraction of the benefits that the super-extents based layout in *INSTalytics* provides (some benefit to filters but no benefit to joins). As we demonstrate in this paper *INSTalytics* achieves benefits for a wide class of queries without compromising on availability or recovery cost. Replex [25] is a multi-key value store for the OLTP scenario that piggybacks on replication to support multiple indexes for point reads with lower additional storage cost. The recovery cost problem is dealt with by introducing additional hybrid replicas. *INSTalytics* instead capitalizes on the bulk read nature of analytics queries and exploits intra-extent data layout to enable more efficient recovery, without the need for additional replicas. Further the authors do not discuss the availability implications of logical replication, which we comprehensively address in this paper. Erasure coding [19, 12] is a popular approach to achieve fault-tolerance with low storage-cost. However, the recovery cost with erasure coding is much higher than 3-way replication; the layout in *INSTalytics* achieves similar recovery cost as physical replication. Many performance sensitive analytics clusters including ours use 3-way replication.

Data layout In the big-data setting, the benefits of partitioning [30, 27, 24, 21] and co-location [7, 9] are well understood. *INSTalytics* enables partitioning and co-location on multiple dimensions without incurring a prohibitive cost. The techniques in *INSTalytics* are complementary to column-level partitioning techniques such as column stores [15]; in large data sets, one needs both column group-level filtering and row-level partitioning. Logical replication in *INSTalytics* can actually amplify the benefit of column groups by using different (heterogeneous) chocies of column groups in each logical replica within an intra-extent bucket, a focus of ongoing work.

10 Conclusion

The scale and cost of big data analytics, with exabytes of data on the cloud, makes it important from a systems viewpoint. A common approach to speed up big data analytics is to throw parallelism or use expensive hardware (*e.g.*, keep all data in RAM). *INSTalytics* provides a way to *simultaneously* both speed up analytics *and* drive down its cost significantly. *INSTalytics* is able to achieve these twin benefits by fundamentally reducing the actual work done to process queries, by adopting techniques such as logical replication and compute-aware co-ordinated request scheduling. The key enabler for these techniques is the co-design between the storage layer and the analytics engine. The tension in co-design is doing so in a way that only involves surgical changes to the interface, so that the system is pragmatic to build and maintain; with a real implementation in a production stack, we have shown its feasibility. We believe that a similar vertically-integrated approach can benefit other large-scale cloud applications.

Acknowledgements

We thank our shepherd Nitin Agrawal and the anonymous reviewers for their valuable comments and suggestions. We thank Raghu Ramakrishnan, Solom Heddayya, Baskar Sridharan, and other members of the Azure Data Lake product team in Microsoft, for their valuable feedback and support, including providing access to a test cluster.

References

[1] Amp big-data benchmark. In *https://amplab.cs.berkeley.edu/benchmark/*.

[2] ARMBRUST, M., XIN, R. S., LIAN, C., HUAI, Y., LIU, D., BRADLEY, J. K., MENG, X., KAFTAN, T., FRANKLIN, M. J., GHODSI, A., AND ZAHARIA, M. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1383–1394.

[3] ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Information and control in gray-box systems. In *ACM SIGOPS Operating Systems Review* (2001), vol. 35, ACM, pp. 43–56.

[4] BINDSCHAEDLER, L., MALICEVIC, J., SCHIPER, N., GOEL, A., AND ZWAENEPOEL, W. Rock you like a hurricane: Taming skew in large scale analytics. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 20:1–20:15.

[5] BLAUM, M., BRADY, J., BRUCK, J., AND MENON, J. Evenodd: An efficient scheme for tolerating double disk failures in raid architectures. *IEEE Transactions on computers 44*, 2 (1995), 192–202.

[6] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[7] DITTRICH, J., QUIANÉ-RUIZ, J.-A., JINDAL, A., KARGIN, Y., SETTY, V., AND SCHAD, J. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 515–529.

[8] DITTRICH, J., QUIANÉ-RUIZ, J.-A., RICHTER, S., SCHUH, S., JINDAL, A., AND SCHAD, J. Only aggressive elephants are fast elephants. *Proc. VLDB Endow. 5*, 11 (July 2012), 1591–1602.

[9] ELTABAKH, M. Y., TIAN, Y., ÖZCAN, F., GEMULLA, R., KRETTEK, A., AND MCPHERSON, J. Cohadoop: Flexible data placement and its exploitation in hadoop. *Proc. VLDB Endow. 4*, 9 (June 2011), 575–585.

[10] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[11] HSIAO, H.-I., AND DEWITT, D. J. Chained declustering: A new availability strategy for multiprocessor database machines. In *Data Engineering, 1990. Proceedings. Sixth International Conference on* (1990), IEEE, pp. 456–465.

[12] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, 2012), USENIX, pp. 15–26.

[13] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 3–14.

[14] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *ACM SIGPLAN Notices* (1996), vol. 31, ACM, pp. 84–92.

[15] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases* (2010), pp. 330–339.

[16] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (2009), ACM, pp. 165–178.

[17] RAMAKRISHNAN, R., SRIDHARAN, B., DOUCEUR, J. R., KASTURI, P., KRISHNAMACHARI-SAMPATH, B., KRISHNAMOORTHY, K., LI, P., MANU, M., MICHAYLOV, S., RAMOS, R., SHARMAN, N., XU, Z., BARAKAT, Y., DOUGLAS, C., DRAVES, R., NAIDU, S. S., SHASTRY, S., SIKARIA, A., SUN, S., AND VENKATESAN, R. Azure data lake store: A hyperscale distributed file service for big data analytics. In *SIGMOD Conference* (2017).

[18] RAMAMURTHY, R., DEWITT, D. J., AND SU, Q. A case for fractured mirrors. *The VLDB Journal 12*, 2 (2003), 89–101.

[19] RASHMI, K., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 331–342.

[20] REDELL, D. D., DALAL, Y. K., HORSLEY, T. R., LAUER, H. C., LYNCH, W. C., MCJONES, P. R., MURRAY, H. G., AND PURCELL, S. C. Pilot: An operating system for a personal computer. *Communications of the ACM 23*, 2 (1980), 81–92.

[21] SHANBHAG, A., JINDAL, A., MADDEN, S., QUIANE, J., AND ELMORE, A. J. A robust partitioning scheme for ad-hoc query workloads. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, ACM, pp. 229–241.

[22] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–10.

[23] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-smart disk systems.

[24] SUN, L., KRISHNAN, S., XIN, R. S., AND FRANKLIN, M. J. A partitioning framework for aggressive data skipping. *Proc. VLDB Endow. 7*, 13 (Aug. 2014), 1617–1620.

[25] TAI, A., WEI, M., FREEDMAN, M. J., ABRAHAM, I., AND MALKHI, D. Replex: A scalable, highly available multi-index data store. In *USENIX Annual Technical Conference* (2016), pp. 337–350.

[26] THEKKATH, C. A., MANN, T., AND LEE, E. K. *Frangipani: A scalable distributed file system*, vol. 31. ACM, 1997.

[27] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow. 2*, 2 (Aug. 2009), 1626–1629.

[28] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., GHODSI, A., GONZALEZ, J., SHENKER, S., AND STOICA, I. Apache spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.

[29] ZHANG, H., CHO, B., SEYFE, E., CHING, A., AND FREEDMAN, M. J. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 43:1–43:15.

[30] ZHOU, J., BRUNO, N., WU, M.-C., LARSON, P.-A., CHAIKEN, R., AND SHAKIB, D. Scope: Parallel databases meet mapreduce. *The VLDB Journal 21*, 5 (Oct. 2012), 611–636.

[31] ZHOU, J., LARSON, P. A., AND CHAIKEN, R. Incorporating partitioning and parallel plans into the scope optimizer. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)* (March 2010), pp. 1060–1071.

GRAPHONE: A Data Store for Real-time Analytics on Evolving Graphs

Pradeep Kumar H. Howie Huang
The George Washington University

Abstract

There is a growing need to perform real-time analytics on evolving graphs in order to deliver the values of big data to users. The key requirement from such applications is to have a data store to support their diverse data access efficiently, while concurrently ingesting fine-grained updates at a high velocity. Unfortunately, current graph systems, either graph databases or analytics engines, are not designed to achieve high performance for both operations. To address this challenge, we have designed and developed GRAPHONE, a graph data store that combines two complementary graph storage formats (edge list and adjacency list), and uses *dual versioning* to decouple graph computations from updates. Importantly, it presents a new data abstraction, *GraphView*, to enable data access at two different granularities with only a small data duplication. Experimental results show that GRAPHONE achieves an ingestion rate of two to three orders of magnitude higher than graph databases, while delivering algorithmic performance comparable to a static graph system. GRAPHONE is able to deliver 5.36× higher update rate and over 3× better analytics performance compared to a state-of-the-art dynamic graph system.

1 Introduction

We live in a world where information networks have become an indivisible part of our daily lives. A large body of research has studied the relationships in such networks, e.g., biological networks [33], social networks [20, 41, 46], and web [9, 31]. In these applications, graph queries and analytics are being used to gain valuable insights from the data, which can be classified into two broad categories: *batch analytics* (e.g. PageRank [61], graph traversal [11, 49, 51]) that analyzes a static snapshot of the data, and *stream analytics* (e. g. anomaly detection [8], topic detection [64]) that studies the incoming data over a time window of interest. Generally speaking, batch analytics prefers a *base (data) store* that can provide indexed access on the non-temporal property of the graph such as the source vertex of an edge, and on the other hand, stream analytics needs a *stream (data) store* where data can be stored quickly and can be indexed by their arrival order for temporal analysis.

Increasingly, one needs to perform batch and stream processing together on evolving graphs [78, 68, 10, 69]. The key requirement here is to sustain a large volume of fine-grained updates at a high velocity, and simultaneously provide high-performance real-time analytics and query support.

This trend poses a number of challenges to the underlying storage and data management system. First, batch and stream analytics perform different kinds of data access, that is, the former visits the whole graph while the latter focuses on the data within a time window. Second, each analytic has a different notion of real time, that is, data is visible to the analytics at different granularity of data ingestion (updates). For example, an iterative algorithm such as PageRank can run on a graph that is updated at a coarse granularity, but a graph query to output the latest shortest path requires data visibility at a much finer granularity. Third, such a system should also be able to handle a high arrival rate of updates, and maintain data consistency while running concurrent batch and stream processing tasks.

Unfortunately, current graph systems can neither provide diverse data access nor at the right granularity in the presence of a high data arrival rate. Many dynamic graph systems [47, 54] only support batched updates, and a few others [21, 70] offer data visibility at fine granularity of updates but with a weak consistency guarantee, which as a result may cause an analytic iteration to run on different data versions and produce undesired results. Relational and graph databases such as Neo4j [59] can handle fine-grained updates, but suffer from poor ingestion rate for the sake of strong consistency guarantee [56]. Also, such systems are not designed to support high-performance streaming data access over a time window. On the other hand, graph stream engines [58, 17, 32, 72, 75, 67] interleave incremental computation with data ingestion, i.e., graph updates are batched and not applied until the end of an iteration. In short, the existing systems manage a private data store in a way to favor their specialized analytics.

In principle, one can utilize these specialized graph systems side-by-side to provide data management functions for dynamic graphs and support a wide spectrum of analytics and queries. However, such an approach would be suboptimal [78], as it is only as good as the weakest component, in many cases the graph database with poor performance for streaming data. Worse, this approach could also lead to excessive data duplication, as each subsystem would store a replica of the same underlying data in their own format.

In this work, we have designed GRAPHONE, a unified graph data store offering diverse data access at various granularity levels while supporting data ingestion at a high ar-

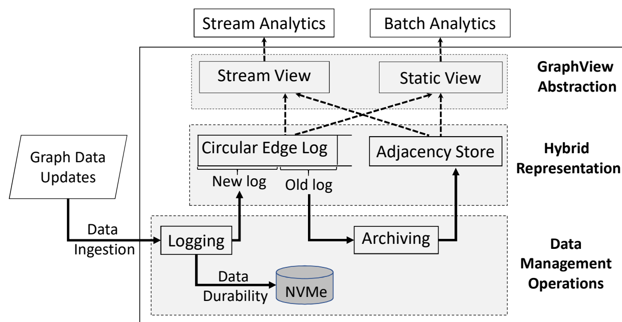


Fig. 1: High-level architecture of GRAPHONE. Solid and dotted arrows show the data management and access flow respectively.

rival rate. Fig. 1 provides a high-level overview. It leverages a *hybrid graph store* to combine a small circular edge log (henceforth *edge log*) and an *adjacency store* for their complementary advantages. Specifically, the edge log keeps the latest updates in the edge list format, and is designed to accelerate data ingestion. At the same time, the adjacency store holds the snapshots of the older data in the adjacency list format that is moved periodically from the edge log, and is optimized for batch and streaming analytics. It is important to note that the graph data is not duplicated in two formats, although a small amount of overlapping is allowed to keep the original composition of the versions intact.

GRAPHONE enforces data ordering using the temporal nature of the edge log, and keeps the per-vertex edge arrival order intact in the adjacency store. A *dual versioning* technique then exploits the fine-grained versioning of the edge list format and the coarse-grained versioning of the adjacency list format to create real-time versions. Further, GRAPHONE allows independent execution of analytics that run parallel to data management, and can fetch a new version at the end of its own incremental computation step. Additionally, we provide two optimization techniques, cacheline sized memory allocation and special handling of high degree vertices of power-law graphs, to reduce the memory requirement of versioned adjacency store.

GRAPHONE simplifies the diverse data access by presenting a new data abstraction, *GraphView*, on top of the hybrid store. Two types of GraphView are supported as shown in Fig. 1 : (1) the *static view* offers real-time versioning of the latest data for batch analytics; and (2) the *stream view* supports stream analytics with the most recent updates. These views offers *visibility* of data updates to analytics at two levels of granularity where the edge log is used to offer it at the edge level, while the adjacency store provides the same at coarse granularity of updates. As a result, GRAPHONE provides high-level applications with the flexibility to trade-off the granularity of data visibility for a desired performance. In other words, the edge log can be accessed if fine-grained data visibility is required, which can be tuned (§7.3).

We have implemented GRAPHONE as an in-memory graph datastore with a durability guarantee on external non-volatile memory express solid-state drives (NVMe SSD). For comparison, we have evaluated it against three types of in-memory graph systems: Neo4j and SQLite, two graph data management systems; Stinger [21], a dynamic graph system; and Galois [60], a static graph system, as well as GRAPHONE itself working with static graphs. The experimental results show that GRAPHONE can support a high data ingestion rate, specifically it achieves two to three orders of magnitude higher ingestion rate than graph databases, and 5.36× higher ingestion rate than Stinger. In addition, GRAPHONE outperforms Stinger by more than 3× on different analytics, and delivers equivalent algorithmic performance compared to Galois. The stream processing in GRAPHONE runs parallel to data ingestion which offers 26.22% higher ingestion rate compared to the current practice of interleaving the two.

To summarize, GRAPHONE makes three contributions:

- Unifies stream and base stores to manage the graph data in a dynamic environment;
- Provides batch and stream analytics through dual versioning, smart data management, and memory optimization techniques;
- Supports diverse data access of various usecases with GraphView and data visibility abstractions.

The rest of the paper is organized as follows. We present a usecase in §2, opportunities and GRAPHONE overview in §3, the hybrid store in §4, data management internals and optimizations in §5, GraphView data abstraction in §6, evaluations in §7, related work in §8, and conclusion in §9.

2 Use Case: Network Analysis

Graph analytics is a natural choice for data analysis on an enterprise network. Fig. 2(a) shows a graph representation of a simple computer network. Such a network can be analyzed in its entirety by calculating the diameter [48], and betweenness centrality [13] to identify the articulation points. This kind of batch analysis is very useful for network infrastructure management. In the meantime, as the dynamic data flow within the network captures the real-time behaviors of the users and machines, the stream analytics is used to identify security risks, e.g., denial of service, and lateral movement, which can be expressed in the form of path queries, parallel paths and tree queries on a streaming graph [38, 18].

Los Alamos Nation Laboratory (LANL) recently released a comprehensive data set [37] that captures a wide range of network information, including authentication events, process events, DNS lookups, and network flows. The LANL data covers over 1.5 billion events, 12,000 users, and 17,000 computers, and spans 58 consecutive days. For example, the network authentication data captures the login information that a user logs in to a network machine, and also from that machine to other machines. When the network defense system identifies a malicious user and node, it needs to find all the nodes that may have been infected. Instead of analyzing every node of the network, one can quickly run a path traver-

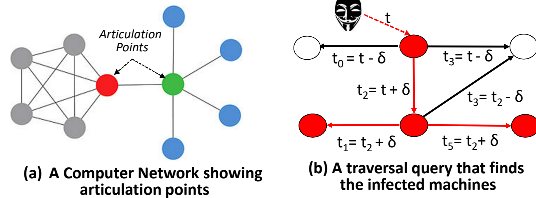


Fig. 2: Graph traversal can locate possible infected nodes using real-time authentication graph if infected user and node are known

sal query on the real-time authentication graph to identify the possible infected nodes, that is, find all the nodes whose login has originated from the chain of nodes that are logged in from the first infected machine [38] as shown in Fig. 2(b).

In summary, a high-performance graph store that captures dynamic data in the network, combined with user, machine information and network topology, is advantageous in understanding the health of the network, accelerating network service, and protecting it against various attacks. This work presents a graph storage and APIs for such usecases.

3 Opportunities and Overview

A graph can be defined as G = (V, E, W), where V is the vertex set, and E is the edge set, and W is the set of edge weights. Each vertex may also have a label. In this section, graph formats and their traits are described as relevant for GRAPHONE, and then we present its high-level overview.

3.1 Graph Representation: Opportunities

Fig. 3 shows three most popular data formats for a sample graph. First, the *edge list* is a collection of edges, a pair of vertices, and captures the incoming data in their arrival order. Second, the *compressed sparse row (CSR)* groups the edges of a vertex in an *edge array*. There is a metadata structure, *vertex array*, that contains the index of the first edge of each vertex. Third, the *adjacency list* manages the neighbors of each vertex in separate per-vertex edge arrays, and the vertex array stores a count (called degree) and pointer to indicate the length and the location of the corresponding edge arrays respectively. This format is better than the CSR for ingesting graph updates as it affects only one edge array at a time.

In the edge list, the neighbors of each vertex are scattered across, thus is not the optimal choice for many graph queries and batch analytics who prefer to get the neighboring edges of a vertex quickly [34, 29, 30, 12] etc . On the other hand, the adjacency list format loses the temporal ordering as the incoming updates get scattered over the edge arrays, thus not suited for stream analytics. Given their advantages and disadvantages, neither format is ideally suited for supporting both batch and stream analytics on its own. We now identify two opportunities for this work:

Opportunity #1: Utilize both the edge list and the adjacency list within a hybrid store. The edge list format preserves the data arrival order and offers a good support for fast updates as each update is simply appended to the end of the list. On the other hand, the adjacency list keeps all the neighbors of a vertex indexed by the source vertex, which provides efficient data access for graph analytics. Thus it allows GRAPHONE to achieve high-performance graph computation while simultaneously supporting fine-grained updates.

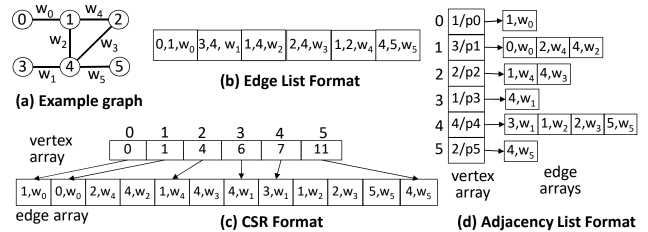


Fig. 3: Sample graph and its various storage format

Opportunity #2: Fine-grained snapshot creation with the edge list format. Graph analytics and queries require an immutable snapshot of the latest data for the duration of their execution. The edge list format provides a natural support for fine-grained snapshot creation without creating a physical snapshot due to its temporal nature, as tracking a snapshot is just remembering an offset in the edge list. Meanwhile, the adjacency list format through its coarse-grained snapshot capability [54, 26] is used to complement the edge list.

3.2 Overview

GRAPHONE utilizes a hybrid graph data store (discussed in §4) that consists of a small circular *edge log* and the *adjacency store*. Fig. 4 shows an high-level overview of GRAPHONE architecture. The hybrid store is managed in several phases (presented in §5). Specifically, during the *logging phase*, the edge log records the incoming updates in the edge list format in their arrival order, and supports a high ingestion rate. We define *non-archived edges* as the edges in the edge log that are yet to be moved to the adjacency store. When their number crosses the *archiving threshold*, a parallel *archiving phase* begins, which merges the latest edges to the adjacency store to create a new adjacency list snapshot. This duration is referred to as an *epoch*. In the *durable phase*, the edge log is written to a disk.

To efficiently create and manage immutable versions for data analytics in presence of the incoming updates, we provide a set of GraphView APIs (discussed in §6). Specifically, *static view* API is for batch processing, while *stream view* API is for stream processing. Internally, the views utilize *dual versioning* technique where the versioning capability of both formats are exploited. For example, a real-time static view can be composed by using the latest coarse-grained version of the adjacency store, and the latest fine-grained version of non-archived edges.

It is important to note that the GraphView also provides analytics with the flexibility to trade-off the granularity of data visibility for better performance, e.g., the analytics that prefer running only on the latest adjacency list store will avoid the cost associated with the access of the latest edges from the non-archived edges.

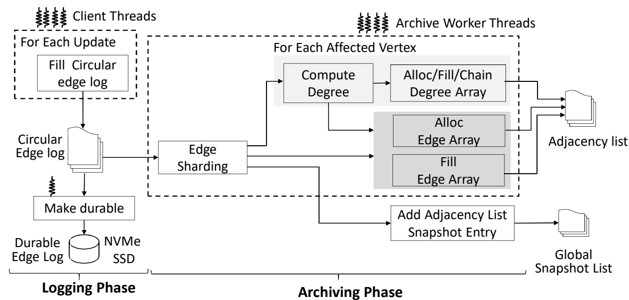


Fig. 4: Architecture of GRAPHONE. Operations related to same data structures have been grayed out in archiving phase. Compaction Phase is not shown.

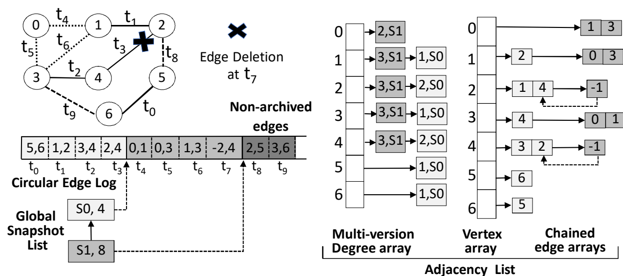


Fig. 5: The hybrid store for the data arrived from time t_0 to t_9: The vertex array contains pointers to the first and the last block of each edge array, while degree array contains deleted and added edge counts. However, only the pointer to the first block in the vertex array, and total count in the degree array are shown for brevity.

4 Hybrid Store

The hybrid store design presented in Fig. 5 consists of a small *circular edge log* that is used to record the latest updates in the edge list format. For deletion cases, we use tombstones, specifically the edge log also adds a new entry but the most significant bit (MSB) of the source vertex ID of the edge is set to denote its deletion as shown in Fig 5 for deleted edge $(2,4)$ at time t_7.

The *adjacency store* keeps the older data in the adjacency list format. The adjacency store is composed of *vertex array*, per-vertex *edge arrays*, and multi-versioned *degree array*. The *vertex array* contains a per-vertex flag and pointers to the first and last block of the edge arrays. Addition of a new vertex is done by setting a special bit in the per-vertex flag. Vertex deletion sets another bit in the same flag, and adds all of its edges as deleted edges to the edge log. These bits help GRAPHONE in garbage collecting the deleted vertex ID.

The *edge array* contains per-vertex edges of the adjacency list. It may contain many small edge blocks, each of which contains a count of the edges in the block and a memory pointer to the next block. The connection of edge blocks are referred to as *chaining*. An edge addition always happens at the end of the edge array of each vertex, which may require the allocation of a new edge block and linked to the last block. Fig. 5 shows chained edge arrays for the vertices with ID 1 to 4 for data updates that arrive in between t_4 to t_7. The adjacency list treats an edge deletion as an addition but the deleted edge entry in the edge array keeps the negative position of the original edge, while the actual data is not modified at all, as shown for edge $(2,4)$. As a result, deletion never breaks the convergence of a previous computation as it does not modify the dataset of the computation.

The *degree array* contains the count of neighboring edges of each vertex. Thus, a degree array from an older adjacency store snapshot can identify the edges to be accessed even from the latest edge arrays due to the latter's append-only property. Hence, the degree array in GRAPHONE is *multi-versioned* to support adjacency store snapshots. It keeps the total added and deleted edge counts of each vertex. Both counts help in efficiently getting the valid neighboring edges, as a client can do the exact memory allocation (refer to the *get-nebrs-*()* API in Table 2). When an edge is added or deleted for a vertex, a new entry is added for this vertex in the degree array in each epoch. Two different versions $S0$ and $S1$ of the degree array are shown in Fig. 5 for two epochs $t_0 - t_3$ and $t_4 - t_7$.

One can note that degree nodes are shared across epochs if there is no later activity in a vertex. For example, the same degree nodes for vertices with ID 5 and 6 are valid for both epochs in Fig. 5. The degree array nodes of an older versions may be garbage collected when the corresponding adjacency store snapshot retires, i.e., not being used actively by any analytics, and is tracked using reference counting mechanism through the global snapshot list, which will be discussed shortly. For example, if snapshot S0 is retired, then the degree nodes of snapshot S0 for vertices with ID $1-4$ can be reused by later snapshots (e.g. S2).

The *global snapshot list* is a linked list of snapshot objects to manage the relationship between the edge log and adjacency store at each epoch. Each node contains an absolute offset to the edge log where the adjacency list snapshot is created, and a reference count to capture the number of views using this adjacency list snapshot. A new entry in the global snapshot list is created after each epoch, and it implies that the edge log data of the last epoch has been moved to the adjacency store atomically, and is now visible to the world.

Weighted Graphs. Edge weights are generally embedded in the edge arrays along with the destination vertex ID. Some graphs have static weights, e.g., an edge weight in an enterprise network can represent the network speed between the two nodes. A weight change is then treated internally as an edge deletion followed by an edge addition. On the other hand, if edge weights are dynamic, such as network data flow, then such weights are suited for various analytics if kept for a configurable time window, e.g., anomaly detection in the network flow. In this case GRAPHONE is configured to treat weight changes as a new edge to aid such analytics.

Dual Versioning and Data Overlap GRAPHONE uses *dual versioning* to create the instantaneous read-only graph views (snapshot isolation) for data analytics. It exploits both the fine-grained versioning property of the edge log, and the

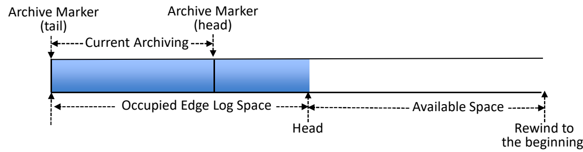


Fig. 6: Circular Edge log design showing various offset or markers. Markers for durable phase are similar to archiving and are omitted.

coarse-grained versioning capability of the adjacency list format. It should be noted that the adjacency list provides one version per epoch, while the edge log supports multiple versions per epoch, as many as the number of edges arrived during the epoch. So the dual versioning provides many versions within an epoch which is the basis for static views, and should not be confused with the adjacency list snapshots. In Fig. 5, static view at the time t_6 would be adjacency list snapshot $S0$ plus the edges from $t_4 - t_6$.

A small amount of *data overlap* between the two stores keeps the composition of the view intact. This makes the view accessible even when the edge log data is moved to the adjacency store to create a new adjacency list version. Thus both stores have the copy of a few epochs of the same data. For one or more long running iterative analytics, we may use the durable edge log or a private copy of non-archived edges to provide data overlap, so that analytics can avoid interfering with data management operations of the edge log.

5 Data Management and Optimizations

Data management faces the key issues of minimizing the size of non-archived edges, providing atomic updates, data ordering, and cleaning of older snapshots. Addition and deletion of vertices and edges, and edge weight modification are all considered as an atomic update.

5.1 Data Management Phases

Fig. 4 depicts the internals of the data management operations. It consists of four phases: *logging*, *archiving*, *durable* and *compaction*. Client threads send updates, and the logging to the edge log happens in the same thread context synchronously. The archiving phase moves the non-archived edges to the adjacency store using many worker threads, and one of them assumes the role of the master, called the archive thread. The durable phase happens in a separate thread, while compaction is multi-threaded but happens much later.

A client thread wakes up the archive thread and durable thread to start the archiving and durable phases when the number of non-archived edges crosses a threshold, called *archiving threshold*. The logging phase continues as usual in parallel to them. Also, the archive thread and durable thread check if any non-archived edges are there at the end of each phase to repeat their process, or wait for work with a timeout.

The edge log has a distinct offset or marker, *head*, for logging, which is incremented every time an edge is ingested as shown in Fig. 6. For archiving, GraphOne manages a pair of markers, i.e. the archiving operation happens from

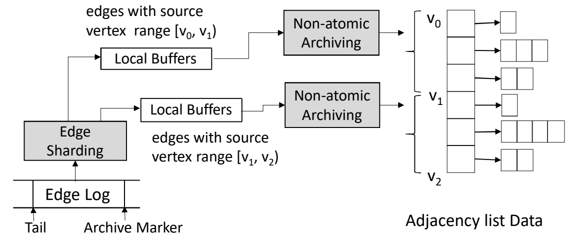


Fig. 7: Edge sharding separates the non-archived edges into many buffers based on their source vertex ID, so that the per-vertex edge arrays can keep the edge log arrival order, and enables non-atomic archiving.

the *tail archive marker* to the *head archive marker*, because the *head* will keep moving due to new updates. The durable phase also has a pair of markers to work with. Markers are always incremented and used with the modulo operator.

5.1.1 Logging Phase

The incoming update is converted to numerical identifiers, and acquires an edge list format. The mapping between vertex label to vertex ID and vice-versa manages this translation. Then a unique spot is claimed within the edge log by the atomic increment of the head, and the edge is written to a spot calculated using the modulo operation on the head, that also stores the operator (§4), addition or deletion, along with the edges. The atomicity of updates is ensured by the atomic increment of the head. The edge log is automatically reused in the logging phase due to its circular nature, and thus is overwritten by newer updates. Hence the logging may get blocked occasionally if the whole buffer is filled as the archiving or durable phases may not be able to catch up. We keep sufficiently large edge log to avoid frequent blocking. In case of blocked client threads, they are woken up when the archiving or durable phases complete.

5.1.2 Archiving Phase

This phase moves the non-archived edges from the edge log to the adjacency store. A naive multi-threaded archiving, where each worker can directly work on a portion of non-archived edges, may not keep the data ordering intact. If a deletion comes after the addition of an edge within the same epoch, the edge may become alive or dead in the edge arrays depending on the archiving order of the two data points.

An *edge sharding* stage in the archiving phase (Fig. 7) maintains per-vertex edges as per the edge log arrival to address the ordering problem. It shards the non-archived edges to multiple local buffers based on the range of their source vertex ID. For undirected graphs, the total edge count in the local buffer is twice of the non-archived edge count, as the ordering of reverse edges is also managed. For directed edges, both directions have their own local buffers.

The edges in each local buffer are then archived in parallel without using any atomic instructions. A heuristic is required for workload distribution, as the equal division is not possible among threads, thereby the last thread may get

more work assigned. To handle the workload imbalance among worker threads, we create a larger number of local buffers with smaller vertex range than the available threads, and assign different numbers of local buffers to each thread so that each gets an approximately equal number of edges to archive. The idea here is to assign slightly more than equal work to each thread, so that all the threads are balanced while the last thread is either balanced or lightly loaded.

This stage allocates new degree nodes or can reuse the same from the older degree array versions if they are not being used by any analytics. We follow these rules for reusing the degree array from older versions. We track the degree array usage by analytics using reference counting per epoch [40], and can be reused if all static views created within that epoch have expired, i.e., the references are dropped to zero (not being used by any running analytics). It also ensures that a newly created view uses the latest adjacency list snapshot that should never be freed.

The stage then populates the degree array, and allocates memory for edge blocks that are chained before filling those blocks. We then create a new snapshot object, fill it up with relevant details, and add it atomically to the global snapshot list. At the end of the archiving phase, the archive thread sets the tail archive marker atomically to the value of the head archive marker, and wakes up any the blocked client threads.

5.1.3 Durable Phase and Recovery

The edge log data is periodically appended to a durable file in a separate thread context instead of logging immediately to the disk to avoid the overhead of IO system calls during each edge arrival. Also this will not guarantee durability unless fsync() is called. The logging uses buffered sequential write, and allows the buffer cache to work as spillover buffer for the access of non-archived edges if the edge log is over-written.

The durable edge log is a prefix of the whole ingested data, so GRAPHONE may lose some recent data in the case of an unplanned shutdown. The recovery depends on upstream backup that keep the latest data for some time, such as kafka [42], and replays it for the lost data, and creates the adjacency list on the whole data. Recovery is faster than building the data structures at an edge level, as only the archiving phase is involved working on bulk of data. Alternatively, persistent memory may be used for the edge log to provide durability at each update [45].

The durable phase also performs an incremental checkpointing of the adjacency store data from an old time-window, and frees the memory associated with it. This is useful for streaming data such as LANL network flow, where the old adjacency data can be checkpointed in disk, as the in-memory adjacency store within the latest time window is sufficient for stream analytics. By default, it is not enabled. During checkpointing the adjacency store, the vertex ID and length of the edge array are persisted along with edge arrays so that data can be read easily later, if required.

5.1.4 Compaction Phase

The compaction of the edge arrays removes deleted data from per-vertex edge array blocks up to the latest retired snapshot identified via the reference counting scheme discussed in §5.1.2. The compaction needs a similar reference counting for the private static views (§6.1). For each vertex, it allocates new edge array block and copies valid data up to the latest retired snapshot from the edge arrays, and creates a link to the rest of the original edge array blocks. The newly created edge array block is then atomically replaced in the vertex array, while freeing happens later to ensure that cached references of the older data are dropped. This phase is generally clubbed with archiving phase where the degree array is updated to reflect the new combination.

5.2 Memory Overhead and Optimizations

The edge log and degree array are responsible for versioning. The edge log size is relatively small as it contains only the latest updates which moves quickly to the base store, e.g, the archiving threshold of 2^{16} edges translates to only 1MB for a plain graph assuming 8 byte vertex ID. Thus the edge log is only several MBs. The memory in degree arrays are also reused (§5.1.2). This leaves us to memory analysis of edge arrays which may consume a lot of memory due to excessive chaining in their edge blocks. For example, GRAPHONE runs archiving phase for 2^{16} times for Kron-28 graph if the archiving threshold is 2^{16}. In this case, the edge arrays would consume 148.73GB memory and have average 29.18 chain per-vertex. We will discuss the graph datasets used in this paper shortly. If all the edges were to be ingested in one archiving phase, this static system needs only an average 0.45 chain and 33.80GB memory. The chain count is less than one as 55% vertices do not have any neighbor.

GRAPHONE uses two memory allocation techniques, as we discuss next, to reduce the level of chaining to make the memory overhead of edge arrays modest compared to a static engine. The techniques work proactively, and do not affect the adjacency list versioning. Compaction further reduces the memory overhead to bring GRAPHONE at par with static analytics engine, but is performed less frequently.

Optimization #1: Cacheline Sized Memory Allocation. Multiples of cacheline sized memory is allocated for the edge blocks. One cacheline (64 bytes) can store up to 12 neighbors for the plain graph of 32bit type, leaving the rest of the space for storing a count to track space usage in the block and a link to the next block. In this allocation method, the ma-

Table 1: Impact of two optimizations on the chain count and memory consumption on the kronecker graph.

Optimizations	Chain Count		Memory Needed (GB)
	Average	Maximum	
Baseline System	29.18	65,536	148.73
+Cacheline memory	2.96	65,536	47.42
+Hub Vertex Handling	2.47	3,998	45.79
Static System	0.45	1	33.81

Table 2: Basic GraphView APIs

Static View APIs	
snap-handle	create-static-view(global-data, simple, private, stale)
status	delete-static-view(snap-handle)
count	get-nebr-length-{in/out}(snap-handle, vertex-id)
count	get-nebrs-{in/out}(snap-handle, vertex-id, ptr)
count	get-nebrs-archived-{in/out}(snap-handle, vertex-id, ptr)
count	get-non-archived-edges(snap-handle, ptr)
Stateless Stream View APIs	
stream-handle	reg-stream-view(global-data, window-sz, batch-sz)
status	update-stream-view(stream-handle)
status	unreg-stream-view(stream-handle)
count	get-new-edges-length(stream-handle)
count	get-new-edges(stream-handle, ptr)
Stateful Stream View APIs	
sstream-handle	reg-sstream-view(global-data, window-sz, v-or-e-centric, simple, private, stale)
status	update-sstream-view(sstream-handle)
status	unreg-sstream-view(sstream-handle)
bool	has-vertex-changed(sstream-handle, vertex-id)
count	get-nebr-length-{in/out}(sstream-handle, vertex-id)
count	get-nebrs-{in/out}(sstream-handle, vertex-id, ptr)
count	get-nebrs-archived-{in/out}(sstream-handle, vertex-id, ptr)
count	get-non-archived-edges(sstream-handle, ptr)
Historic View APIs	
count	get-prior-edges(global-data, start, end, ptr)

jority of the vertices will need only a few levels of chaining. For example, in a Twitter graph, 88.43% of the vertices will need at most 3 cachelines only, and so do 92.49% for Kron-28 graph. This optimization reduces the average chain count by 9.88×, and memory consumption by 3.14× in comparison to a baseline system as shown in Table 1. The baseline system uses a dynamic block size which is equivalent to the number of edges arrived during each epoch for each vertex.

Optimization #2: Hub Vertex Handling. A few vertices, called hub-vertices, have very high degree in a graph that follows power-law distribution [22]. They are very common in real-life graphs, such as for the twitter follower graph whose degree distribution we analyze. Such vertices are likely to participate in each archiving phase. Hence they will have a lot of chaining in their edge arrays, and the aforementioned memory management technique alone is not enough. In this case, we allocate in multiples of 4KB page-aligned memory for vertices that already have 8,192 edges or if the number of neighbors in any archiving phase crosses 256. The average chain count is further reduced to 2.47, leading to reduction in memory utilization by 1.63GB as listed in Table 1. One can vary the threshold to identify a hub vertex but performance remains similar to the cacheline sized memory (Fig. 15).

6 GraphView Abstraction

GraphView data abstraction hides the complexity of the hybrid store by providing simple data access APIs as shown in Table 2. The *static view* is suited for batch analytics and queries, while the *stream view* for stream processing. Both offer diverse data access at two granularities of data visibility of updates. At any time, a number of views may co-exist without incurring much memory overhead, as the view data

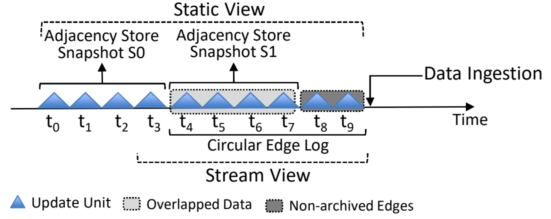


Fig. 8: GraphOne hybrid store illustrating various views with two adjacency store versions, S0 and S1, with a small edge log

is composed of the same adjacency store and non-archived edges as shown in Fig. 8. The access of non-archived edges provides data visibility at the edge level granularity.

Due to the cost of indexing the non-archived edges, GraphView provides an option to trade-off the granularity of data visibility to gain performance. Further, one can use vertex-centric compute model [73] on the adjacency list plus edge-centric compute model [81, 43, 66] on non-archived edges, so there is no need to index the latter as plotted later to find its optimal minimum size (Fig. 13).

6.1 Static View

Batch analytics and queries prefer snapshots for computation, which can be created in real-time using *create-static-view()* API. It is represented by an opaque handle that identifies the view composition, i.e., the non-archived edges and the latest adjacency list snapshot, and serves as input to other static view APIs. A created handle should be destroyed using *delete-static-view()*. Based on the input supplied to *create-static-view()* API, many types of static view are defined.

Basic Static View. This view is very useful for advanced users and higher level library development which prefer more control and performance. The main low-level API are: *get-nebrs-archived-*()* that returns the reference to the per-vertex edge array; and *get-non-archived-edges()* that returns the non-archived edges. On the other hand, it also provides a high-level API, *get-nebrs-*()*, that returns the neighbor list of a vertex by combining the adjacency store and the non-archived edges in a user supplied memory buffer. It may be preferable by queries with high selectivity that only need to scan the non-archived edges for one or a few vertex, e.g. 1-hop query, and is not apt for long running analytics.

The implementation of *get-nebrs()* for the non-deletion case is a simple two step process: copy the per-vertex edge array to the user supplied buffer, followed by a scan of the non-archived edges to find and add the rest of the edges of the vertex to the buffer. For the deletion case, both the steps track the deleted positions in the edge arrays, and the last few edges from edge arrays and/or non-archived edge log are copied into those indexes of the buffer.

Private Static View. For long running analytics, keeping basic static views accessible have some undesirable impacts: (1) all the static views may have to use the durable edge log if the corresponding non-archived edges in the edge log has been overwritten; (2) the degree array cannot be reused in

Algorithm 1 Traditional BFS using static view APIs

```
1:  handle ← create-static-view(global-data, private=true, simple=true)
2:  level = 1; active-vertex = 1; status-array[root-vertex] = level;
3:  while active-vertex do
4:      active-vertex = 0;
5:      for vertex-type v = 0; v < vertex-count; v++ do
6:          if status-array[v] == level then
7:              degree ← get-nebrs-out(handle, v, nebr-list);
8:              for j=0; j < degree; j++ do
9:                  w ← nebr-list[j];
10:                 if status-array[w] == 0 then
11:                     status-array[w] ← level + 1; ++active-vertex;
12:     ++level;
13: delete-static-view(handle)
```

the archiving phase as it is still in use. To solve this, one can create a *private static view* by passing *private=true* in the *create-static-view()* API. In this case, a private copy of the non-archived edges and the degree array are kept inside the view handle with their global references dropped to make it independent from archiving. One can pass *simple=true* in the *create-static-view()* to create a temporary in-memory adjacency list from the non-archived edges for optimizing get-nebr-*() API, as shown in Algorithm 1 for a simplified BFS (push model) implementation. This approach is more flexible than static analytics engine which converts the whole data, or dynamic graph system that disallows the user to choose fine-grained control on snapshot creation.

Creation to many private static views may introduce memory overhead. To avoid this, a reference of the private degree array is kept in the snapshot object and is shared by other static views created within that epoch, and are locally reference counted for freeing. Thus, creating many private views within an epoch has overhead of just one degree array. However, creating many private static views across epochs may still cause the memory overhead, if older views are still being accessed by long running analytics. This also means that the machine is overloaded with computations, and they are not real-time in nature. In such a case, a user may prefer to copy the data to another machine to execute them.

Stale Static View. Many analytics are fine with data visibility at coarse-grained ingestions, thus some stale but consistent view of the data may be better for their performance. In this case, passing *stale=true* returns the snapshot of the latest adjacency list only. This view can be combined with private static view where degree array will be copied.

6.2 Stream View

Stream computations follow a pull method in GraphOne, i.e., the analytics pulls new data at the end of incremental compute to perform the next phase of incremental compute. The stream view APIs around the handle simplify the data access and its granularity in presence of the data ingestion. Also, checkpointing the computation results and the associated data offset is the responsibility of the stream engine, so that the long running computation can be resumed from that point onwards in case of a fault.

Algorithm 2 A stateless stream compute skeleton

```
1: handle ← reg-stream-view(global-data, batch-sz=10s)
2: init-stream-compute(handle)                    ▷ Application specific
3: while true do                                  ▷ Or application specific criteria
4:     if update-stream-view(handle) then
5:         count = get-new-edges(handle, new-edges)
6:         for j=0; j < count; j++ do
7:             do-stream-compute(handle, new-edges[j]) ▷ Or any method
8: unreg-sstream-view(handle)
```

Stateless Stream Processing. A stateless computation, e.g. counting incoming edges (aggregation), only needs a batch of new edges. It can be registered using the *reg-stream-view()* API, and the returned handle contains the batch of new edges. Algorithm 2 shows how one can use the API to do stateless stream computation. The handle also allows a pointer to point to analytics results to be maintained by the stream compute implementation. The implementation also needs to checkpoint only the edge log offset and the computation results as GraphOne keeps the edge log durable.

An extension of the model is to process on a data window instead on the whole arrived data. For sliding window implementation, GraphOne manages a cached batch of edge data around the start marker of the data window in addition to the batch of new edges. The old cached data can be accessed by the analytics for updating the compute results, e.g., subtracting the value in aggregation over the data window. The cached data is fetched from the durable edge log, and shows sequential read due to the sliding nature of the window. A tumbling window implementation is also possible where the batch size of new edges is equal to the window size, and hence does not require older data to be cached. Additional checkpointing of the starting edge offset is required along with the edge log offset and computation results.

Stateful Stream Processing. A complex computation, such as graph coloring [67], is stateful that needs the streaming data and complete base store to access the computational state of the neighbors of each vertex. A variant of static view is better suited for it because its per-vertex neighbor information eases the access of the computational state of neighbors. It is registered using *reg-sstream-view()*, and returns *sstream-handle*. For edge-centric computation, the handle also contains a batch of edges to identify the changed edges. For vertex-centric computation, the handle contains per-vertex one-bit status to denote the vertex with edge updates that can be identifies using the *has-vertex-changed()* API. This is updated during *update-sstream-view()* call that also updates the degree array. Algorithm 3 shows an example code snippet.

As the degree array plays an important role for a stateful computation due to its association with the static view, using an additional degree array at the start marker of the data window eases the access of the data within the window from the adjacency store. The *sstream-handle* manages the degree array on behalf of the stream engine, and internally keeps a batch of cached edges around the start marker of the window

Algorithm 3 A stateful stream compute (vertex-centric) skeleton

```
1: handle ← reg-sstream-view(global-data, v-centric, stale=true)
2: init-sstream-compute(handle)                 ▷ Application specific
3: while true do                                ▷ Or application specific criteria
4:   if update-sstream-view(handle) then
5:     for v=0; v < vertex-count; v++ do
6:       if has-vertex-changed(handle, v) then
7:         do-sstream-compute(handle, v)        ▷ Application specific
8: unreg-sstream-view(handle)
```

to update the old degree array. The *get-nebrs-*()* function returns the required neighbors only. Checkpointing the computational results, the edge log offset at the point of computation, and window information is sufficient for recovery.

6.3 Historic Views

GRAPHONE provides many views from recent past, but it is not designed for getting arbitrary historic views from the adjacency store. However, durable edge log can provide the same using *get-prior-edges()* API in edge list format as it keeps deleted data, behaving similar to existing data stores [14, 23]. Moreover, in case of no deletion, one can create a degree array at a durable edge log offset by scanning the durable edge log, and the degree array will serve older static or stream view from the adjacency store to gain insights from the historical data. For data access from a historical time-window in this case, one need to build two degrees arrays at both the offsets of the durable edge log.

7 Evaluations

GRAPHONE is implemented in around 16,000 lines of C++ code including various analytics. It supports plain graphs and weighted graphs with either 4 byte or 8 byte vertex sizes. We store the fixed weights along with the edges, variable length weights in a separate weight store using indirection. Any type of value can be stored in place of weight such as integers, float/double, timestamps, edge-id or any custom weight as the code is written using C++ templates. So one can write a small plug-in describing the weight structures and other functions, and GRAPHONE would be ready to serve a custom weight. All experiments are run on a machine with 2 Intel Xeon CPU E5-2683 sockets, each having 14 cores with hyper-threading enabled. It has 512GB memory, Samsung NVMe 950 Pro 512GB, and CentOS 7.2. Prior results have also been performed on the same machine.

We choose data ingestion, BFS, PageRank and 1-Hop query to simulate the various real-time usecases to demonstrate the impact of GRAPHONE on analytics. BFS and PageRank are selected because many real-time analytics are iterative in nature, e.g. shortest path, and many prior graph systems readily implement them for comparison. 1-Hop query accesses the edges of random 512 non-zero degree vertices and sums them up to make sure we access them all. 1-Hop query simulates many small query usecases, such as listing one's friends, or triangle completion to get friend suggestions in a social graph, etc. During the ingestion, vertex name to vertex ID conversion was not needed as we directly used the vertex ID supplied with these datasets as followed by other graph systems. All the edges will be stored twice in the adjacency list: in-edges and out-edges for directed graphs, and symmetric edges for undirected graphs. No compaction was running in any experiments unless mentioned.

Datasets. Table 3 lists the graph datasets. Twitter [3], Friendster [1] and Subdomain [4] are real-world graphs, while Kron-28 and Kron-21 are synthetic kronecker graphs generated using graph500 generator [25], all with 4 byte vertex size and without any weights. LANL network flow dataset [74] is a weighted graph where vertex and weight sizes are 4 bytes and 32 bytes respectively, and weight changes are treated as new streaming data. We run experiment on first 10 days of data. We test deletions on a weighted RMAT graph [15] generated with [56] where vertex and weight sizes are 8 bytes. It contains 4 million vertices, and 64 million edges, and a update file containing 40 million edges out of which 2,501,937 edges are for deletions.

7.1 Data Ingestion Performance

Logging and Archiving Rate. Logging to edge log is naturally faster, while archiving rate depends upon the archiving threshold. Table 3 lists the logging rate of a thread, and archiving rate at the archiving threshold of 2^{16} edges for our graph dataset. A thread can log close to 80 million edges per second, while archiving rate is only around 45 million edges second at the archiving threshold for most of the graphs. Both the rates are lower for LANL graph, as the weight size is 32 bytes, while others have no weights.

Ingestion Rate. It is defined as single threaded ingestion to the edge log at one edge at a time, and leaving the archive thread and durable phase to automatically change with the arrival rate. The number is reported when all the data are in the adjacency store, and persisted in the NVMe ext4 file. GRAPHONE achieves an ingestion rate of more than 45 million edges per second, except LANL graph. The ingestion rate is higher than archiving rate (at the archiving threshold) except in Kron-21, as edges more than the archiving threshold are archived in each epoch due to higher logging rate. This indicates that GRAPHONE can support a higher arrival rate as archiving rate can dynamically boost with increased arrival velocity. The Kron-21 graph is very small graph, and the thread communication cost affects the ingestion rate.

Compaction Rate. We run compaction as a separate benchmark after all the data has been ingested. The graph compaction rate is 345.53 million edges per second for the RMAT graph which has more than 2.5 million deleted edges out of total 104 million edges. Results for other graphs are shown in Table 3. The poor rate for LANL graph is due to long tail for compacting edge arrays of few vertices. As shown later in Fig. 12, the compaction improves the analytics performance where the static GRAPHONE serves compacted adjacency list as it had no link in its edge arrays.

Table 3: Graph datasets showing vertex and edge counts in **millions** (M), and different rates in **millions edges/s** (M/s). The results show that the ingestion rate would be upper and lower bounded by the logging and archiving rate. D = Directed, U = Undirected. For deletions see §7.2.

Graph Name	Type	Vertex Count (M)	Edge Count (M)	Individual Phases (M/s)		In-Memory Rate (M/s)		Ext-Memory Rate (M/s)		Compaction Rate (M)
				Logging	Archiving	Ingestion	Recovery	Ingestion	Recovery	
LANL	D	0.16	1,521.19	35.98	28.91	26.99	30.23	25.26	29.48	41.85
Twitter	D	52.58	1,963.26	82.62	47.98	66.39	71.28	61.13	71.87	541.71
Friendster	D	68.35	2,586.15	82.85	49.32	60.40	95.78	58.35	95.44	520.65
Subdomain	D	101.72	2,043.20	82.86	43.43	68.25	180.75	61.54	151.96	444.84
Kron-28	U	256	4,096	79.23	43.68	52.39	116.18	49.70	107.61	798.91
Kron-21	U	2	32	78.91	78.40	58.31	90.44	57.02	66.66	1011.68

Durability. The durable phase has less than 10% impact on the ingestion rate. Table 3 shows the in-memory ingestion rate and can be compared against that of GRAPHONE, which uses NVMe SSD for durability. This is because durable phase runs in a separate thread context, and exhibits only sequential write. The NVMe SSD can support up to 1500MB/s sequential write and that is sufficient for GRAPHONE as it only needs smaller write IO throughput, as shown in Fig. 9 for Friendster graph. This indicates that a higher logging rate can easily be supported by using a NVMe SSD.

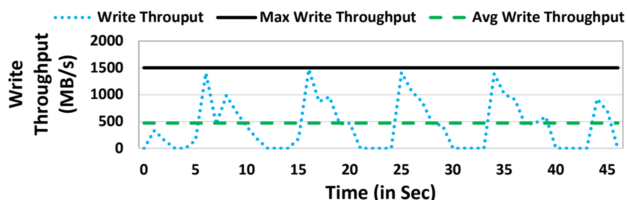


Fig. 9: Write throughput for friendster in GRAPHONE comparing against average requirement and maximum available in an NVMe

Recovery. Recovery only needs to perform archiving phase at bulk of data. As we will show later in Fig. 13, the archiving is fastest when around 2^{27}–2^{31} edges are cleaned together. Hence we take the minimum of this size as recovery threshold to minimize the memory requirement of IO buffer and the recovery time, and also gets an opportunity to pipeline the IO read time of the data with recovery. Table 3 shows the total recovery time, including data read from NVMe SSD after dropping the buffer cache. Clearly, GRAPHONE hides the IO time when compared against in-memory recovery. The recovery rate varies a lot for different graph due to different distribution of the batch of graph data that has profound impact on parallelism and hence locality access of edge arrays.

7.2 Graph Systems Performance

We choose different classes of graph systems to compare against GRAPHONE. Stinger is a dynamic graph system, Neo4j and SQLite are graph databases, and Galois and static version of GRAPHONE are static graph systems. Except stream computations, all the analytics in this section are performed on private static view containing no non-archived edges as it is created at the end of the ingestion.

Dynamic Graph System. Stinger is an in-memory graph system that uses atomic instructions to support fine-grained updates. So it cannot provide semantically correct analytics if updates and computations are scheduled at the same time, as different iteration of the analytics will run on the different versions of the data. We used the benchmark developed in [56] to compare the results on the RMAT graph.

Stinger is able to support 3.49 million updates/sec on the same weighted RMAT graph, whereas GRAPHONE ingests 18.67 million edges/sec, achieving 5.36× higher ingestion rate. Part of the reason for poor update rate of Stinger is that unlike GRAPHONE, it directly updates the adjacency store using atomic constructs. We have implemented PageRank and BFS in a similar approach as Stinger. The comparison is plotted in Fig. 10. Clearly, GRAPHONE is able to provide a better support for BFS and PageRank achieving 12.76× and 3.18× speedup respectively. The reason behind the speedup is explicit optimization to reduce the chaining which removes a lot of pointer chasing, and better cache access locality due to cacheline sized edge blocks.

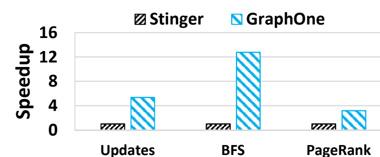


Fig. 10: Comparison against Stinger for in-memory setup

Databases. We compare against SQLite 3.7.15.2, a relational database, Neo4j 3.2.3, a graph database for ingestion test. SQLite and Neo4j support ACID transaction, and do not provide native support for graph analytics. It is known that higher update rate is possible by trading off the strict serializability of databases, however to measure the magnitude of improvement, it is necessary to conduct experiment.

The in-memory configuration of SQLite can ingest 12.46K edges per second, while GRAPHONE is able to support 18.67 million edges per second in the same configuration for above dataset. Neo4j could not finish the benchmark after more than 12 hours, which is along the same line as observed in [56]. Hence we have tested on a smaller graph with 32K vertices, 256K edges, and 100K updates. Neo4j is configured to use disk to make it durable. Neo4j and GRAPHONE both use the buffer cache while persisting the graph data. Neo4j can ingest only 14.81K edges per second, whereas GRAPHONE ingests at 3.63M edges per second.

Static Graph System. We compare against Galois, a representative in-memory static graph engine based on CSR format. It does not provide the data management capabil-

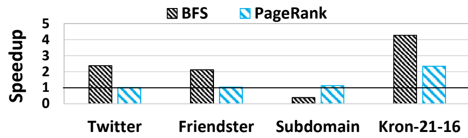


Fig. 11: Speedup comparison of GRAPHONE with Galois (pre-processing cost not included).

ity, so the whole graph is constructed in one time, called pre-processing time, which takes a significant amount of time [55]. In contrast, GRAPHONE can start the analytics without any pre-processing. Fig. 11 shows the speed up of GRAPHONE for PageRank and BFS over Galois (without pre-processing cost) for all the graphs except Kron-28 as Galois had a memory error. The PageRank results are almost same as it is compute intensive, thus effect of chaining is not observed. For Kron-21-16 which is very small, the performance of Galois is bad. We suspect that the cost of manual workload division in Galois for small graphs affects its performance, while we use dynamic scheduling of OpenMP.

For BFS, GRAPHONE performs better than Galois with an exception in the Subdomain graph. Both systems have same BFS implementation (direction-optimized BFS [11]) with a minor implementation difference. Our BFS is implemented using the status array metadata where the level of each vertex is maintained as one byte word, and tracking the active vertices requires revisiting whole status array. Galois uses the frontier queue where active vertices are kept in a separate work queue. Based on our experience with graph systems, status array implementation is faster for small diameter graphs, otherwise frontier queue approach is better. The Subdomain graph has 140 BFS levels (the highest of all graphs) hence we perform poorly, but Kron-21 has only 7 levels (the least of all the graphs) so the speedup is the highest.

Static GRAPHONE. GRAPHONE is expected to perform slightly worse than the static graph engine without including the pre-processing cost, but much better if including. Therefore to demonstrate the performance overhead of data management and chaining without any specific algorithm differences, we compare GRAPHONE against the static configuration of itself where maximum chain count is just one.

Fig. 12 shows this performance drop (without including pre-processing cost), specifically trading off just 17% average performance for real-world graphs (26% for all the graphs plotted) from the static system, one can support high arrival velocity of fine-grained updates. However, the performance drop is only temporary as the compaction process will remove the chaining in the background. Moreover, when adding the pre-processing cost to the static system, GRAPHONE is able to perform better. For example, the pre-processing cost for Kron-28 graph is 32.73s, one or multiple orders of magnitude longer than the runtime of these algorithms, e.g. 34.12× more than the run-time of BFS.

Stream Graph Engines. The logging and archiving operations are examples of different categories of stream analytics: logging is a proxy to continuous stateless stream analytics, while archiving is same to the discrete stateful stream analytics. Thus, Table 3 is also an indication of their performance. We have also implemented a streaming weakly connected components using ideas from COST [57] using stateless stream view APIs and it can process 33.60 million stream edges/s on Kron-28 graph.

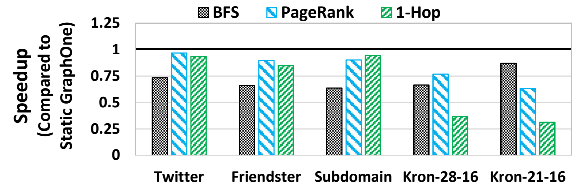


Fig. 12: Graph analytics performance in GRAPHONE compared to its static version that have no chaining requirement.

GRAPHONE runs stream computation and data ingestion concurrently, while prior stream processing systems interleave them that results into lower ingestion rate. To demonstrate the advantage of this design decision, we have implemented a streaming PageRank using stateful stream view APIs that runs in parallel to data ingestion in GRAPHONE. To simulate the prior stream processing we interleave the two. The execution shows that GRAPHONE improves the data ingestion by 26.22% for Kron-28 graph. We leave the comparison against other stream processing engine as future work as the focus of this work is on graph data-store.

7.3 System Design Parameters

Performance Trade-off in Hybrid Store. We first characterize the behavior of the hybrid store for different number of non-archived edges. Fig. 13 shows the performance variation of archiving rate, BFS, PageRank, and 1-hop query for Kron-28 graph when the non-archived edge counts are increased, while the rest of the edges are kept in the adjacency store for Kron-28. The figure shows that up to 2^{17} non-archived edges in the edge log brings negligible drop in the analytics performance. Hence, we recommend the value of archiving threshold as 2^{16} edges as the logging overlaps with the archiving. GRAPHONE is able to sustain an archiving rate 43.68 million edges per second at this threshold. The 1-Hop query latency of all 512 queries together is only 53.766 ms, i.e. 0.105 ms for each query.

The archiving threshold of 2^{16} edges is not unexpected as it is small compared to total edge count (2^{33}) in Kron-28, and

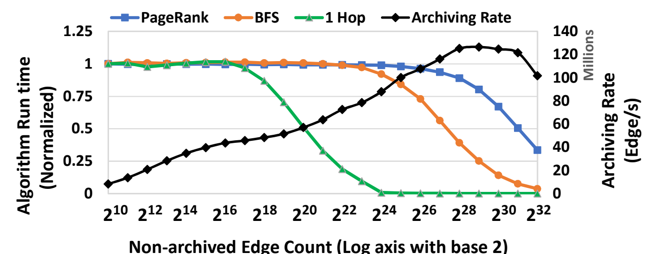


Fig. 13: Algorithmic performance and archiving rate variation for different non-archived edge count

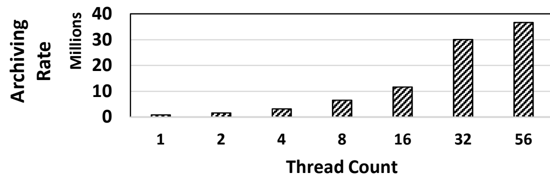


Fig. 14: Archiving rate scaling with thread count

the analytics on non-archived edges are parallelized. Further, the parallelization cost dominates when the number of non-archived edges are small (2^{10}). Thus the analytics cost drops only when the number of non-archived edges becomes large.

Fig. 13 also shows that higher archiving threshold leads to better archiving rate, e.g., a archiving threshold of 1,048,576 (2^{20}) edges can sustain a archiving rate of 56.99 million edges/second. The drawback is that the analytics performance will be reduced as it will find more number of non-archived edges. On the contrary, archiving works continuously and tries to minimize the number of non-archived edges, so a smaller arrival rate will lead to frequent archiving, and thus fewer non-archived edges will be observed at any time. The drop in archiving rate at the tail is due to the impact of large working set size that leads to more last-level-cache transactions and misses while filling the edge arrays.

Scalability. The edge sharding stage removes the need of atomic instruction or locks completely in the archiving phase, which results into better scaling of archiving rate with increasing number of threads as plotted in Fig. 14. There is some super-linear behavior when thread count is increased from 16 to 32. This is due to the second socket coming into picture with its own hardware caches, and non-atomic behavior makes it to scale super-linear. This observation is confirmed by running the archiving using 16 threads spread equally across two sockets, and achieves higher archiving rate compared to the case when the majority of threads belong to one socket.

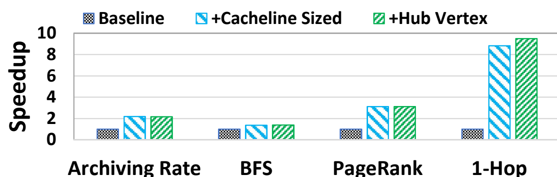


Fig. 15: Caheline sized memory allocation brings huge performance gain, while hub vertex handling on top of cacheline size memory allocation improves the query performance only.

Memory Allocation. Fig. 15 shows that the cacheline sized memory allocation and special handling of hub-vertices improve the performance of the archiving and analytics. The cacheline sized memory optimization improves the archiving rate at the archiving threshold by 2.20× for Kron-28 graph, while speeding up BFS, PageRank and 1-Hop query performance by 1.37×, 3.11× and 8.82×. Hub vertices handling additionally improves the query performance (by 7.5%).

Edge Log Size. Fig. 16 shows the effect of edge log size on overall ingestion rate on Kron-28 graph. Clearly, an edge log

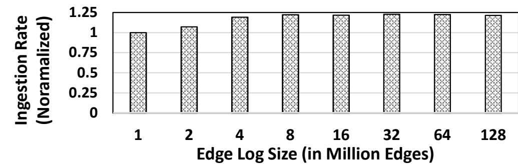


Fig. 16: Showing Ingestion rate when edge log size increases.

size greater than 4 million edges (32 MB) does not have any impact on overall ingestion rate.

8 Related Work

Static graph analytics systems [66, 50, 43, 60, 52, 79, 16, 36, 53, 65, 27, 44, 81, 28, 76, 7, 80] support only batch analytics where pre-processing consumes much more time than the computation itself [55]. Grapchi [47] and other snapshot based systems [35, 54, 62, 39, 26] support bulk updates only. Naiad [58], a timely dataflow framework, supports iterative and incremental compute but does not offer the data window on the graph data. Other stream analytics systems [17, 32, 58, 72] support stream processing and snapshot creation, some offering data window but all at bulk updates only. Stream databases [5, 6] provide only stream processing. TIDE [77] introduces probabilistic edge decay that samples data from base store.

Prior works [24, 69] follow integrated graph system model that manage online updates and queries in the database, and replicate data in an offline analytics engines for long running graph analytics tasks. As we have identified in §1, they suffers from excessive data duplication and weakest component problem. Zhang et al [78] also argue that such composite design is not optimal. GraPU [71] proposes to pre-processes the buffered updates instead of making them available to compute as in GraphOne. Trading-off granularity of data visibility is similar to Lazybase [19], but we additionally tune the access of non-archived edges to reduce performance drop in our setup and offer diverse data views.

The in-memory adjacency list in Neo4j [59] is optimized for read-only workloads, and new updates generally require invalidating and rebuilding those structures [63]. Titan [2], an open source graph analytics framework, is built on top of other storage engines such as HBase and BerkeleyDB, and thus does not offer adjacency list at the storage layer.

9 Conclusion

We have presented GraphOne, a unified graph data store abstraction that offers diverse data access at different granularity for various real-time analytics and queries at high-performance, while simultaneously supporting high arrival velocity of fine-grained updates.

Acknowledgments

The authors thank the USENIX FAST'19 reviewers and our shepherd Ashvin Goel for their suggestions. This work was supported in part by National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774.

References

[1] Friendster Network Dataset – KONECT. http://konect.uni-koblenz.de/networks/friendster.

[2] Titan Graph Database. https://github.com/thinkaurelius/titan.

[3] Twitter (MPI) Network Dataset – KONECT. http://konect.uni-koblenz.de/networks/twitter_mpi.

[4] Web Graphs. http://webdatacommons.org/hyperlinkgraph/2012-08/download.html.

[5] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The Design of the Borealis Stream Processing Engine. In *Cidr*, volume 5, pages 277–289, 2005.

[6] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):12039, 2007.

[7] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. Squeezing out All the Value of Loaded Data: An out-of-core Graph Processing System with Reduced Disk I/O. In *2017 USENIX Annual Technical Conference (USENIX ATC 17),(Santa Clara, CA)*, pages 125–137, 2017.

[8] L. Akoglu, H. Tong, and D. Koutra. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, May 2015.

[9] R. Albert, H. Jeong, and A.-L. Barabási. Internet: Diameter of the World-Wide Web. *Nature*, 401(6749):130–131, Sept. 1999.

[10] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM, 2011.

[11] S. Beamer, K. Asanovic, and D. Patterson. Direction-Optimizing Breadth-First Search. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[12] B. Bhattarai, H. Liu, and H. H. Huang. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, 2019.

[13] U. Brandes. A faster algorithm for betweenness centrality*. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[14] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[15] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SDM*, 2004.

[16] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[17] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In *Proceedings of the 7th ACM european conference on Computer Systems*, 2012.

[18] S. Choudhury, L. B. Holder, G. Chin, K. Agarwal, and J. Feo. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *18th International Conference on Extending Database Technology (EDBT)*, 2015.

[19] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey III, C. A. Soules, and A. Veitch. LazyBase: trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 169–182. ACM, 2012.

[20] D. Easley and J. Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[21] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High Performance Data Structure for Streaming Graphs. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5. IEEE, 2012.

[22] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-Law Relationships of the Internet Topology. In *ACM SIGCOMM computer communication review*, volume 29, pages 251–262. ACM, 1999.

[23] FlockDB. https://blog.twitter.com/engineering/en_us/a/2010/introducing-flockdb.html, 2010.

[24] Graph Compute with Neo4j. https://neo4j.com/blog/graph-compute-neo4j-algorithms-spark-extensions/, 2016.

[25] Graph500. http://www.graph500.org/.

[26] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: A Graph Engine for Temporal Graph Analysis. In *EuroSys*, 2014.

[27] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013.

[28] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for Easy and Efficient Graph Analysis. In *ACM SIGARCH Computer Architecture News*, 2012.

[29] Y. Hu, P. Kumar, G. Swope, and H. H. Huang. Trix: Triangle Counting at Extreme Scale. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

[30] Y. Hu, H. Liu, and H. H. Huang. TriCore: Parallel Triangle Counting on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 14. IEEE Press, 2018.

[31] B. A. Huberman and L. A. Adamic. Internet: Growth dynamics of the World-Wide Web. *Nature*, 1999.

[32] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. Time-Evolving Graph Processing at Scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 5. ACM, 2016.

[33] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási. The large-scale organization of metabolic networks. *Nature*, 2000.

[34] Y. Ji, H. Liu, and H. H. Huang. iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 58. IEEE Press, 2018.

[35] X. Ju, D. Williams, H. Jamjoom, and K. G. Shin. Version Traveler: Fast and Memory-Efficient Version Switching in Graph Processing Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 523–536. USENIX Association, 2016.

[36] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. GBASE: A Scalable and General Graph Management System. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2011.

[37] A. D. Kent. Comprehensive, Multi-Source Cyber-Security Events. Los Alamos National Laboratory, 2015.

[38] A. D. Kent, L. M. Liebrock, and J. C. Neil. Authentication graphs: Analyzing user behavior within an enterprise network. *Computers & Security*, 48:150–166, 2015.

[39] U. Khurana and A. Deshpande. Efficient Snapshot Retrieval over Historical Graph Data. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 997–1008. IEEE, 2013.

[40] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1675–1687. ACM, 2016.

[41] D. Knoke and S. Yang. *Social network analysis*, volume 154. Sage, 2008.

[42] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: a Distributed Messaging System for Log Processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[43] P. Kumar and H. H. Huang. G-Store: High-Performance Graph Store for Trillion-Edge Processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[44] P. Kumar and H. H. Huang. Falcon: Scaling IO Performance in Multi-SSD Volumes. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association*, pages 41–53, 2017.

[45] P. Kumar and H. H. Huang. SafeNVM: A Non-Volatile Memory Store with Thread-Level Page Protection. In *IEEE International Congress on Big Data (BigData Congress), 2017*, pages 65–72. IEEE, 2017.

[46] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[47] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-Scale Graph Computation on Just a PC. In *OSDI*, 2012.

[48] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187. ACM, 2005.

[49] H. Liu and H. H. Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[50] H. Liu and H. H. Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[51] H. Liu, H. H. Huang, and Y. Hu. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the SIGMOD International Conference on Management of Data*, 2016.

[52] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment (VLDB)*, 2012.

[53] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, 2017.

[54] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 363–374. IEEE, 2015.

[55] J. Malicevic, B. Lepers, and W. Zwaenepoel. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 631–643, Santa Clara, CA, 2017. USENIX Association.

[56] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader. A Performance Evaluation of Open Source Graph Databases. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, pages 11–18, New York, NY, USA, 2014. ACM.

[57] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[58] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[59] Neo4j Inc. https://neo4j.com/, 2016.

[60] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[61] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the Web. 1999.

[62] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On Querying Historical Evolving Graph Sequences. *Proceedings of the VLDB Endowment*, 4(11):726–737, 2011.

[63] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, 2013.

[64] D. M. Romero, B. Meeder, and J. Kleinberg. Differences in the mechanics of information diffusion across topics: idioms,

political hashtags, and complex contagion on twitter. In *Proceedings of the 20th international conference on World wide web*, pages 695–704. ACM, 2011.

[65] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *SOSP*. ACM, 2015.

[66] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric Graph Processing using Streaming Partitions. In *SOSP*. ACM, 2013.

[67] S. Sallinen, K. Iwabuchi, S. Poudel, M. Gokhale, M. Ripeanu, and R. Pearce. Graph Colouring as a Challenge Problem for Dynamic Graph Processing on Distributed Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 30. IEEE Press, 2016.

[68] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis. *Proceedings of the VLDB Endowment*, 6(14):1906–1917, 2013.

[69] M. Sevenich, S. Hong, O. van Rest, Z. Wu, J. Banerjee, and H. Chafi. Using Domain-specific Languages for Analytic Graph Databases. *Proc. VLDB Endow.*, 9(13):1257–1268, Sept. 2016.

[70] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the SIGMOD International Conference on Management of Data*, 2013.

[71] F. Sheng, Q. Cao, H. Cai, J. Yao, and C. Xie. GraPU: Accelerate Streaming Graph Analysis Through Preprocessing Buffered Updates. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, 2018.

[72] X. Shi, B. Cui, Y. Shao, and Y. Tong. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *Proceedings of the 2016 International Conference on Management of Data*, pages 417–430. ACM, 2016.

[73] J. Shun and G. 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 (PPoPP)*, 2013.

[74] M. J. M. Turcotte, A. D. Kent, and C. Hash. Unified Host and Network Data Set. *ArXiv e-prints*, Aug. 2017.

[75] K. Vora, R. Gupta, and G. Xu. Kickstarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 237–251. ACM, 2017.

[76] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. GRAM: Scaling Graph Computation to the Trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[77] W. Xie, Y. Tian, Y. Sismanis, A. Balmin, and P. J. Haas. Dynamic interaction graphs with probabilistic edge decay. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1143–1154. IEEE, 2015.

[78] Y. Zhang, R. Chen, and H. Chen. Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 614–630. ACM, 2017.

[79] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[80] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *OSDI*, pages 301–316, 2016.

[81] X. Zhu, W. Han, and W. Chen. GridGraph: Large-scale Graph Processing on a Single Machine Using 2-level Hierarchical Partitioning. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference*, 2015.

Automatic, Application-Aware I/O Forwarding Resource Allocation

Xu Ji[1,2], Bin Yang[2,3], Tianyu Zhang[2,3], Xiaosong Ma[4], Xiupeng Zhu[2,3], Xiyang Wang[3], Nosayba El-Sayed[*5], Jidong Zhai[1], Weiguo Liu[2,3], and Wei Xue[† 1,2]

[1]Tsinghua University, [2]National Supercomputing Center in Wuxi, [3]Shandong University, [4]Qatar Computing Research Institute, HBKU, [5]Emory University

Abstract

The I/O forwarding architecture is widely adopted on modern supercomputers, with a layer of intermediate nodes sitting between the many compute nodes and backend storage nodes. This allows compute nodes to run more efficiently and stably with a leaner OS, offloads I/O coordination and communication with backend from the compute nodes, maintains less concurrent connections to storage systems, and provides additional resources for effective caching, prefetching, write buffering, and I/O aggregation. However, with many existing machines, these forwarding nodes are assigned to serve a fixed set of compute nodes.

We explore an automatic mechanism, DFRA, for application-adaptive *dynamic forwarding resource allocation*. We use I/O monitoring data that proves affordable to acquire in real time and maintain for long-term history analysis. Upon each job's dispatch, DFRA conducts a history-based study to determine whether the job should be granted more forwarding resources or given dedicated forwarding nodes. Such customized I/O forwarding lets the small fraction of I/O-intensive applications achieve higher I/O performance and scalability, meanwhile effectively isolating disruptive I/O activities. We implemented, evaluated, and deployed DFRA on Sunway TaihuLight, the current No.3 supercomputer in the world. It improves applications' I/O performance by up to 18.9×, eliminates most of the inter-application I/O interference, and has saved over 200 million of core-hours during its test deployment on TaihuLight for 11 months. Finally, our proposed DFRA design is not platform-dependent, making it applicable to the management of existing and future I/O forwarding or burst buffer resources.

1 Introduction

Supercomputers today typically organize the many components of their storage infrastructure into a parallel and global controlled file system (PFS). Performance optimization by manipulating the many concurrent devices featuring different performance characteristics is a complicated yet critical task to administrators, application developers, and users. Moreover, it gets more challenging due to I/O contention and performance interference caused by concurrent jobs sharing the same PFS, bringing significant I/O performance fluctuation [28, 38, 40, 44, 61]. Meanwhile, different applications have vastly different I/O demands and behaviors, making it impossible for center administrators to decide one-size-for-all I/O configurations.

The task is even more difficult when it comes to the design and procurement of future systems. It is hard for machine owners to gauge the I/O demand from future users and design a "balanced" system with coordinated computation, network, and I/O resources. In particular, design and procurement typically happen years before any application could test run, while even decades-old programs usually see very different performance and scalability due to newer architecture/hardware/software on the more powerful incoming machine.

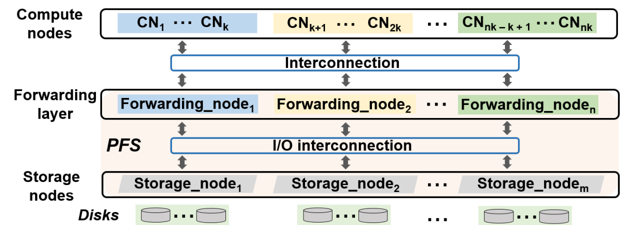


Figure 1: Typical I/O forwarding architecture for supercomputers

To give an example, consider the design of an *I/O forwarding infrastructure* [19], a widely adopted I/O subsystem organization that adds an extra forwarding layer between the compute nodes and storage nodes, as illustrated in Figure 1. This layer decouples file I/O from the compute nodes (CN_i in Fig 1), shipping those functions to the *forwarding nodes* instead, which are additional I/O nodes responsible for transferring I/O requests. It also enables compute nodes (1) to adopt a lightweight OS [48, 53, 64] that forwards file system calls to forwarding nodes, for higher and more consistent application performance [19], (2) to maintain fewer concurrent connections to the storage subsystem than having clients directly access file system servers, for better operational reliability, and (3) to facilitate the connection between two different network domains, typically set up with different topol-

[*]Most work conducted during appointment at Qatar Computing Research Institute.

[†]Wei Xue is the corresponding author. Email: xuewei@tsinghua.edu.cn

ogy and configurations, for computation and storage respectively. Finally, it provides an additional layer of prefetching/caching (or, more recently, burst buffer operations [51]), significantly improving user-perceived I/O performance and reducing backend data traffic.

Rank	Machine	Vendor	# C_node	# F_node	File system
3	Taihulight [11]	NRCPC	40,960	240	Lustre [24]
4	Tianhe-2A [69]	NUDT	16,000	256	Lustre + H2FS
5	Piz Daint [9]	Cray	6,751	54	Lustre + GPFS [57]
6	Trinity [17]	Cray	19,420	576	Lustre
9	Titan [15]	Cray	18,688	432	Lustre
10	Sequoia [10]	IBM	98,304	768	Lustre
12	Cori [2]	Cray	12,076	130	Lustre + GPFS
14	Oakforest-PACS [8]	Fujitsu	8,208	50	Lustre
18	K computer [5]	Fujitsu	82,944	5184	FEFS [56]

Table 1: I/O forwarding adopters among TOP20 machines (Nov 18)

Due to these advantages, I/O forwarding is quite popular, adopted by 9 out of the current TOP20 supercomputers (by the latest TOP500 list [16]). Table 1 summarizes their current TOP500 rankings and system configurations, including the number of compute and forwarding nodes. Note that recent Cray installations such as Cori and Trintiy use forwarding nodes with SSD-based burst buffers [3]. Forwarding architecture is also targeted in an Exascale storage design [45].

Despite the I/O forwarding layer's nature in decoupling compute nodes from backend storage nodes and enabling flexible I/O resource allocation, to provision a future system with forwarding resources (or to manage them for a current one) is challenging, as reasoned earlier. As a result, existing systems mostly adopt a *fixed forwarding-node mapping (FFM)* strategy between compute nodes and forwarding nodes, as illustrated in Figure 1. Though compute nodes are connected to all forwarding nodes, each forwarding node is assigned a fixed subset of k compute nodes to serve [48, 49, 63]. E.g., the compute-to-forwarding mapping is fixed at 512-1 at the No.3 supercomputer TaihuLight [30], and 380-1 at the No.5 Piz Daint [55].

This paper proposes a new method of forwarding resource provisioning. Rather than making fixed mapping decisions based on rough estimates, supercomputer owners could enable *dynamic forwarding resource allocation (DFRA)*, with flexible, application-aware compute-to-forwarding node mappings. We argue that DFRA not only alleviates center management's difficult hardware provisioning burdens, but significantly improves forwarding resource utilization *and* inter-application performance isolation.

DFRA is motivated by results of our whole-system I/O monitoring at a leading supercomputing center and extensive experiments. Specifically, we found the common practice of FFM problematic: (1) while the default allocation suffices on average in serving applications' I/O demands, the forwarding layer could easily become a performance bottleneck, leading to *poor application I/O performance and scalability* as well as *low backend resource utilization*; meanwhile the majority of forwarding nodes tend to stay under-utilized. (2) Forwarding nodes shared among relatively small jobs or partitions of large jobs become a contention point, where applications with conflicting I/O demands could inflict *severe performance interference* to each other. Section 2 provides a more detailed discussion of these issues.

Targeting these two major limitations of FFM, we devised a practical *forwarding-node scaling method*, which estimates the number of forwarding nodes needed by a certain job based on its I/O history records. We also performed an in-depth *inter-application interference study*, based on which we developed an interference detection mechanism to prevent contention-prone applications from sharing common forwarding nodes. Both approaches leverage automatic and online I/O subsystem monitoring and performance data analysis that require no user effort.

We implemented, evaluated, and deployed our proposed approach in the production environment of Sunway TaihuLight, currently the world's No.3 supercomputer. Deployment on such a large production system requires us to adopt practical and robust decision making and reduce software complexity when possible. In particular, we positioned DFRA as a "remapping" service, performed only when projected I/O time savings significantly offset the node-relinking overhead.

Since its deployment in Feb 2018, DFRA has been applied to ultra-scale I/O intensive applications on TaihuLight and has brought savings of bringing around 30 million core-hours per month, benefiting major users (who together consume over 97% of total core-hours). Our results show that our remapping can achieve up to 18.9× improvement to real, large-scale applications' I/O performance. Finally, though our development and evaluation are based on the TaihuLight supercomputer, the proposed dynamic forwarding resource allocation is not platform-specific and can be applied to other machines adopting I/O forwarding.

2 Background and Problems

Modern I/O forwarding architectures in HPC machines typically deploy a static mapping strategy [18] (referred to as FFM for the rest of the paper), with I/O requests from a compute node mapped to a fixed forwarding node. Here we demonstrate the problems associated with this approach, using the world's No.3 supercomputer TaihuLight as a sample platform. Specifically, we discuss *resource misallocation*, *inter-application interference*, and *forwarding node anomalies*, proceeded by introduction to the platform and the real-world applications to be discussed.

2.1 Overview of Platform and Applications

Platform Sunway TaihuLight is currently the world's No.3 supercomputer [30], with over 10M cores and 125-Petaflop peak performance. Its main storage system is a 10PB Lustre parallel file system [24], delivering 240GB/s and 220GB/s aggregating bandwidths for reads and writes respectively,

using 288 storage nodes and 144 Sugon DS800 disk arrays. Between its compute nodes and the Lustre backend is a globally-shared layer of 240 I/O forwarding nodes. Each forwarding node provides a bandwidth of 2.5GB/s and plays a dual role, both as a Lightweight File System (LWFS) [6] server to the compute nodes and a client to the Lustre backend. Before our DFRA deployment, 80 forwarding nodes were used for daily service, the other 160 reserved as backup or for large production runs with whole-system reservations.

In addition, TaihuLight has an online, end-to-end I/O monitoring system, Beacon [21]. It provides rich profiling information such as average application I/O bandwidth, I/O time and I/O access mode, as well as real-time system load and performance measurements across different layers of TaihuLight's storage system.

Applications Our test programs include 11 real-world applications and one parallel I/O benchmark. Six of them are 2017 and 2018 ACM Gordon Bell Prize contenders: CESM [37] (Community Earth System Model) is an earth simulation software system which consists of many climate models; CAM [59] is a standalone global atmospheric model deriving from the CESM project for climate simulation/projection; AWP [25] is a widely-used earthquake simulator [26,54]; Shentu [35] is an extreme-scale graph engine; LAMMPS [68] (Large-scale Atomic/Molecular Massively Parallel Simulator) is a popular molecular dynamics software; Macdrp [23] is a new earthquake simulation tool, specializing in accurate replay of earthquake scenarios with complex surface topography. CAM and AWP were among the three 2017 Gordon Bell Prize finalists (AWP being the final winner), while Shentu is in the 2018 finalist.

Note that although all 6 applications above can scale to the full TaihuLight system's 40,000+ compute nodes, full-scale production runs are conducted mostly with pre-arranged system-wide reservation. In most cases, we do not have such reservation or the largest-scale input datasets to evaluate their maximum-scale executions. However, throughout the year, their developers and users conducted many mid-size runs, each using hundreds or thousands of compute nodes. Most of our experiments evaluate at such scale, where I/O performance improvement can save shared I/O resources and reduce application execution time. Meanwhile, our findings here remain applicable to larger-scale runs.

The remaining large-scale applications in our testbed are: DNDC [32] (biogeochemistry application for agroecosystems simulation), WRF [1] (regional numerical weather prediction system), APT [67] (particle dynamics simulation code), XCFD (computational fluid dynamics simulator), and swDNN [29] (deep neural network engine). For the ease of controlling I/O behaviors and execution parameters, we also use MPI-IO [7], a widely-used MPI-IO benchmark by LANL.

These programs represent diverse data access behaviors regarding request characteristics, I/O volume, I/O library, and file sharing mode. Table 2 summarizes their I/O profiles. Here we roughly label each application as "high" or "low" in three dimensions: I/O throughput, IOPS, and metadata operation intensity, using empirical thresholds.[1]

App	Throughput	IOPS	Metadata	I/O Lib	I/O Mode
$\texttt{MPI-IO}_N$	High	Low	Low	MPI-IO	N-N
DNDC	Low	Low	High	POSIX	N-N
APT	Low	High	Low	HDF5	N-N
$\texttt{WRF}_1$	Low	Low	Low	NetCDF	1-1
$\texttt{WRF}_N$	High	High	Low	NetCDF	N-N
CAM	Low	Low	Low	NetCDF	1-1
AWP	Low	Low	Low	MPI-IO	N-1
Shentu	High	High	Low	POSIX	N-N
Macdrp	High	Low	Low	POSIX	N-N
LAMMPS	High	Low	Low	MPI-IO	N-N
XCFD	High	Low	Low	POSIX	N-N
CESM	High	Low	Low	NetCDF	N-N
swDNN	Low	Low	Low	HDF5	N-N

Table 2: Summary of test programs' I/O characteristics. "N-N" mode means N processes operate N separate files. "N-1" means N processes operate on one shared file. "1-1" means only one process among all processes operates on one file.

2.2 Motivation 1: Resource Misallocation

As shown above, applications have drastically different I/O demands, some requiring a much lower compute-to-forwarding nodes ratio than others. Traditional FFM does not account for varying I/O behaviors across applications, leading to significant resource misallocation. Below we discuss concrete sample scenarios.

Forwarding node under-provisioning The default I/O forwarding node allocation of one per 512 compute nodes in TaihuLight is adequate for the majority of applications we have profiled, but severely low for the most I/O intensive applications, where the forwarding nodes become an I/O performance bottleneck. Due to the transparent nature of the forwarding layer, such bottleneck is often obscure and hard to detect by application developers or users.

Figure 2 demonstrates the impact of allocating more forwarding nodes to two representative real-world applications: XCFD and $\texttt{WRF}_1$. We plot the I/O performance speedup (normalized to that under the default allocation of one forwarding node), as a function of the number of exclusive forwarding nodes assigned to the application.

We find that XCFD benefits significantly from increased forwarding nodes. XCFD adopts an N-N parallel I/O mode, where each MPI process accesses its own files. Thus many backend storage nodes and OSTs (Object Storage Targets, Lustre term for a single exported backend object storage volume) are involved in each I/O phase, especially when N is large. In general, applications with such I/O behavior suffer under FFM, due to the limited processing bandwidth in the assigned forwarding nodes.

[1] Calculated by $\alpha\times$ per-forwarding-node peak performance. In this paper we set α as 0.4, resulting in thresholds of 1GB/s for throughput, 10,000 for IOPS, and 200/s for metadata operation rate, respectively.

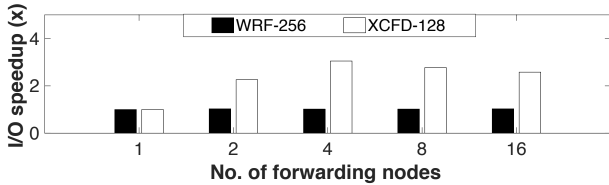


Figure 2: I/O performance speedup of $\mathtt{WRF}_1$ and XCFD with increasing dedicated forwarding node allocation. For the rest of the paper, the number after application name gives the number of compute nodes used in execution.

Our I/O profiling on TaihuLight indicates that among jobs using at least 32 compute nodes, around 9% use the N-N I/O mode, potentially seeing significant performance improvement given more forwarding nodes. Such under-provisioning was observed on other supercomputers, e.g., recent Cray systems where I/O requests issued by a single compute node can saturate a forwarding node [27].

Applications like $\mathtt{WRF}_1$, meanwhile, adopt the 1-1 I/O mode, where they aggregate reads/writes to a single file in each I/O phase. Intuitively, such applications do not benefit from higher forwarding node allocation. In addition, on TaihuLight applications with the 1-1 mode typically do not generate large I/O volumes in a single I/O phase, though they tend to run longer. Combining these two factors, 1-1 applications are mostly insensitive to additional forwarding layer resources beyond the default allocation.

Forwarding node load imbalance Application-oblivious forwarding resource allocation can lead to severe load imbalance across forwarding nodes. To verify this, we examined historical I/O traces collected on TaihuLight's forwarding nodes to check how they are occupied over time.

For every forwarding node, TaihuLight's profiling system records its per-second pass-through bandwidth. Analysis of such results first indicates that during the majority of profiled time intervals, the forwarding nodes are severely underutilized, echoing other studies' findings on overall low supercomputer I/O resource utilization [43, 47]. Meanwhile we

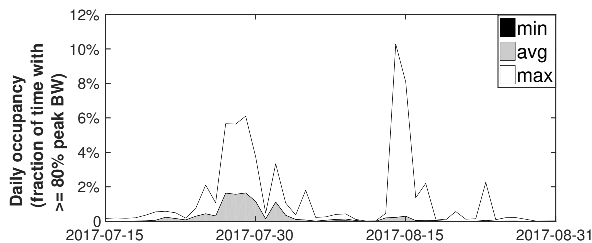


Figure 3: Sample TaihuLight forwarding layer load history

found high variability of loads across forwarding nodes and high day-to-day variances on forwarding node occupancy. We illustrate this with the forwarding nodes' *daily occupancy*, calculated as the fraction of 1-second windows in a day where a node's average bandwidth reaches 80% of the peak forwarding bandwidth of 2.5 GB/s. Figure 3 plots the minimum, average, and maximum daily occupancy across the 80 TaihuLight forwarding nodes, between July 15th and August 31st, 2017. We see both high variability in overall load (irregular average and maximum curves) and high load imbalance (large difference between the two).[2]

With recent and emerging systems adopting a burst buffer (BB) layer, such under-utilization and imbalance could bring wasted NVM spaces, buffer overflow, unnecessary data swapping, or imbalanced device wear.

2.3 Motivation 2: Inter-job Interference

I/O interference is a serious problem known to modern supercomputer users [28, 38, 40, 44, 61]. The common FFM practice not only neglects individual applications' I/O demands, but also creates an additional contention point by sharing forwarding nodes among concurrent jobs with conflicting I/O patterns. Figure 4 illustrates this using three real applications: AWP Shentu, and LAMMPS.All used the default 512-1 compute-to-forwarding mapping.

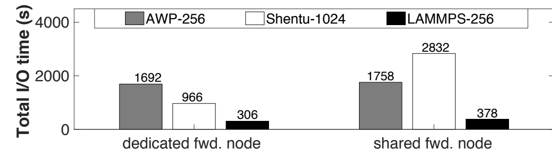


Figure 4: I/O performance impact of forwarding node sharing

We tested two execution modes, with each application allocated *dedicated* forwarding nodes vs. applications using *shared* ones. In both modes all three applications ran simultaneously. Note that for Shentu in the shared mode, it was allocated one dedicated forwarding node and two more nodes to share with other applications: one with AWP and one with LAMMPS, which were each running on 256 compute nodes (and thus allocated *half* of a forwarding node each).

As expected, all three experienced faster I/O with dedicated forwarding nodes. However, some suffered much higher performance interference. While AWP and LAMMPS saw mild slowdowns (4% and 23% increase in total I/O time), Shentu had a 3× increase. This is due to the highly disruptive behavior of AWP's N-1 I/O mode (discussed in more details later), causing severe slowdown of Shentu processes accessing the same forwarding node. Given the synchronous nature of many parallel programs, their barrier-style parallel I/O operations wait for all processes involved to finish. Thus slowdown from the "problem forwarding node" shared with AWP is propagated to the entire application, despite that it had one dedicated forwarding node and shared the final one with a much more friendly LAMMPS.

In Section 5, we present an in-depth inter-application interference study, based on which we perform application-aware interference estimation to avoid sharing forwarding nodes among applications prone to interference.

[2]Our recent paper on TaihuLight's Beacon monitoring system gives more details on workload characteristics [21].

2.4 Motivation 3: Forwarding Node Anomaly

Finally, when certain forwarding nodes show abnormal behavior due to software or hardware faults, applications assigned to work through these *slow* nodes under FFM would suffer. We found TaihuLight forwarding nodes prone to correctable failures in memory or network, confirming the "fail-slow" phenomenon observed at data centers [33].

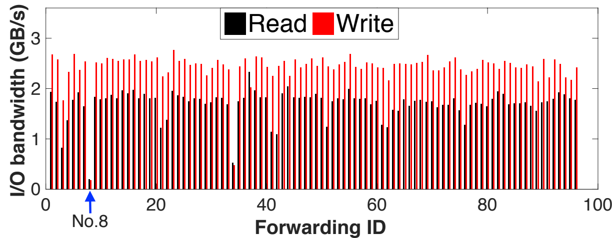


Figure 5: Forwarding node peak performance. Forwarding nodes with IDs 3, 8 and 34 show abnormal performance.

Figure 5 shows sample benchmarking results measuring read/write bandwidth across 96 currently active forwarding nodes, conducted during system maintenance. While most forwarding nodes do report consistent bandwidth levels (with expected variability due to network contention, disk status, etc.), a small number of them clearly exhibit performance anomalies. In particular, forwarding node No.8 (highlighted with arrow) is an obvious outlier, with average read and write bandwidth at 7% and 12% of peak, respectively.

Fortunately, the I/O monitoring system in TaihuLight performs routine, automatic node anomaly detection across all layers of the I/O infrastructure. As shown in Section 3, our proposed dynamic forwarding system leverages such anomaly detection to skip nodes experiencing anomalous behavior in its dynamic allocation.

3 System Overview

Given the above multi-faceted problems caused by FFM, we propose a practical-use and efficient dynamic forwarding resource allocation mechanism, DFRA. DFRA works by *remapping* a group of compute nodes (scheduled to soon start executing an application) to other than their default forwarding node assignments, whenever the remapping is expected to produce significant application I/O time savings. It serves three specific purposes: (1) to perform application-aware forwarding node allocation to avoid resource under-provisioning for I/O-intensive jobs, (2) to mitigate inter-application performance interference at the forwarding layer, and (3) to (temporarily) exclude forwarding nodes identified as having performance anomalies.

To remap at the job granularity does not pose much technical difficulty by itself. The challenge lies in developing an automatic workflow that examines both the application's I/O demands and real-time system status, and performs effective inter-application I/O interference estimation, while remaining as transparent as possible to users and relieving administrators from labor-intensive manual optimizations.

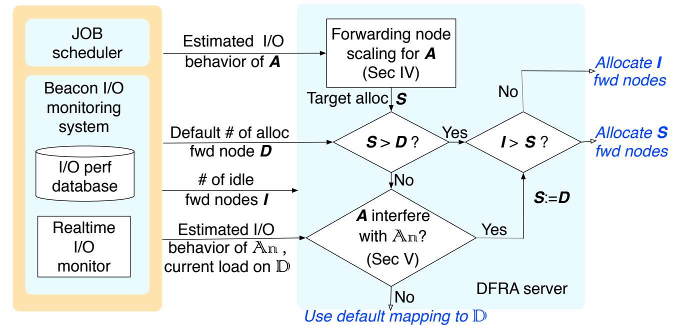


Figure 6: DFRA decision-making workflow

To this end, we leverage the Beacon I/O monitoring system on TaihuLight to perform continuous application I/O profiling and learn the I/O characteristics of applications from history.[3] Assisted with all-layer profiling data, from the compute nodes to the backend OSTs, *plus* per-job scheduling history that reveals the mapping of a job's processes to compute nodes, we obtain a detailed understanding of each past job's I/O behavior, including peak bandwidth per compute node, request type/size distribution, periodic I/O frequency, I/O mode (N-N, N-1, 1-1, etc.), and metadata access intensity. Given that HPC platforms typically see applications run repeatedly, with very similar I/O patterns [62], there is high likelihood that the past reflects the future.

We have designed, implemented, and deployed a dynamic forwarding resource allocation mechanism on TaihuLight, as depicted in Figure 6 It determines whether a *target job* A, scheduled to begin execution on a certain set of compute nodes, needs to have forwarding nodes remapped and if so, to which nodes. Implementation-wise, such proposed dynamic forwarding resource allocation component resides on a single dedicated server (DFRA server). It interacts with the Beacon I/O monitoring system and the job scheduler. Beacon provides an I/O performance database to query using A's job information (e.g., application name, project name, user name, and execution scale) and estimates its I/O characteristics based on historical records.

The job's expected I/O features, such as I/O mode and the number of compute nodes performing I/O, are then fed to the DFRA server. First it checks whether this application needs to *scale out* to use more forwarding nodes. If not (the more likely case), then we incorporate real-time scheduling information to know about the "neighbor applications" $\mathbb{A}_n$ (the set of applications currently running on $\mathbb{D}$, the forwarding nodes to be assigned under the default allocation). This allows the DFRA server to check whether the default mapping will produce significant *performance interference* with neighbors already running there. If significant interference is expected, we keep the default allocation size D, but would

[3]Partial I/O traces released at https://github.com/Beaconsys/Beacon

remap the compute nodes to dedicated forwarding nodes.[4]

If scaling *is* required, we first calculate S, the number of forwarding nodes needed. We then check I, the number of *idle* forwarding nodes currently available, excluding those undergoing performance anomaly, and allocate the fewer between S and I. Though more sophisticated "partial-node" allocation is possible, we choose the more simple scheme considering the overall forwarding node under-utilization.

In summary, there are two types of "upgrades": to grant more forwarding nodes (for capacity) or grant unused forwarding nodes (for isolation). In both cases, as we only allocate *dedicated nodes* from the idle pool, no interference check is further needed. In the specific case of TaihuLight, at the beginning of this research, beside the 80 forwarding nodes using the default 512-1 mapping, more than 100 are reserved for backup or manual allocation. For systems without such over-provisioning, we recommend the default allocation be lowered to serve the majority of jobs, who are not I/O-intensive, and have a set of "spare" forwarding nodes for ad-hoc remapping.

Note that this is a best-effort system transparent to users. Additionally, for the majority of applications, who are not I/O-intensive enough to warrant higher allocation and not significantly interference-prone with expected neighbors, the decision is to remain with default mapping.

The actual remapping process is conducted upon the jobs' dispatch and involves making RPCs from the DFRA server to the compute nodes concerned, instructing them to drop off the original connection and connect to newly assigned forwarding nodes, allocated from the current available forwarding node pool. Considering that a job tends to have consistent I/O behavior, this remapping is done once per job execution, rather than per request. If remapped, when A completes, its compute nodes will be reset to default mapping, making DFRA maintenance simple and robust.

4 Automatic Forwarding Node Scaling

To decide on the "upgrade eligibility" of a job, we estimate its multiple I/O behavior metrics based on the query results of I/O monitoring database. When historical information is not sufficient, e.g., as in the case of new applications, our system does *not* change the default mapping. I/O monitoring data collected from these runs will help forwarding resource allocation in future executions.

Our scaling decision-making adopts a per-job forwarding node allocation algorithm. It considers both the application-specific I/O workload characteristics and historical performance data of forwarding node load levels while serving this application. Most of the threshold values are set empirically according to our extensive benchmarking of the system, and can be further adjusted based on continuous I/O performance monitoring. More specifically, the target job A needs to meet the following criteria to be *eligible for a higher forwarding resource allocation* than the default setting:

1. its total I/O volume is over V_{min} during its previous execution;
2. it has at least N_{min} compute nodes performing I/O; and
3. it is *not* considered metadata-operation-bound, i.e., its past average number of metadata operations waiting at a forwarding node's queues is under $W_{metadata}$.

The rationale is based on the primary reason for a job to have an upgraded allocation: it possesses enough I/O parallelism to benefit from more forwarding resources. For such benefit to offset the forwarding node remapping overhead, first the application needs to generate a minimum amount of I/O traffic. Applications diagnosed as metadata-operation-heavy, regardless of their total I/O volume or I/O parallelism, are found to not benefit from more forwarding nodes as their bottleneck is the metadata server (MDS).

If A passes this test, with past history showing that it is expected to use N_A of its compute nodes to perform I/O, the number of its forwarding node allocation S is calculated as $\lceil N_A/F \rceil$. Here F is a *scaling factor* that reflects typically how many I/O-intensive compute nodes can be handled by a forwarding node without reaching its performance cap. In our implementation, F is set as $\lceil B_f/B_c \rceil$, where B_f and B_c are the peak I/O bandwidths of a single forwarding and compute node, respectively. If not enough idle forwarding nodes are available, we allocate all the available nodes. We expect this case to be extremely rare, as given the typical system load, there are enough idle forwarding nodes to satisfy all allocation upgrades.

In our deployment on TaihuLight, we empirically set V_{min} at 20 GB, N_{min} at the F value (32 based on the formula above), and $W_{metadata}$ at twice the per-forwarding-node thread pool size, also 32. These can be easily adjusted based on machine specifications and desired aggressiveness.

We do *not* downgrade allocations for the metadata-heavy or 1-1 I/O mode jobs, considering their baseline per-process I/O activities (such as executable loading, logging, and standard output). Also considering TaihuLight's sufficient backup forwarding nodes, we opt not to pay the remapping overhead for downgrading allocations in this deployment, though downgrading is easy to implement when needed.

Figure 7 shows how application I/O performance, in aggregate I/O bandwidth measured from the application side, changes with different compute-to-forwarding node ratios. As these tests used dedicated forwarding nodes, we started from the 256-1 allocation, rather than the default 512-1.

Here several applications, namely `APT`, `DNDC`, `WRF`$_1$, and `CAM`, due to insufficient I/O parallelism or being metadata-heavy, do not pass the eligibility test. Their I/O performance results confirm that they would have received very little performance improvement with more forwarding nodes been allocated. The other applications, however, see substantial I/O

[4] This does not consider the jobs' duration, as history records or job script specified run times are not reliable indicators. Such conservative strategy is allowed by the typical abundance of idle forwarding nodes.

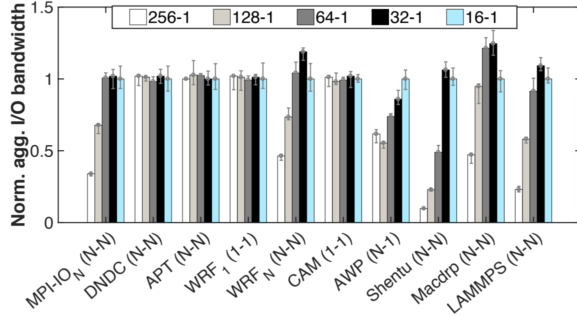


Figure 7: Aggregate I/O bandwidth under increasing forwarding node allocation (compute-to-forwarding mapping ratio), normalized to the 16-1 case. All applications ran using 256 compute nodes (1024 MPI processes). Each test was repeated 5 times, with average results plotted and error bars giving 95% confidence intervals.

bandwidth enhancement with increased forwarding node allocations, by up to a factor of 10.9×. Judging from results across all such applications, our current F setting of 32 deliver best aggregate I/O bandwidth in most cases.

5 Interference Analysis and Avoidance

Our DFRA system attempts to mitigate this performance interference by assigning jobs that are expected to interfere to *different* forwarding nodes. Note that prior work on interference detection and optimization focused mostly on deriving offline, analytical interference models (e.g., [28,31]). In contrast, our work focuses on designing practical online interference estimation techniques that DFRA can use effectively.

5.1 Inter-application Interference Analysis

We first conduct a rather controlled study, to observe I/O interference behavior between pairs of representative I/O applications. From the applications described in Table 2, we select eight that cover different I/O resource-consumption patterns. Next, we perform pairwise co-execution among these selected applications. For this, we use 256 compute nodes (1024 MPI processes) each, so that the paired workloads have equal execution scale. Under the default allocation, the 512 compute nodes running the two programs hence share one forwarding node.

To gauge interference, we measure each application's *I/O slowdown* by calculating the application's relative slowdown factor in overall I/O performance (the time spent in the I/O interference interval) from that of its solo run. Table 3 shows the pairwise results, with high-interference pairs (with either slowdown factor >3) marked in **bold**, and medium-interference ones (those among the rest with either slowdown factor >2) marked with "*".

The majority of our applications in this study are intensive in at least one dimension of I/O resource usage and are expected to see I/O performance slowdown when they share the same I/O path. Results in Table 3 confirm this. An application exhibits an I/O slowdown of around 2× when co-running with itself (another instance of the same application), due to the expected resource contention. The remaining pairwise slowdown results reveal several interesting interference behaviors.

First, we find that applications with *low* demands in all three dimensions (throughput, IOPS, and metadata operation rate) do not introduce or suffer significant I/O slowdown when co-running with other applications, with the exception of applications using the N-1 I/O mode (recall Table 2).

To understand the reasons behind, we conducted follow-up investigations. The three applications that fall into the "Low/Low/Low" category are `WRF`$_1$, `CAM`, and `AWP`. Among them, `AWP` turns out to be a highly disruptive workload, causing high degrees of I/O slowdown to whoever runs with it. We performed additional experiments, including `MPI-IO` tests emulating its behavior with different I/O parameters, and identified the problem being its N-1 file sharing mode. While N-1 writes have been notoriously slow (such as with Lustre [20], also verified by our own benchmarking), our study reveals that it brings high disturbance (average of 38.4× to other applications tested).

Further examination of profiling results identified the forwarding layer as the source of interference. Each forwarding node maintains a fixed thread pool, processing client requests from the compute nodes it is in charge of. While designed to allow parallel handling of concurrent client requests, applications using the N-1 file sharing mode generate a large number of requests and flood the thread pool. Their occupation of the forwarding layer thread resources is further prolonged by the slow Lustre backend processing of such I/O requests (often involving synchronization via locks). The result is that other concurrent applications, whose I/O requests might be far fewer and more efficient, are blocked waiting for thread resources, while the I/O system remains under-utilized.

Such effect is highlighted by follow-up test results in Figure 8. We pair 2 benchmarks, `MPI-IO`$_1$ (N-1) and `MPI-IO`$_N$ (N-N), running at different scales. The bars (left y axis) show the queue lengths of pending requests at the forwarding layer. While the queue length increases proportionally to the number of compute processes, as expected, the "co-run" queue length of `MPI-IO`$_N$ does not grow significantly from its solo run. The much greater increase in `MPI-IO`$_N$ latency (red curves using the right y axis), meanwhile, comes from the slowdown of each `MPI-IO`$_1$ request.

Secondly, we observe from Table 3 that `DNDC` introduced significant slowdown to all other workloads (by a factor from 2.4× to 33.3×). A closer look finds that `DNDC` is the only application in our testbed with significant metadata access intensity. `DNDC`'s production runs are not particularly large (only using 2048 processes), which simultaneously read 64,000 small files (up to several KBs each). The large number of open/close requests pile up and block requests from other applications obviously.

More profiling reveals that read requests see much faster

Apps	MPI-IO$_N$	APT	DNDC	WRF$_1$	WRF$_N$	Shentu	CAM	AWP
MPI-IO$_N$	*(2.1, 2.1)	**(1.1, 9.3)**	**(4.8, 1.1)**	(1.0, 1.0)	*(2.1, 2.0)	**(1.3, 4.5)**	(1.0, 1.0)	**(3.3, 1.1)**
APT	-	*(2.0, 2.1)	**(33.3, 1.0)**	(1.0, 1.0)	**(4.3, 1.4)**	**(6.3, 1.3)**	(1.0, 1.0)	**(50.0, 1.1)**
DNDC	-	-	*(2.0, 2.0)	**(1.0, 25.0)**	**(1.0, 11.1)**	**(1.1, 16.7)**	**(1.0, 33.3)**	*(2.2, 2.4)
WRF$_1$	-	-	-	(1.0, 1.0)	(1.0, 1.0)	(1.0, 1.0)	(1.0, 1.0)	**(50.0, 1.0)**
WRF$_N$	-	-	-	-	*(2.1, 2.1)	*(2.0, 2.3)	(1.0, 1.0)	**(12.5, 1.3)**
Shentu	-	-	-	-	-	*(2.0, 2.0)	(1.0, 1.0)	**(12.5, 1.1)**
CAM	-	-	-	-	-	-	(1.0, 1.0)	**(100.0, 1.0)**
AWP	-	-	-	-	-	-	-	*(2.0, 2.0)

Table 3: I/O slowdown factor pairs of applications listed in row and column headers. *E.g.*, in the 1st row, 2nd column, MPI-IO$_N$ has slowdown of 1.1 and APT has 9.3 when they co-execute. (**Bold** and "*" indicate high- and medium-interference, respectively)

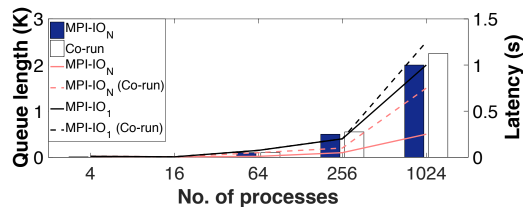


Figure 8: Interference between MPI-IO$_1$ and MPI-IO$_N$. Bar represents queue length and line represents latency.

processing than open, with only slight increase in processing time when DNDC joins a read-heavy job, indicating bottleneck-free Lustre handling. The wait time, however, sees almost 4× increase for read and around 2× for open operations. Besides that the forwarding node thread pool being the point of contention, the asymmetric delay prompted us to examine its scheduling policy. We found that metadata requests were given higher priority over normal file I/O, favoring interactive file system user experience. This, combined with their longer processing time, makes metadata-heavy applications like DNDC unsuspected disruptive workloads. While our ongoing work targets more adaptive policies, for DFRA we specifically check jobs' metadata operation intensity for interference estimate.

Finally, we find that even applications with seemingly *orthogonal* resource usage patterns may not get along well, with *asymmetric* performance impact on each other. In particular, we find that high-bandwidth, low-IOPS applications impact the performance of low-bandwidth, high-IOPS ones (but not vice versa). This can be seen from the APT-MPI-IO$_N$ results in Table 3, with the high-IOPS APT suffering an almost 10× slowdown while the high-bandwidth MPI-IO$_N$ is hardly impacted. A closer look reveals that APT reaches IOPS of over 80,000, with requests sized under 1KB. The reason behind the asymmetric slowdown is then intuitive: high-bandwidth applications likely perform sequential I/O with large request sizes, which force the many small requests from the high-IOPS applications to wait long.

In summary, we discover that I/O interference not only comes from bandwidth-intensive applications and problematic access patterns (as assumed by previous studies [31,44]), but also from applications issuing inefficient I/O requests, while *simultaneously incurring high contention and low utilization*, such as in the metadata-heavy and high-IOPS cases.

5.2 Inter-job Interference Estimate for DFRA

We now discuss DFRA's inter-application interference check, introduced in Section 5.1. Recall that it is needed only when we decide that the target job A, which is to be scheduled, does not need more forwarding nodes than granted by the default mapping. The interference check is then performed pairwise, between A and each member of its neighbor application set $\mathbb{A}_n$.

As actual I/O interferences incurred during co-executions of applications depend on other factors such as their I/O phases' frequency and interleaving, we use our interference analysis results to make conservative, qualitative decisions. More specifically, for A and each of its neighbor in $\mathbb{A}_n$, we consider interference is likely if either A or the neighbor is:

1. using the N-1 I/O mode, *or*
2. considered "metadata operation heavy" (average number of metadata operations waiting at a forwarding nodes queue $> W_{metadata}$), *or*
3. considered "high-bandwidth" or "high-IOPS" (using criteria described in Section 2.1).

For A, the above check has to be based on our monitoring system's per-application I/O performance history data. For jobs in $\mathbb{A}_n$, however, our history-based I/O behavior inference can and should be complemented with real-time I/O behavior analysis. In particular, as the inferred I/O behavior includes pattern information such as I/O phase frequency, I/O volume per process performing I/O, and I/O mode, such estimates can be verified by actual data collected during the neighbors' current execution. E.g., if a forwarding node is receiving unexpectedly low I/O load from an application running, DFRA considers the application turns off I/O for this run, overriding its positive interference estimate. Similarly, if an application is issuing I/O at intensity not indicated by its past history, we play safe and use the peak load level measured during its execution so far on the forwarding node(s) involved, to determine whether interference is likely.

6 Evaluation

6.1 Job Statistics from I/O History Analysis

First, DFRA's working relies on applications' overall consistency in I/O behavior. We verified this with the 18-month TaihuLight I/O profiling results, confirming observations by

existing studies [31, 44]. Specifically, if we simply forecast a new job's I/O mode and volume as those in its latest run using the same number of compute nodes, we can successfully predict these parameters with under 20% deviation for 96,621 jobs (90.3%) out of 107,001 in total.

Category	Count	Count(%)	Core-hour(%)
Total jobs	107,001	100%	100%
Job benefits from DFRA	14,712	13.7%	79.0%
Job's I/O volume $< V_{min}$ (20 GB)	83,562	78.1%	18.9%
Job's I/O nodes $< N_{min}$ (32)	8,727	8.2%	2.1%
Job's metadata queue length $\geqslant W_{metadata}$ (32)	0	0.0%	0.0%

Table 4: DFRA eligibility screening results, based on using per-job I/O history between April 2017 and August 2018

We then give statistics about DFRA's decisions and its potential beneficiaries, by running these 18-month job I/O profiles through DFRA's scaling decision making. For jobs that were refused allocation upgrades, we categorize them by the first test failed during the DFRA allocation scaling eligibility check (Section 4). Table 4 lists the results.

First, 13.7% jobs (minority in count yet accounting for 79.0% of core-hours) are granted upgrades and expected to benefit from DFRA. This demonstrates that though the I/O system is overall underutilized, there are substantial amount of I/O-intensive jobs as potential beneficiaries. Among the rest, most fail to meet the total I/O volume threshold V_{min}, followed by the number of I/O nodes involved. No job fails at the metadata-intensity check, as such applications in this particular job history do not pass the I/O volume test.

Also, throughout this history "replay" using DFRA, the average forwarding node consumption is 171.2, suggesting that DFRA can get much better I/O performance while working well under the total 240-node forwarding capacity.

6.2 Performance/Consistency Improvement

Next we examine the impact of DFRA's deployment on real applications' I/O performance in the TaihuLight production environment. We run the 11 applications (introduced in Section 2.1) each for 10 times at randomly selected times during a 1-month period, each time under DFRA and FFM within the same job execution, with remapping done in between. To control total resource usage, `Shentu`, `LAMMPS`, and `Macdrp` run with 1,024 compute nodes, with the other applications run at their typical mid-size run scale (`swDNN` using 512 nodes while the rest using 256). They are further divided into two groups: *scaling*, with more forwarding nodes granted by DFRA, and *non-scaling*, with dedicated forwarding node allocation if deemed interference-prone by DFRA (which may depend on their neighbor jobs under the default mapping, though `APT` and `DNDC` are always isolated).

DFRA brings an average I/O speedup of 3.5× across all 11 applications, from 1.03×(`CAM`) to 18.9×(`Shentu`). As expected, applications in the scaling group receive higher speedup (average at 4.8× and up to 18.9×), while non-scaling applications benefit more from reduced performance variability (and potential slowdown incurred on their neighbors). However, the scaling group also obtains dramatic improvement in *I/O performance consistency*, with average reduction of 91.1% in range of I/O times.

The reason lies in the "mis-alignment" of compute nodes to forwarding nodes using FFM. Our job history finds over 99% of large-scale jobs (using 512 compute nodes or more) assigned to share forwarding nodes with other jobs, though their job scales are often perfect multiples of the default factor of 512. Intuitively, such fragmentation often also leads to dramatic load imbalance across forwarding nodes (partially) serving the same I/O-intensive application.

App	Comp. time	I/O time w. FFM	I/O time w. DFRA	Total time reduction
`Shentu`-1024	1,303s	1,204s	64s	45%
`LAMMPS`-1024	3,510s	431s	97s	8%
`Macdrp`-1024	6,932s	260s	105s	2%
`swDNN`-512	0s	132,710s	31,476s	76%
`AWP`-256	2,301s	255s	204s	2%
`CESM`-256	4,742s	942s	846s	2%
$\mathtt{WRF}_N$-256	1,640s	135s	89s	3%
`DNDC`-256	992s	222s	216s	0.5%
$\mathtt{WRF}_1$-256	1,640s	513s	479s	2%
`APT`-256	222s	46s	24s	8%
`CAM`-256	3,226s	899s	876s	0.6%

Table 5: Per-phase computation and I/O time of applications

Table 5 describes the impact of DFRA on resource-intensive applications' overall performance. All applications but one (`swDNN`) have clear repeated phases alternating between computation and I/O, while the number of such computation-I/O cycles may vary across runs according to users' needs. Therefore we illustrate the relative impact by listing the more stable per-cycle average computation time, average I/O time (with FFM and DFRA respectively), and the percentage of total time saving by DFRA. The last column does not change when a particular production run adjusts the number of computation-I/O cycles. `swDNN`, unlike timestep numerical simulations, is a parallel deep learning model training application that has fine-grained, interleaving computation and I/O, therefore we treat its execution as a single I/O phase.

As most applications conform to their "total I/O budget" by adjusting their I/O frequency, by taking one snapshot every k computation timesteps, DFRA is not expected to yield significant overall runtime reduction, especially with the non-scaling ones. However, it does bring impressive total time savings for I/O-bound applications `Shentu` (45%) and `swDNN` (76%), as well as over 8% savings for `APT` and `LAMMPS`. Meanwhile, making I/O faster also implies that the applications could afford to output more frequently under the same I/O budget.

Figure 10 illustrates this scenario using a `Shentu` test run using 1024 compute nodes, which are not allocated contiguously. Under DFRA, its 32 dedicated forwarding nodes serve equal partitions of compute nodes, as each compute node can be individually remapped to any forwarding node, allowing almost all compute nodes finishing I/O simultaneously. Under FFM, instead, these dispersed 1024 compute nodes are

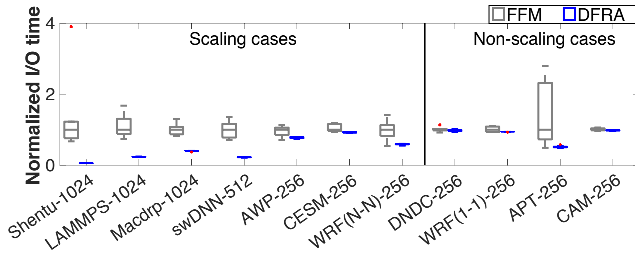


Figure 9: Impact of DFRA on application I/O performance, with results normalized to median I/O time under FFM

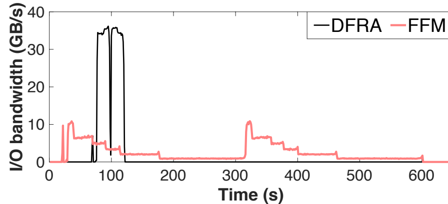
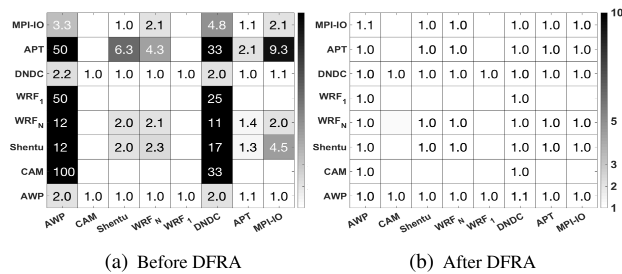


Figure 10: Sample `Shentu`-1024 I/O bandwidth timelines

mapped to 7 forwarding nodes. As a result, the same I/O activities take much longer, with multiple stair-steps produced by completion of different forwarding nodes.



(a) Before DFRA (b) After DFRA

Figure 11: Impact of DFRA's interference avoidance on pairwise application co-run slowdown. Darkness of each block reflects the slowdown factor of the application at row header by the application at the column header. Blocks with slowdown factor values give co-running pairs with interference anticipated by DFRA and hence allocated separate forwarding resources. In all experiments, the compute-to-forwarding mapping uses the default setting (512-1), with DFRA allocating dedicated forwarding nodes to application pairs it considers interference-prone.

To evaluate our proposed interference avoidance, we rerun the pairwise experiments (see Section 5) with DFRA on TaihuLight. Results are in Figure 11. We found DFRA can detect potential interference with pairs having slowdown factors over 1.1 at either side. In this test, we only separate these applications, without scaling up forwarding nodes, to isolate the benefits brought by interference avoidance.

Compared with the left plot, where just by sharing a forwarding node, certain applications could perceive a 2× to 100× I/O slowdown, the right plot reduces such slowdown to uniformly under 1.1×. With many jobs on TaihuLight sharing forwarding nodes, DFRA removes the infrequent (yet highly damaging) inter-application interference cases.

Finally, we evaluate an alternative approach, *RR*, which maps compute nodes to forwarding nodes in a round-robin manner. We test RR 32-1, where each group of contiguous 32 compute nodes are assigned to one forwarding node. Figure 12 gives the speedup (again over the 512-1 fixed allocation) of running one of the 5 applications given at the x-axis simultaneously with either `DNDC` or `AWP`. Each application runs on 256 compute nodes, with two co-running applications sharing 8 forwarding nodes using RR. For fair comparison, DFRA uses 64-1 allocation here, so that all co-run experiments enlist 8 forwarding nodes in total. RR spreads the load of each application to all 8 forwarding nodes, but does not offer the performance isolation brought by DFRA, when two applications running on disjoint compute nodes get mapped to common forwarding nodes. DFRA gives the two applications each a 64-1 *dedicated* allocation, delivering much higher I/O speedup in most cases, *plus* performance isolation from co-executing applications.

6.3 DFRA Decision Analysis

We now validate DFRA's forwarding node scaling decisions. Figure 13 shows, *in log scale,* performance of `MPI-IO` benchmarks with parameters uniformly sampled from a range, to adopt different I/O modes (N-1, N-N and N-M), I/O performing nodes, I/O request sizes, and metadata operation ratios. All tests are again divided into the *scaling* and *non-scaling* groups, referring to cases where the DFRA automatic scaling decision making processes chose to upgrade a job's forwarding node allocation, or retain the default one. The final results for both cases are consistent with the estimations projected by DFRA. Scaling cases can achieve on average 2.6× speedup (min at 1.1× and max at 7.1×), while the non-scaling ones' performance receives only trivial performance improvement (up to 1.05×).

Next we further examine the effectiveness of DFRA scaling, by measuring the queue length and I/O bandwidth of real-world applications on TaihuLight. Figure 14 shows results, again in log scale, with representative applications covering all I/O categories mentioned in Table 2. Among them, `DNDC` and `APT` are "non-scaling": `DNDC` is metadata-intensive and `APT` issues a large number of small-size I/O requests. We find their performance bottleneck not at the forwarding layer, explaining their little improvement in queue length and bandwidth when given more forwarding nodes. `AWP` adopts an N-1 I/O mode, generating high request pressure for forward-

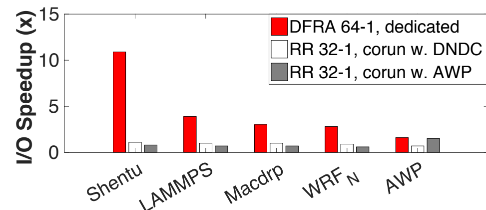


Figure 12: Speedup over 512-1 fixed allocation baseline, with two applications co-running, each using 256 compute nodes. Note that with its dedicated allocation, DFRA's performance is not impacted by co-running applications.

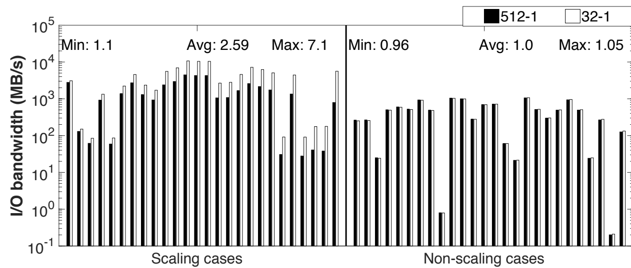
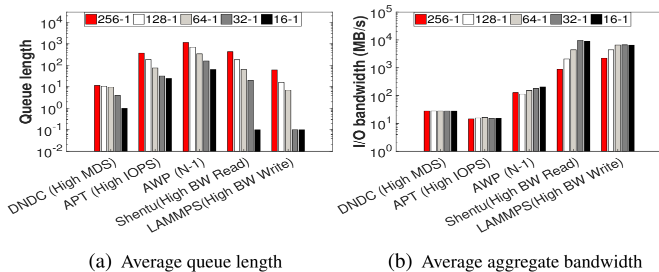


Figure 13: Scaling decision validation using `MPI-IO` benchmark instances. All experiments fix the total number of compute nodes to 256, with different number of them performing I/O, and run with 2 different compute-to-forwarding mapping ratios: 512-1 and 32-1. Results are sorted by speedup. Above the bars we list the minimum, average, and maximum speedup for each group.



(a) Average queue length (b) Average aggregate bandwidth

Figure 14: Average queue length (average number of requests pending in the queue, sampled at 0.01-second intervals) and I/O bandwidth with varying compute-to-forwarding mapping ratios during I/O execution. All applications use 256 compute node (1024 processes), with dedicated forwarding nodes.

ing nodes, thus receiving significant queue length improvement. Both `Shentu` and `LAMMPS` are bandwidth-hungry, benefiting significantly from the bandwidth side. In particular, `Shentu` gets a higher speedup as scaled-up allocation soothes its forwarding-side cache trashing.

6.4 Node Anomaly Screening

DFRA could screen out the abnormal forwarding nodes automatically. During our investigation, anomaly on forwarding nodes occurs for 6 times from Apr 2017 to Aug 2018. Jobs using such abnormal forwarding nodes typically experience substantial performance degradation. Figure 15 shows the performance impact when jobs get allocated an abnormal forwarding node. The I/O performance could see a 20× slowdown, due to the explicit barriers common with parallel

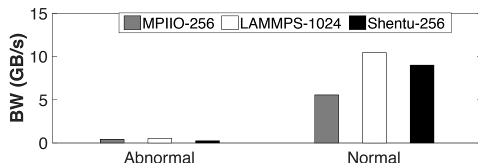


Figure 15: Performance impact when jobs run on abnormal forwarding nodes. All applications run with computing-forwarding ratio of 32-1.

I/O, forcing all processes to wait for the slow progress of the impaired forwarding node.

6.5 Overhead and Overall Resource Saving

Here we assess DFRA's overhead in performing the actual node remapping, while the allocation decision itself takes under 0.1s in all tests on TaihuLight. Figure 16 shows the average remapping time cost for different job sizes, plus the corresponding job dispatch time (without remapping) for reference. Though the remapping overhead increases linearly when more compute nodes are involved, it composes a minor addition to the baseline job dispatch overhead (the latter mainly due to compute nodes' slow wake-up from their power saving mode).

Note that this overhead is offset by our conservative screening based on jobs' past I/O profile. Even with 16,384 compute nodes, such minor delay in job dispatch is negligible compared with the total time saved in I/O phases, especially for long-running jobs. Since its deployment in Feb 2018, DFRA has brought an average execution time saving of over 6 minutes (up to several hours) to I/O-intensive jobs eligible for its remapping, estimated by comparing the I/O bandwidth benchmarked with the same application at the same job scale, before and after DFRA. Going over the actual TaihuLight job history, we thus estimate DFRA's overall resource saving at over 200 million of core-hours.

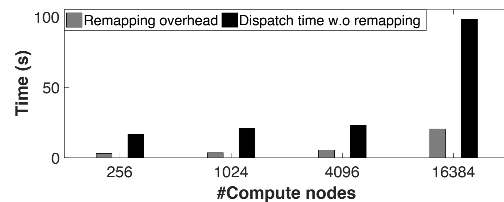


Figure 16: Average dynamic forwarding node remapping overhead

6.6 Extension to Burst Buffer Allocation

Finally, we briefly report our recent effort to apply DFRA techniques to dynamic allocation of burst buffer (BB) resources. We setup a testbed following the BB construction adopted by a previous study [41], containing 8 forwarding nodes, each with one 1.2TB Memblaze SSD to compose remote shared burst buffers.

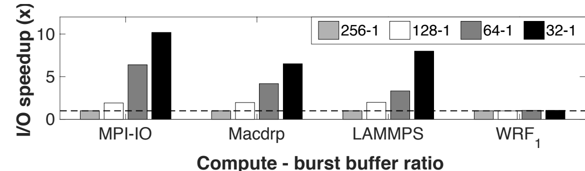


Figure 17: I/O speedup with different compute-to-BB node ratio, over performance with a baseline 256-1 allocation. Results are average from 3 tests, with error bars omitted due to small variance.

Figure 17 shows the performance impact of scaling up BB node allocations. All runs use 256 compute nodes. Not surprisingly, the more I/O-intensive applications (using N-N or

N-1) benefit significantly from more BB nodes, while the 1-1 mode $\texttt{WRF}_1$ sees little improvement. The similarity between such result and that with forwarding resource scaling suggests that DFRA is promising for BB layer management as well. To this end, the next generation Sunway supercomputer will adopt DFRA, including for its planned BB layer.

7 Related Work

I/O forwarding design and optimization Cplant [12] first introduces the I/O forwarding layer, but without support for data caching or request aggregation. I/O forwarding then became popular at extreme scale in IBM Blue Gene (BG) platforms [14, 36, 48]. IOFSL [13] is an open-source, portable, high performance I/O forwarding solution that provides a POSIX-like view of a forwarded file system to an application. The Cray XC series uses Data Virtualization Service (DVS) [4] for I/O forwarding. Our proposed DFRA methodology is compatible with recent trends in I/O forwarding adoption at large supercomputers, such as the Cray series.

For better I/O forwarding performance, Ohta et al. [50] present two optimization methods to reduce I/O bottlenecks: I/O pipelining and request scheduling. Vishwanath et al. [63] boost I/O forwarding through a work-queue model to schedule I/O and asynchronous data staging. PLFS adds an interposition software layer that transparently partitions files to improve N-1 performance [20]. DFRA is orthogonal to these optimizations and focuses on application-aware forwarding allocation and performance isolation.

Resource-aware scheduling This work echoes efforts in resource-aware scheduling, such as approaches improving utilization of datacenters/cloud resources, including CPU, cache, memory, and storage [22,34,58]. Our focus, however, is on HPC systems. To this end, AID [44] identifies applications' I/O patterns and reschedules heavy I/O applications to avoid congestion. CALCioM [28] coordinates applications' I/O activities dynamically via inter-application communication. Gainaru et al. propose a global scheduler [31], which based on system condition and applications' historical behavior prioritizes I/O requests across applications to reduce I/O interference. The libPIO [65] library monitors resource usage at the I/O routers and based on the loads allocates OSTs to specific I/O clients.

Regarding application-level I/O aware scheduling, ASCAR [40] is a storage traffic management framework that improves bandwidth utilization by I/O pattern classification. Lofstead et al. [46] propose an adaptive approach that groups processes and directs their output to particular storage targets, with inter-group coordination. IOrchestrator [71] builds a monitoring program to retrieve spatial locality information and schedules future I/O requests.

Our proposed scheme takes a different path that does not require any application or I/O middleware modification. It observes application and system I/O performance, and based on both real-time monitoring results and past monitoring history, automatically adjusts its default allocation to grant more or dedicated forwarding resources.

I/O interference analysis On detecting and mitigating interference, Yildiz et al. [70] examine sources of I/O interference in HPC storage systems and identify the bad flow control across the I/O path as a main cause. CALCiom [28] and Gainaaru's study [31] show that concurrent file system accesses lead to I/O bursts, and propose scheduling strategy enhancements. On relieving burst buffer congestion, Kougkas et al. [39] leverage burst buffer coordination to stage application I/O. TRIO [66] orchestrates application's write requests in the burst buffer and Thapaliya et al. [60] manage interference in the shared burst buffer through I/O request scheduling. The ADIOS I/O middleware manages interference by dynamically shifting workload from heavily used OSTs to those less loaded [42]. Qian et al. [52] present a token bucket filter in Lustre to guarantee QoS under interference.

This work is complementary to the above studies and uses interference analysis as a tool, achieving performance isolation using *interference avoidance*.

8 Conclusion

In this work, we explore adaptive storage resource provisioning for the widely used I/O forwarding architecture. Our experience of deploying it on the No.3 supercomputer and evaluating with ultra-scale applications finds dynamic, per-application forwarding resource allocation highly profitable. Judiciously applied to a minor fraction of jobs expected to be sensitive to forwarding node mapping, our remapping scheme both generates significant I/O performance improvement and mitigates inter-application I/O interference. We also report multiple prior findings by other researchers as confirmed or contradicted by our experiments. Finally, though this study has focused on the allocation of forwarding nodes, the same approach can apply to other resource types, such as burst buffer capacity/bandwidth allocation.

Acknowledgement

We thank Prof. Zheng Weimin for his valuable guidance and advice. We appreciate the thorough and constructive comments/suggestions from all reviewers. We thank our shepherd, Rob Johnson, for his guidance during the revision process. We would like to thank the National Supercomputing Center in Wuxi for great support to this work, as well as the Sunway TaihuLight users for providing test applications. This work is partially supported by the National Key R&D Program of China (Grant No. 2017YFA0604500 and 2016YFA0602100), and National Natural Science Foundation of China (Grant No. 61722208, 41776010, and U1806205).

References

[1] A description of the advanced research WRF version 3. http://www2.mmm.ucar.edu/wrf/users/.

[2] Cori supercomputer. http://www.nersc.gov/users/computational-systems/cori/.

[3] Cray burst buffer in Cori. http://www.nersc.gov/users/computational-systems/cori/burst-buffer/burst-buffer/.

[4] Cray data virtualization service (DVS). https://pubs.cray.com/content/S-0005/CLE%206.0.UP05/xctm-series-dvs-administration-guide/introduction-to-dvs.

[5] K supercomputer. http://www.aics.riken.jp/en/.

[6] Lightweight file systems. https://software.sandia.gov/trac/lwfs.

[7] MPI-IO test. http://freshmeat.sourceforge.net/projects/mpiiotest.

[8] Oakforest-PACS supercomputer. http://jcahpc.jp/eng/ofp_intro.html.

[9] Piz Daint supercomputer. https://www.cscs.ch/computers/dismissed/piz-daint-piz-dora/.

[10] Sequoia supercomputer. https://computation.llnl.gov/computers/sequoia.

[11] Sunway TaihuLight supercomputer. https://www.top500.org/system/178764.

[12] The computational plant. http://www.sandia.gov/~rbbrigh/slides/conferences/salinas-cplant-lci02-slides.pdf.

[13] The IOFSL project. https://www.mcs.anl.gov/research/projects/iofsl/about/.

[14] The ZeptoOS project. http://www.mcs.anl.gov/research/projects/zeptoos/.

[15] Titan supercomputer. https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/.

[16] Top 500 list. https://www.top500.org/resources/top-systems/.

[17] Trinity supercomputer. http://www.lanl.gov/projects/trinity/.

[18] Adiga, N. R., Almasi, G., Almasi, G. S., Aridor, Y., Barik, R., Beece, D. K., Bellofatto, R., Bhanot, G., Bickford, R., Blumrich, M. A., et al. An overview of the Blue Gene/L supercomputer. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2002).

[19] Ali, N., Carns, P., Iskra, K., Kimpe, D., Lang, S., Latham, R., Ross, R., Ward, L., and Sadayappan, P. Scalable I/O forwarding framework for high-performance computing systems. In *IEEE International Conference on Cluster Computing (CLUSTER)* (2009).

[20] Bent, J., Gibson, G., Grider, G., McClelland, B., and et al. PLFS: A checkpoint file system for parallel applications. In *Proceedings of Supercomputing* (2009).

[21] Bin Yang, Xu Ji, X. M. T. Z. X. Z. X. W. N. E.-s. J. Z. W. L. W. X. End-to-end I/O Monitoring on a Leading Supercomputer. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2019).

[22] Bindschaedler, L., Malicevic, J., Schiper, N., Goel, A., and Zwaenepoel, W. Rock you like a hurricane: Taming skew in large scale analytics. In *European Conference on Computer Systems (EuroSys)* (2018).

[23] Bingwei Chen, Haohuan Fu, Y. W. C. H. W. Z. Y. L. W. W.-W. Z. L. G. W. Z. Z. Z. G. Y. X. C. Simulating the Wenchuan Earthquake with accurate surface topography on Sunway TaihuLight. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2018).

[24] Braam, P. J., and Zahir, R. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc* (2002).

[25] Cui, Y., Olsen, K. B., Jordan, T. H., Lee, K., Zhou, J., Small, P., Roten, D., Ely, G., Panda, D. K., Chourasia, A., et al. Scalable earthquake simulation on petascale supercomputers. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2010).

[26] Cui, Y., Poyraz, E., Olsen, K. B., Zhou, J., Withers, K., Callaghan, S., Larkin, J., Guest, C., Choi, D., Chourasia, A., et al. Physics-based seismic hazard analysis on petascale heterogeneous supercomputers. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2013).

[27] Daley, C. S., Ghoshal, D., Lockwood, G. K., Dosanjh, S., Ramakrishnan, L., and Wright, N. J. Performance characterization of scientific workflows for the optimal use of burst buffers. *Future Generation Computer Systems* (2017).

[28] Dorier, M., Antoniu, G., Ross, R., Kimpe, D., and Ibrahim, S. CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2014).

[29] Fang, J., Fu, H., Zhao, W., Chen, B., Zheng, W., and Yang, G. swDNN: A library for accelerating deep learning applications on Sunway Taihulight. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2017).

[30] Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., and Yang, G. The Sunway TaihuLight supercomputer: System and applications. *Science CHINA Information Sciences* (2016).

[31] Gainaru, A., Aupy, G., Benoit, A., Cappello, F., Robert, Y., and Snir, M. Scheduling the I/O of HPC applications under congestion. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2015).

[32] Giltrap, D. L., Li, C., and Saggar, S. DNDC: A process-based model of greenhouse gas fluxes from agricultural soils. *Agriculture, Ecosystems & Environment* (2010).

[33] Gunawi, H. S., Suminto, R. O., Sears, R., Golliher, C., Sundararaman, S., Lin, X., Emami, T., Sheng, W., Bidokhti, N., McCaffrey, C., et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *16th USENIX Conference on File and Storage Technologies (FAST)* (2018).

[34] Helgi Sigurbjarnarson, Petur Orri Ragnarsson, Y. V. M. B. Harmonium: Elastic cloud storage via file motifs. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2014).

[35] Heng Lin, Xiaowei Zhu, B. Y. X. T. W. X. W. C. L. Z.-T. H. X. M. X. L. W. Z., and Xu, J. ShenTu: Processing multi-trillion edge graphs on millions of cores in seconds. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2018).

[36] Iskra, K., Romein, J. W., Yoshii, K., and Beckman, P. ZOID: I/O-forwarding infrastructure for petascale architectures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2008).

[37] Kay, J., Deser, C., Phillips, A., Mai, A., Hannay, C., Strand, G., Arblaster, J., Bates, S., Danabasoglu, G., Edwards, J., et al. The Community Earth System Model (CESM) large ensemble project: A community resource for studying climate change in the presence of internal climate variability. *Bulletin of the American Meteorological Society* (2015).

[38] Kim, Y., Atchley, S., and Shipman, G. M. LADS: Optimizing data transfers using layout-aware data scheduling. In *13th USENIX Conference on File and Storage Technologies (FAST)* (2015).

[39] Kougkas, A., Dorier, M., Latham, R., Ross, R., and Sun, X. H. Leveraging burst buffer coordination to prevent I/O interference. In *IEEE International Conference on E-Science* (2017).

[40] Li, Y., Lu, X., Miller, E. L., and Long, D. D. E. ASCAR: Automating contention management for high-performance storage systems. In *IEEE International Conference on Massive Storage Systems and Technology (MSST)* (2015).

[41] Liu, N., Cope, J., Carns, P., Carothers, C., Ross, R., Grider, G., Crume, A., and Maltzahn, C. On the role of burst buffers in leadership-class storage systems. In *IEEE International Conference on Massive Storage Systems and Technology (MSST)* (2012).

[42] Liu, Q., Logan, J., Tian, Y., Abbasi, H., Podhorszki, N., Choi, J. Y., Klasky, S., Tchoua, R., Lofstead, J., Oldfield, R., et al. Hello ADIOS: The challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience* (2014).

[43] Liu, Y., Gunasekaran, R., Ma, X., and Vazhkudai, S. S. Automatic identification of application I/O signatures from noisy server-side traces. In *12th USENIX Conference on File and Storage Technologies (FAST)* (2014).

[44] Liu, Y., Gunasekaran, R., Ma, X., and Vazhkudai, S. S. Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2016).

[45] Lofstead, J., Jimenez, I., Maltzahn, C., Koziol, Q., Bent, J., and Barton, E. DAOS and friends: A proposal for an exascale storage system. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2016).

[46] Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., and Wolf, M. Managing variability in the I/O performance of petascale storage systems. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2010).

[47] Luu, H., Winslett, M., Gropp, W., Ross, R., Carns, P., Harms, K., Prabhat, M., Byna, S., and Yao, Y. A multiplatform study of I/O behavior on petascale supercomputers. In *Proceedings of the 24th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2015).

[48] Moreira, J., Brutman, M., Castano, J., and Engelsiepen, T. Designing a highly-scalable operating system: The Blue Gene/L story. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2006).

[49] Oberg, M., Tufo, H. M., and Woitaszek, M. Exploration of parallel storage architectures for a Blue Gene/L on the TeraGrid. In *9th LCI International Conference on High-Performance Clustered Computing* (2008).

[50] Ohta, K., Kimpe, D., Cope, J., Iskra, K., Ross, R., and Ishikawa, Y. Optimization techniques at the I/O forwarding layer. In *IEEE International Conference on Cluster Computing (CLUSTER)* (2010).

[51] Petersen, T. K., and Bent, J. Hybrid flash arrays for HPC storage systems: An alternative to burst buffers. In *IEEE High Performance Extreme Computing Conference (HPEC)* (2017).

[52] Qian, Y., Li, X., Ihara, S., Zeng, L., Kaiser, J., Süss, T., and Brinkmann, A. A configurable rule based classful token bucket filter network request scheduler for the lustre file system. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2017).

[53] Riesen, R., Brightwell, R., Hudson, T., Hudson, T., Maccabe, A. B., Widener, P. M., and Ferreira, K. Designing and implementing lightweight kernels for capability computing. *Concurrency & Computation Practice & Experience* (2009).

[54] Roten, D., Cui, Y., Olsen, K. B., Day, S. M., Withers, K., Savran, W. H., Wang, P., and Mu, D. High-frequency nonlinear earthquake simulations on petascale heterogeneous supercomputers. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2016).

[55] Sadaf Alam, Nicola Bianchi, N. C. M. C. M. G. S. G. M. K.-C. M. M. P. C. P. F. V. An operational perspective on a hybrid and heterogeneous Cray XC50 system. In *Proceedings of Cray User Group Conference (CUG)* (2017).

[56] Sakai, K., Sumimoto, S., and Kurokawa, M. High-Performance and Highly Reliable File System for the K computer. *Fujitsu Scientific & Technical Journal* (2012).

[57] Schmuck, F. B., and Haskin, R. L. GPFS: A shared-disk file system for large computing clusters. In *1st USENIX Conference on File and Storage Technologies (FAST)* (2002).

[58] Sigurbjarnarson, H., Ragnarsson, P. O., Yang, J., Vigfusson, Y., and Balakrishnan, M. Enabling space elasticity in storage systems. In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR)* (2016).

[59] Smith, R., and Gent, P. Reference manual for the Parallel Ocean Program (POP), ocean component of the Community Climate System Model (CCSM2. 0 and 3.0). Tech. rep., Technical Report LA-UR-02-2484, Los Alamos National Laboratory, Los Alamos., (2002).

[60] Thapaliya, S., Bangalore, P., Lofstead, J., Mohror, K., and Moody, A. Managing I/O interference in a shared burst buffer system. In *International Conference on Parallel Processing (ICPP)* (2016).

[61] Tsujita, Y., Yoshizaki, T., Yamamoto, K., Sueyasu, F., Miyazaki, R., and Uno, A. Alleviating I/O interference through workload-aware striping and load-balancing on parallel file systems. In *International Supercomputing Conference (ISC)* (2017).

[62] Vijayakumar, K., Mueller, F., Ma, X., and Roth, P. C. Scalable I/O tracing and analysis. In *IEEE/ACM Petascale Data Storage Workshop (PDSW)* (2009).

[63] Vishwanath, V., Hereld, M., Iskra, K., Kimpe, D., Morozov, V., Papka, M. E., Ross, R., and Yoshii, K. Accelerating I/O forwarding in IBM Blue Gene/P systems. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2010).

[64] Wallace, D. Compute node Linux: Overview, progress to date, and roadmap. In *Proceedings of Cray User Group Conference (CUG)* (2007).

[65] Wang, F., Oral, S., Gupta, S., Tiwari, D., and Vazhkudai, S. Improving large-scale storage system performance via topology-aware and balanced data placement. *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep* (2014).

[66] Wang, T., Oral, S., Pritchard, M., Wang, B., and Yu, W. TRIO: Burst buffer based I/O orchestration. In *IEEE International Conference on Cluster Computing (CLUSTER)* (2015).

[67] Wang, Y., Liu, J., Qin, H., Yu, Z., and Yao, Y. The accurate particle tracer code. *Computer Physics Communications* (2017).

[68] XIAOHUI DUAN, PING GAO, T. Z. M. Z. W. L. W. Z. W. X.-H. F. L. G. D. C. X. M. G. Y. Redesigning LAMMPS for petascale and hundred-billion-atom simulation on Sunway TaihuLight. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2018).

[69] XU, W., LU, Y., LI, Q., ZHOU, E., SONG, Z., DONG, Y., ZHANG, W., WEI, D., ZHANG, X., CHEN, H., ET AL. Hybrid hierarchy storage system in MilkyWay-2 supercomputer. *Frontiers of Computer Science* (2014).

[70] YILDIZ, O., DORIER, M., IBRAHIM, S., ROSS, R., AND ANTONIU, G. On the root causes of cross-application I/O interference in HPC storage systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016).

[71] ZHANG, X., DAVIS, K., AND JIANG, S. IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2010).

Design Tradeoffs for SSD Reliability

Bryan S. Kim
Seoul National University

Jongmoo Choi
Dankook University

Sang Lyul Min
Seoul National University

Abstract

Flash memory-based SSDs are popular across a wide range of data storage markets, while the underlying storage medium—flash memory—is becoming increasingly unreliable. As a result, modern SSDs employ a number of in-device reliability enhancement techniques, but none of them offers a *one size fits all* solution when considering the multi-dimensional requirements for SSDs: performance, reliability, and lifetime.

In this paper, we examine the design tradeoffs of existing reliability enhancement techniques such as data re-read, intra-SSD redundancy, and data scrubbing. We observe that an uncoordinated use of these techniques adversely affects the performance of the SSD, and careful management of the techniques is necessary for a graceful performance degradation while maintaining a high reliability standard. To that end, we propose a holistic reliability management scheme that selectively employs redundancy, conditionally re-reads, judiciously selects data to scrub. We demonstrate the effectiveness of our scheme by evaluating it across a set of I/O workloads and SSDs wear states.

1 Introduction

From small mobile devices to large-scale storage servers, flash memory-based SSDs have become a mainstream storage device thanks to flash memory's small size, energy efficiency, low latency, and collectively massive parallelism. The popularity of SSDs is fueled by the continued drop in cost per GB, which in turn is achieved by storing multiple bits in a memory cell [7, 42] and vertically stacking memory layers [41, 48].

However, the drive for high storage density has caused the flash memory to become less reliable and more error-prone [9, 20]. Raw bit error rate measurement of a single-level cell flash memory in 2009 was in the order of 10^{-8} [19], but this increased to 10^{-7}–10^{-4} in 2011 for a 3x-nm multi-level cell [52] and to 10^{-3}–10^{-2} in 2017 for a 1x-nm memory [10]. The high error rates in today's flash memory are caused by various reasons, from wear-and-tear [11, 19, 27], to gradual charge leakage [11, 14, 51] and data disturbance [13, 19].

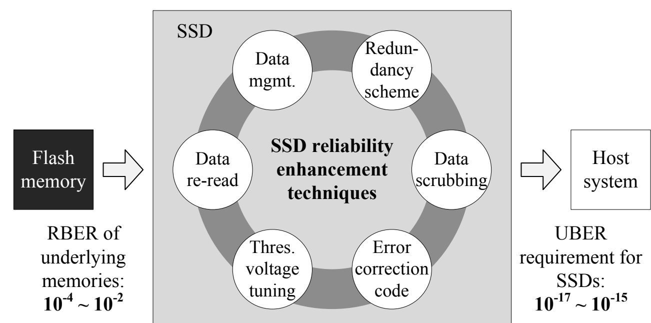


Figure 1: SSD reliability enhancement techniques.

In order to mask out the error-prone nature of flash memory, the state-of-the-art SSDs employ a number of in-device reliability enhancement techniques, as shown in Figure 1. These techniques originate from a wide range of domains, from device physics that tunes threshold voltage levels for sensing memory states [12, 14, 38], coding theory that corrects errors using computed parity information [18, 35], to system-level approaches such as scrubbing that preventively relocates data [22, 37]. This variety is caused by the fact that there is no *one size fits all* solution for data protection and recovery: each technique has a multi-dimensional design tradeoff that makes it necessary to compositionally combine complementary solutions. This is much easier said than done as reliability is only one of the many design goals for SSDs: a study spanning across multiple institutions reveals that these reliability enhancements, in fact, cause performance degradation in SSDs [21].

In this paper, we examine the design tradeoffs of existing techniques across multiple dimensions such as average and tail performance, write amplification, and reliability. Our investigation is inspired by studies in the HDD domain that evaluate the effectiveness of different reliability enhance-

ments [24, 40, 49], but our findings deviate from those work due to the difference in the underlying technology and the SSD's internal management. We make the following three observations from our experiments. First, the use of data re-read mechanisms should be managed, as the repeated re-reads further induce errors, especially for read disturbance-vulnerable cells. Second, in the absence of random and sporadic errors, the overheads of intra-SSD redundancy outweigh its benefits in terms of performance, write amplification, and reliability. Lastly, SSD-internal scrubbing reduces the error-induced long-tail latencies, but it increases the internal traffic that negates its benefits.

Based on our observation, we propose a holistic reliability management scheme that selectively employs intra-SSD redundancy depending on access characteristics of the data, conditionally uses data re-read mechanism to reduce the effects of read disturbance, and judiciously selects data to scrub so that the internal relocation traffic is managed. Redundancy is applied only to infrequently accessed *cold* data to reduce write amplification, and frequently read *read-hot* data are selected for scrubbing based on a cost-benefit analysis (overhead of internal traffic vs. reduction in re-reads). In this paper, we present the following:

- We construct and describe an SSD architecture that holistically incorporates complementary reliability enhancement techniques used in modern SSDs. (§ 3)
- We evaluate the state-of-the-art solutions across a wide range of SSD states based on a number of flash memory error models, and discuss their tradeoffs in terms of performance, reliability, and lifetime. (§ 4)
- We propose a holistic reliability management scheme that self-manages the use of multiple error-handling techniques, and we demonstrate its effectiveness across a set of real I/O workloads. (§ 5)

2 Background

In this section, we describe the causes of flash memory errors and their modeling, and the existing reliability enhancement techniques that correct and prevent errors. For more detailed and in-depth explanations, please refer to Mielke *et al.* [43] for error mechanisms and modeling, and Cai *et al.* [9] for error correction and prevention techniques.

2.1 Errors in Flash Memory

We focus on three major sources of flash memory errors: wear, retention loss, and disturbance.

Wear. Repeated programs and erases (also known as P/E cycling) wear out the flash memory cells that store electrons (data), and cause irreversible damage to them [9, 11, 19]. Flash memory manufacturers thus specify an *endurance limit*, a number of P/E cycles a flash memory block can withstand, and this limit has been steadily decreasing for every new generation of flash memory. However, the endurance limit is *not* a hard limit: not all blocks are created equally due to process variations, and a number of studies dynamically measure the lifetime of a block to extend its usage [27].

Retention loss. Electrons stored in flash memory cells gradually leak over time, making it difficult to correctly read the data stored, and errors caused by retention loss increase as cells wear [9, 14, 43]. While a number of studies indicate that retention loss is a dominant source of errors [11, 14], retention errors are fortunately transient: they reset once the block is erased [43].

Disturbance. Reading a wordline in a block *weakly* programs other wordlines in the block, unintentionally inserting more electrons into their memory cells [9, 19, 43]. Disturbance and retention errors are opposing error mechanisms, but they do not necessarily cancel each other out: disturbance mainly affects cells with fewer electrons (erased state), but charge leakage affects those with more (programmed state) [9, 43]. Similar to retention loss, errors caused by read disturbances increase as cells wear and reset once the block is erased [43].

These three sources of errors are used to model the raw bit error rate (RBER) of flash memory with the following additive power-law variant [36, 43]:

$$\begin{aligned} &RBER(cycles, time, reads) && (1)\\ &= \varepsilon + \alpha \cdot cycles^k && \text{(wear)}\\ &\quad + \beta \cdot cycles^m \cdot time^n && \text{(retention)}\\ &\quad + \gamma \cdot cycles^p \cdot reads^q && \text{(disturbance)} \end{aligned}$$

where ε, α, β, and γ are coefficients and k, m, n, p, and q are exponents particular to a flash memory. These *nine* parameters define the RBER of a flash memory chip, and Mielke *et al.* [43] and Liu *et al.* [36] further explain the validity for the additive power-law model in detail.

2.2 SSD Reliability Enhancement Techniques

Table 1 outlines the tradeoffs for the commonly used reliability enhancement techniques.

Error correction code (hard-decision). Hard-decision ECC such as BCH (code developed by Bose, Ray-Chaudhuri, and Hocquenghem) [35] is the first line of defense against flash memory errors. When writing, the ECC encoder computes additional parity information based on the data, which is typically stored together in the same page. Flash memory manufacturers conveniently provide additional spare bytes within a page for this purpose. When reading, the hard-decision ECC decoder returns the error-corrected data or reports a failure after a fixed number of cycles. With the continued decline in flash reliability, it is

Table 1: Comparison of SSD reliability enhancement techniques.

Techniques	Impact on average performance	Impact on tail performance	Write amplification	Management overhead	Related work
ECC (hard-decision)	Negligible	None	Negligible	None	BCH, LDPC [35]
ECC (soft-decision)	None	High	Negligible	Negligible	LDPC [18,35]
Threshold voltage tuning	None	High	None	Voltage levels	Read retry [12] Voltage prediction [14,38]
Intra-SSD redundancy	High for small stripes; low for large stripes	Low for small stripes; high for large stripes	High	Stripe group information	Dynamic striping [31,33] Intra-block striping [46] Parity reduction [25,34]
Background data scrubbing	Depends	Depends	Depends	Block metadata such as erase count or read count	Read reclaim [22] Read refresh [37]

becoming increasingly inefficient to rely solely on stronger ECC engines [15].

Error correction code (soft-decision). Soft-decision ECC such as LDPC (low-density parity-check) [18,35] also encodes additional parity, but uses soft-information—the probability of each bit being 1 or 0—for decoding. This requires the data in flash memory to be read multiple times. The error correction strength of soft-decision decoding is orders of magnitude greater than its hard-decision counterpart, but this is achieved at an expense of multiple flash memory reads.

Threshold voltage tuning. The electrons stored in flash memory cells gradually shift over time due to charge leakage [11, 14] and disturbance [13, 19]. To counteract this drift, threshold voltages for detecting the charge levels are adjustable through special flash memory commands [12]. In SSD designs without soft-decision ECC, data are re-read after tuning the threshold voltages if the hard-decision ECC fails [9]. Although the underlying mechanisms are different, both soft-decision ECC and threshold voltage tuning share the same high-level design tradeoff: the greater the probability of correcting errors with repeated reads.

Intra-SSD redundancy. SSDs can internally add redundancy across multiple flash memory chips [31,33,46], similar to how RAID [47] protects data by adding redundancy across multiple physical storage devices. While both ECC and RAID-like redundancy enhance the SSD's reliability by adding extra parity information, striping data across multiple chips protects the SSD against chip and wordline failures that effectively renders the traditional ECC useless. In general, increasing the stripe size trades the overhead of parity writes for the penalty of reconstructing data. In the context of SSDs, employing redundancy amplifies the write traffic not only because of parity writes, but also because the effective over-provisioning factor is decreased.

Data scrubbing. We use this as an umbrella term in this paper for the variety of SSD's housekeeping tasks that enhance reliability. This includes read reclaim [22] that addresses read disturbance-induced errors, and read refresh [37] that handles retention errors. While ECC and voltage tuning *correct* errors, these tasks *prevent* errors: by monitoring SSD-internal information, data are preventively relocated before errors accumulate to a level beyond error correction capabilities. In effect, background data scrubbing reduces the overhead of error correction by creating additional internal traffic, but this traffic also affects the QoS performance and accelerates wear.

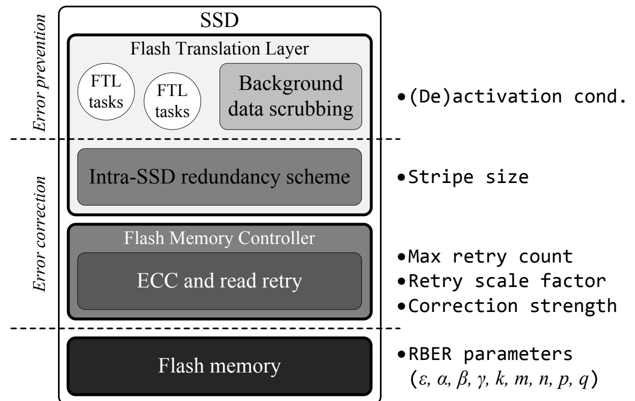


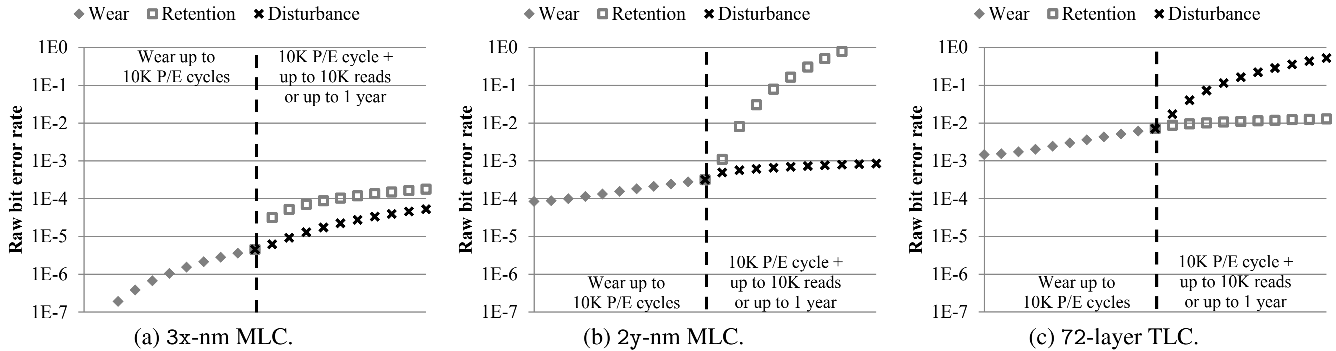
Figure 2: The overall error-handling architecture of an SSD, and its associated configuration parameters.

3 Design Tradeoffs for SSD Reliability

To understand how data protection and recovery schemes in modern SSDs ensure data integrity in the midst of flash memory errors, we construct an SSD model that holistically considers the existing reliability enhancement techniques. Figure 2 illustrates this SSD model with particular emphasis on error-related components and their configurable parameters.

Table 2: RBER model parameters. Parameters ε, α, β, γ, k, m, n, p, and q describe the RBER model in Equation 1. R^2 represents the goodness of fit and is computed using the log values of the data and model, and N is the sample size.

Flash memory	Year	ε	α	β	γ	k	m	n	p	q	R^2	N
3x-nm MLC [52]	2011	5.06E-08	1.05E-14	9.31E-14	4.17E-15	2.16	1.80	0.80	1.07	1.45	0.984	98
2y-nm MLC [13, 14]	2015	8.34E-05	3.30E-11	5.56E-19	6.26E-13	1.71	2.49	3.33	1.76	0.47	0.988	173
72-layer TLC	2018	1.48E-03	3.90E-10	6.28E-05	3.73E-09	2.05	0.14	0.54	0.33	1.71	0.969	54



(a) 3x-nm MLC. (b) 2y-nm MLC. (c) 72-layer TLC.

Figure 3: Projected RBER graphs based on model parameters in Table 2. Each graph shows the error rate caused by the three mechanisms: wear, retention loss, and disturbance. In the first half of the x-axis, RBER increases due to repeated programs and erases (up to 10K cycles). In the second half, the cells are kept at 10K P/E cycle, but the data are repeatedly read (up to 10K reads) to induce disturbance errors or are left unaccessed (up to 1 year) for retention errors.

3.1 Error-Prone Flash Memory

Flash memory is becoming increasingly unreliable in favor of high-density [20], and we observe this trend across error datasets we analyzed. Table 2 shows the *nine* `RBER parameters` (see Equation 1 in § 2.1) for three different flash memory chips: 3x-nm MLC, 2y-nm MLC, and 72-layer TLC. We curve-fit the parameters of the three chips through simulated annealing. The datasets for the 3x-nm MLC [52] and 2y-nm MLC [13, 14][1] are extracted from the figures in the publications using plot digitization [3], while the dataset for the 72-layer TLC is provided by a flash memory manufacturer.

Figure 3 illustrates the contributing factors of errors for these chips. The graphs are generated based on the RBER model parameters in Table 2, and show that the overall error rate increases with the newer and denser flash memories. The most interesting observation, however, is the dominance of disturbance errors in the 72-layer TLC. This is in stark contrast with the 2y-nm MLC whose dominant error is due to retention loss.

In this work, we neither argue the importance of one error type over the other, nor claim a shifting trend in dominant errors. In fact, the sample space and sample size of the three datasets for the flash memory chips are different, making it difficult to compare equally. For example, we do not claim that the 2y-nm MLC in Figure 3b will have a 100% error rate after 1 year: the projected retention loss error is computed based on a limited number of RBER data samples that cover a smaller subset of the sample space. Rather, the graphical representation of Figure 3 is only used to illustrate the wide variation in error characteristics, and that an error-handling technique tailored for one particular memory chip may fail to meet the reliability requirements in others.

3.2 Mechanism in Flash Memory Controller

The flash memory controller not only abstracts the operational details of flash memory, but also handles common-case error correction. In addition to hard-decision ECC, soft-decision ECC and threshold voltage tuning are implemented in the controller as their mechanisms simply iterate through a pre-defined set of threshold voltage levels for successive reads (although setting appropriate voltage levels may involve the firmware).

The hard-decision ECC tolerates up to n-bit errors, defined by the `correction strength`. Increasing the ECC correction strength not only increases the logic complexity and power consumption, but also inflates the amount of parity data that needs to be stored in flash. The fixed number of bytes per page (including the spare) is thus the limiting factor for the ECC's capability. Errors beyond the hard-decision ECC's correction strength are subsequently handled by the flash memory controller with data re-reads. In these cases, the same data are accessed again, repeatedly if needed. If the data cannot be recovered after `max retry count`, the firmware is notified of a read error. We model threshold voltage tuning and soft-decision decoding in a way that each successive reads effectively reduces the RBER of the data by `retry scale factor`. This model is general enough to

cover both mechanisms, and they will be referred to as *data re-reads* henceforth.

3.3 Role of Flash Translation Layer

The flash translation layer (FTL) consists of a number of SSD-internal housekeeping tasks that collectively hide the quirks of flash memory and provide an illusion of a traditional block device. Mapping table management and garbage collection are the widely known FTL tasks, but these will not be discussed in detail. In the context of reliability, we focus on intra-SSD redundancy and data scrubbing.

Intra-SSD redundancy is used to reconstruct data when ECC (both hard-decision and data re-read) fails. In this work, we focus on constructing redundancy based on the physical addresses. Upon writes, *one* parity is added for every s data writes, defined by the `stripe size`. The s data and *one* parity create a stripe group, and the parity is distributed among the stripe group, akin to the workings of RAID 5. If the flash memory controller reports an uncorrectable error, the firmware handles the recovery mechanism by identifying the stripe group that dynamically formed when the data were written [31, 32]. If any of the other data in the same stripe group also fails in the ECC layer, the SSD reports an uncorrectable error to the host system. If the data are not protected by redundancy (`stripe size` of ∞), any ECC failure causes an uncorrectable error.

While the techniques discussed so far correct errors, data scrubbing prevent errors from accumulating by relocating them in the background. The scrubber `activates` and `deactivates` under certain conditions, which depends on the implementation: it can trigger periodically and scan the entire address space [6, 45], it can activate once the number of reads per block exceeds a threshold [22, 30], or it can relocate data based on the expected retention expiration [37].

These firmware-oriented reliability enhancements pay a cost in the *present* to reduce the penalty in the *future*. For intra-SSD redundancy, the increased frequency of writing parity data reduces the number of reads to reconstruct the data. For data scrubbing, proactive relocation of data prevents the ECC in the controller from failing. At the same time, both techniques increase the write amplification and accelerate wear, not only reducing the lifetime of the SSD, but also effectively increasing the chance of future errors. This cyclical dependence makes it difficult to quantify the exact benefits and overheads of these techniques.

4 Evaluation of SSD Reliability

We implement the flash memory error model and the reliability enhancements on top of the DiskSim environment [1] by extending its SSD extension [5]. We construct three SSDs, each with three different initial wear states: The 3x-nm MLC blocks are initialized with 10K, 30K, and 50K P/E cycles on average, the 2y-nm MLC blocks to 2K, 5K, and 10K P/E cycles, and the 72-layer TLC to 1K, 3K, and 5K. These *nine* SSD states have different error rates, but are otherwise identical in configuration. Realistically, these SSDs should have different capacities (page size, number of pages per block, number of blocks, etc.) and even operation latencies, but we use the same internal organization to isolate the effects of reliability enhancement techniques on the overall performance. Table 3 summarizes the SSD's internal configuration.

Table 3: System configuration.

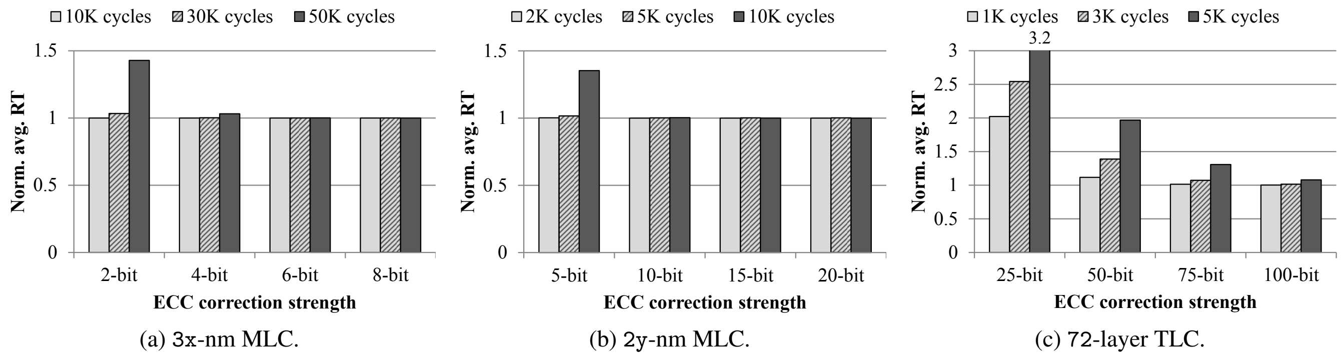
Parameter	Value	Parameter	Value
# of channels	8	Read latency	50μs
# of chips/channel	4	Program latency	500μs
# of planes/chip	2	Erase latency	5ms
# of blocks/plane	1024	Data transfer rate	667MB/s
# of pages/block	256	Physical capacity	256GiB
Page size	16KiB	Logical capacity	200GiB

We extend our prior work [30] that includes all essential FTL functionalities such as host request handling and garbage collection (GC) and implement the discussed reliability enhancement schemes. Host requests are handled in a non-blocking manner to fully utilize the underlying parallelism, and all FTL tasks run independently and concurrently. Flash memory requests from these tasks are generated with some delay to model the think time, and up to *eight* tasks can be active concurrently to model the limited number of embedded processors in SSDs. Host addresses are translated in 4KiB mapping granularity, and the entire map is resident in DRAM. Host data, GC data, and data from the scrubber are written to different sets of blocks to separate hot and cold data, and they are arbitrated by a prioritized scheduler: host requests have the highest priority, then garbage collection, and lastly scrubbing.

For evaluation, we use synthetic workloads of 4KiB read/write with a 70:30 mixture, and the access pattern has a skewed distribution to mimic the SSD endurance workload specification [2]. I/O requests arrive at the SSD every 0.5ms on average (2K IOPS): the I/O intensity is intentionally set so that the garbage collector's impact on the overall performance is properly reflected. The workload runs for an hour, executing 7.2 million I/Os in total. Prior to each experiment, data are randomly written to the entire physical space to emulate a pre-conditioned state.

4.1 Error Correction Code

We first investigate how the SSDs perform when relying solely on the flash memory controller. In this scenario, background scrubbing is disabled, and no intra-SSD redundancy is used. We test a number of `correction strength`, including ∞ that corrects all errors. Realistically, stronger ECC engines require larger ECC parity, but we assume that the



(a) 3x-nm MLC. (b) 2y-nm MLC. (c) 72-layer TLC.

Figure 4: Average read response time for the three SSDs at various wear states. For each graph, the *x*-axis shows the correction strength for the ECC, and the performance is normalized to that with ∞ error correction strength. The response time increases not only when the SSD is more worn out, but also when weaker ECC is used.

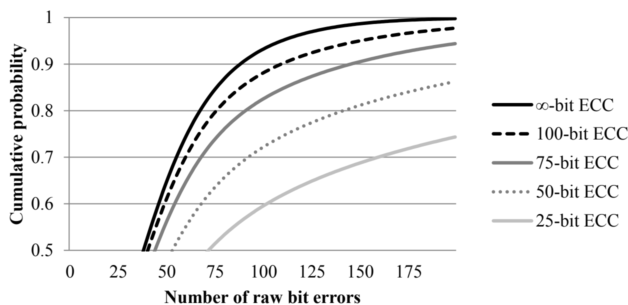


Figure 5: CDF of raw bit errors with varying ECC correction strength for the 72-layer TLC SSD of 5K initial wear state. With a ∞-bit ECC, 85% of data have less than 75-bit errors, but at 25-bit ECC, only 51% of data are. With more data being re-read, read disturbance increases the bit error rate.

flash memory page always has sufficient spare bytes. When ECC fails, data is re-read after adjusting the threshold voltage. Each data re-read reduces the RBER by 50% (`retry scale factor` of *2*), repeating until the error is corrected (`max retry count` of ∞).

Figure 4 shows the average response time for read requests for the three types of SSDs at various wear states. We chose the ECC correction strengths based on the relative RBER of the three flash memories. We observe that the performance degrades not only when the SSD is more worn out but also with weaker ECC correction strength. In the higher SSD wear states, errors are more frequent, and weaker ECC induces more data re-reads.

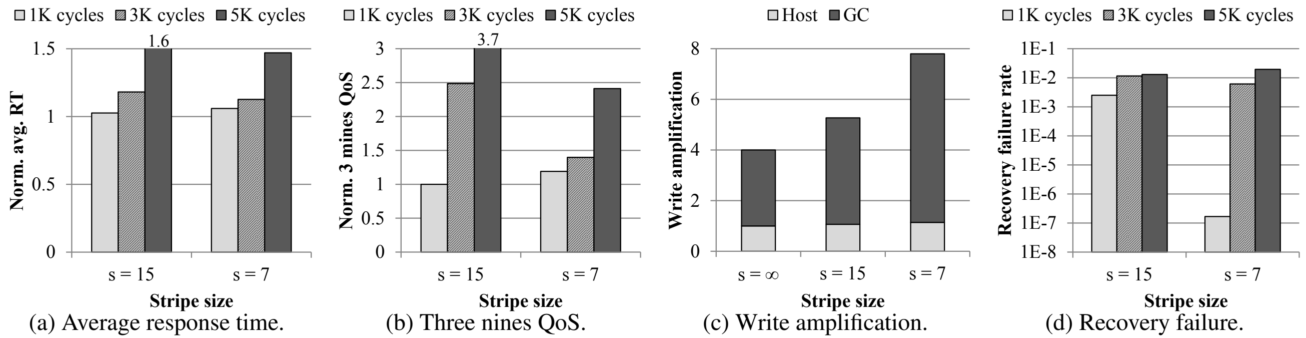
Compared to the 3x-nm (Figure 4a) and 2y-nm MLC (Figure 4b), the 72-layer TLC in Figure 4c shows a greater performance degradation. This is not only because of the higher overall RBER, but also because of relatively higher vulnerability to read disturbances (cf. Figure 3c). Figure 5 illustrates this case: it shows the cumulative distribution of measured raw bit errors with different ECC strengths for the 72-layer TLC at 5K P/E cycle wear state. When no data re-reads occur (∞-bit ECC), 85% of the ECC checks have less than 75-bit errors. With weaker ECC, however, the subsequent data re-reads induce additional read disturbance, lowering the CDF curve. Only 74% of the data have less than 75-bit errors when using a 75-bit ECC, but this further drops to 51% with a 25-bit ECC correction strength.

Thus, for read disturbance-sensitive memories, avoiding frequent data re-reads is critical for improving the performance. However, as illustrated in the upper portion of Figure 5, increasing the ECC strength has diminishing returns. This necessitates the use of ECC-complementary schemes that read pages in other blocks to reconstruct data (intra-SSD redundancy) or reduce the probability of data re-reads through preventive data relocations (data scrubbing).

4.2 Intra-SSD Redundancy

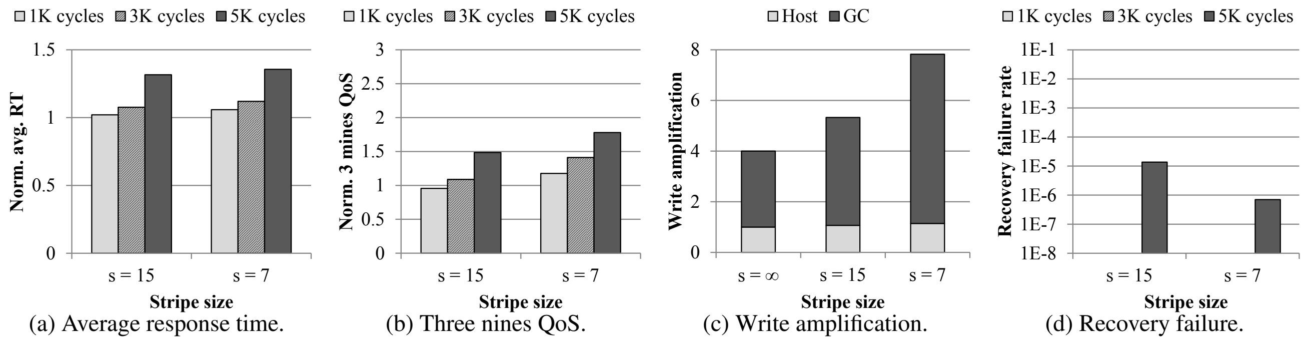
If data cannot be corrected after a given number of data re-read attempts, the data are reconstructed using other data within the stripe group. In this experiment, we examine the performance, reliability, and write amplification aspect of intra-SSD redundancy. We present the results from the 72-layer TLC using 75-bit ECC.

Figure 6 shows the results when `max retry count` is *one*: the flash memory controller attempts a data re-read scheme once before notifying the firmware. As shown in Figure 6a and Figure 6b, attempting frequent data reconstruction degrades performance especially in terms of long-tail latency because of increased internal traffic. The degradation is more severe for higher wear states (more errors), and greater stripe size (more pages accessed). This performance penalty is in addition to the increase in write amplification illustrated in Figure 6c. Even though host data programs are amplified due to parity writes, GC data programs are amplified at a greater rate because of the reduced effective capacity: without redundancy, the effective over-provisioning factor is 28%, but this drops to 20% when s=15, and to 12% when s=7. Furthermore, using intra-SSD redundancy does not guarantee full data recovery: accessing other pages in the stripe group can be uncorrectable through



(a) Average response time. (b) Three nines QoS. (c) Write amplification. (d) Recovery failure.

Figure 6: Performance, write amplification, and reliability for the 72-layer TLC SSD when `max retry count` is *one*. The performances in Figure 6a and Figure 6b are normalized to a system with ∞ correction strength. Using intra-SSD redundancy increases write amplification (Figure 6c), but moreover does not warrant full data recovery (Figure 6d).



(a) Average response time. (b) Three nines QoS. (c) Write amplification. (d) Recovery failure.

Figure 7: Performance, write amplification, and reliability for the 72-layer TLC SSD when `max retry count` is *three*. The performance degradation is not as severe as shown in Figure 6, but the write amplification (Figure 7c) remains similar. Reliability improves, but not all data can be reconstructed fully in the 5K wear state (Figure 7d).

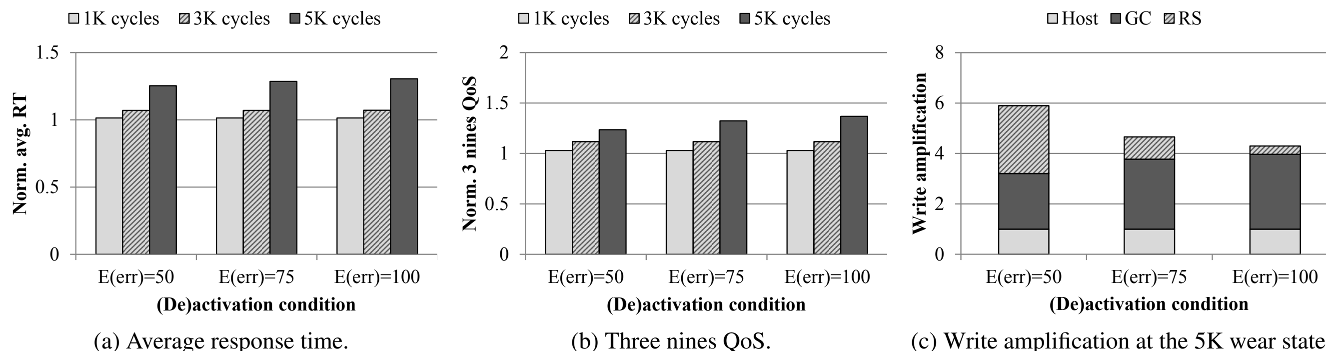
ECC, causing data recovery to fail. Given identical error rates, s=15 should have a higher failure rate as only one error is tolerated within a stripe group. While this is observed in Figure 6d in the lower 1K wear state, s=7 exhibits failure rates as high as those for s=15 in the higher wear states due to higher RBER.

Setting the `max retry count` to *one* reveals more weakness of intra-SSD redundancy than its strength. In Figure 7, we increase this parameter to *three* and observe the differences. In this setting, the average performance (Figure 7a) show a negligible difference to the scheme relying solely on ECC (cf. Figure 4c, 75-bit ECC), and only the 3 nines QoS (Figure 7b) show a more pronounced difference between s=15 and s=7. In this scenario, the performance changes are due to the increase in traffic for writing parity data, and, as expected, the write amplification measurements in Figure 7c are similar to that of Figure 6c. Most data recovery attempts succeed, but still does not warrant full data reconstruction (Figure 7d): While data are fully recovered in the lower wear states, recovery failures are observed in the higher 5K wear state. Further increasing the `max retry count` suppresses the use of data reconstruction through redundancy. In such cases, the benefits of using redundancy scheme are eliminated while the penalty of accelerated wear and increased write amplification remain.

Unlike the proven-effectiveness in the HDD environment, redundancy in SSDs falls short in our experiments. Compared to the scheme that relies on data re-read mechanisms in § 4.1, it performs no better, accelerates wear through increased write amplification, and, what's worse, may not fully recover data due to correlated failures. We expect correlated failures in SSDs to be more prevalent than in HDDs because of flash memory's history-dependence: the error rate in flash memory is a function of its prior history of operations such as the number of erases, number of reads, and time since its last program, and these values are likely to be similar across blocks within a stripe group. With that said, however, the data re-read mechanism is modeled optimistically in our setting, and in the event of a complete chip or wordline failures, SSDs have no other way to recover data aside from device-internal redundancy.

4.3 Background Scrubbing

We perform a set of experiments that measure the effectiveness of data scrubbing, and for this purpose, we assume an *oracle* data scrubber that knows the expected number of errors[2] for each data. This is possible in simulation (though not feasible in practice) as all the error-related parameters for each physical location in the SSD can be tracked to com-



(a) Average response time. (b) Three nines QoS. (c) Write amplification at the 5K wear state.

Figure 8: Performance and write amplification for the 72-layer TLC SSD using oracle scrubbing. The performances are normalized to an SSD with ∞ ECC strength. The oracle scrubber's (de)activation condition uses the expected number of errors per block. The ECC engine corrects up to 75-bit errors, so the E(err)=50 represents an aggressive scrubber.

pute the RBER at any given point in time. The all-knowing scrubber `activates` once the expected number of errors for any data exceeds a threshold, relocating that data to another location, and `deactivates` when the expected number for all data drops below that threshold. We use the oracle scrubber to illustrate the upper-bound benefits. Similar to § 4.2, we show the results from the 72-layer TLC using 75-bit ECC.

Figure 8 illustrates the average response time, 3 nines QoS, and write amplification of the oracle scrubber with three different trigger conditions. E(err)=50 relocates data most aggressively, while E(err)=100 does so lazily. There is little performance loss for the lower wear states, but for the 5K wear state, the difference between the aggressive and the lazy scrubber can be observed in the 3 nines QoS (Figure 8b). By proactively relocating data, the scrubber avoids the long-tail latencies caused by data re-reads. However, this comes at an increase in write amplification in the high wear states, as illustrated in Figure 8c. This shows the relative amount of write amplification per source, including that caused by the read scrubber. The aggressive scrubber in (E(err)=50) moves more data than garbage collection, resulting in a much higher write amplification; this increases the SSD's internal traffic, adding back some of the long-tail latency it reduced. The lazy counterpart, on the other hand, minimally relocates data.

Scrubbing is not a panacea, but it is more suitable than intra-SSD redundancy for complementing the underlying ECC. The scrubber's performance overhead is less than the redundancy scheme, and the increase in write amplification only occurs towards the end-of-life phase. There are several factors that contribute to our results. First, the out-of-place update for flash amplifies the overhead of garbage collection when using intra-SSD redundancy. Second, the history-dependent error patterns of flash memory work against redundancy because of correlated failures, but they make preventive mechanisms more effective because of error predictability.

4.4 Retention Test

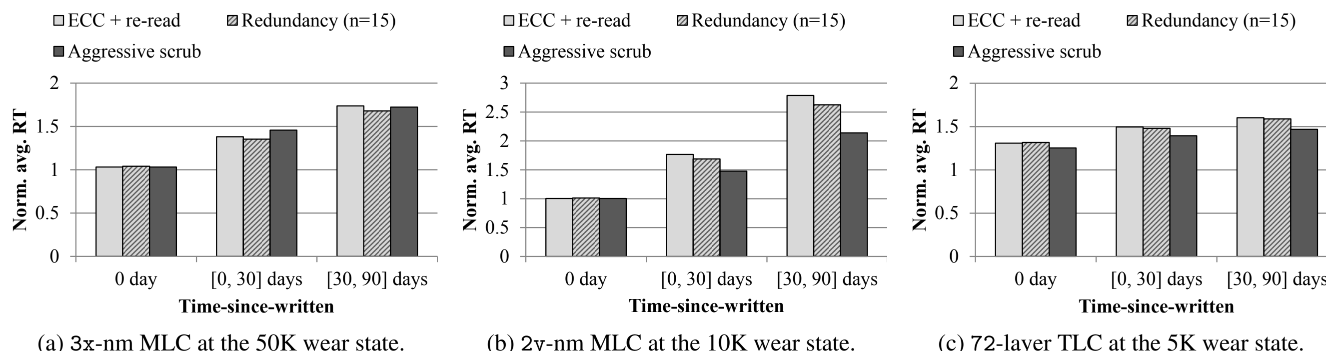
While the experiments so far considered a range of wear states (erase count) and the dynamicity of internal data accesses (read count), the 1 hour experiment is too short to exercise scenarios where data are lost due to retention errors: that is, all the pre-conditioned data are assumed to be written just prior to starting each workload. In this subsection, we explore the effects of data loss due to charge leakage by initializing a non-zero time-since-written value for each data.

Figure 9 shows how the representative error-handling approaches (ECC+re-read of § 4.1, s=15 redundancy of § 4.2, and aggressive scrub of § 4.3) perform when emulating non-zero time-since-written values. SSDs are all at the end-of-life state (50K cycles for the 3x-nm MLC, 10K cycles for the 2y-nm MLC, and 5K cycles for the 72-layer TLC), and they have an ECC correction strength of 4-bits for the 3x-nm, 10-bits for the 2y-nm, and 75-bits for the 72-layer (cf. Figure 4). All performances are normalized to that with an SSD with ∞-bit ECC. We observe that the performance difference between the background scrub approach and others becomes more noticeable. The scrubber proactively relocates data to fresh blocks to prevent upcoming reads from experiencing long-tail latencies. This is particularly more effectively for the 2y-nm MLC that exhibits vulnerability to retention errors (cf. Figure 3b). Compared to the scheme that relies on data re-reads, the aggressive scrubber reduces the performance degradation by 23% for the [30,90] days setting.

4.5 Discussion

We briefly summarize our findings:

- In the high wear states, data re-reads (§ 4.1) severely degrade the performance, increasing the average response time by up to 3.2× when a weak ECC engine is used. Each data re-read further increases the bit error rate that, in turn, cause subsequent accesses to perform more data re-reads.



(a) 3x-nm MLC at the 50K wear state. (b) 2y-nm MLC at the 10K wear state. (c) 72-layer TLC at the 5K wear state.

Figure 9: Average read response time for the three SSDs (all at end-of-life wear state) with various initial time-since-written states. For 0 days, all blocks starts with no retention loss penalty. For [0,30] days, each block starts with an initial time-since-written between 0 and 30 days. Similarly, [30,90] days initializes blocks with values between 30 and 90 days. Performance is normalized to ∞-bit ECC.

- Intra-SSD redundancy (§ 4.2) shows more disadvantages than its merits, in terms of performance, write amplification, and reliability. However, when encountering a random chip and wordline failure, it is the only mechanism to recover data.

- Background scrubbing (§ 4.3) is not a cure-for-all, but is more robust, reducing the performance degradation to as low as 1.25× compared to the ideal no-error scenario even at the end-of-life states. The effectiveness of scrubbing depends on the accuracy of error prediction and internal traffic management. The oracle scrubber circumvents the first issue and reduces the probability of data re-reads, but the created internal traffic degrades performance.

Our experiments, however, are not without limitations. First, the data re-read mechanism we modeled is too optimistic, as it eventually corrects errors given enough re-reads. Because of this, uncorrectable data errors are only observed in the intra-SSD redundancy experiments in the form of recovery failures, while the other experiments are not able to produce such scenarios. A more accurate approach requires an analog model for each flash memory cell, integrated with the SSD-level details such as FTL tasks and flash memory scheduling. Second, the short 1 hour experiments are insufficient to show UBER $< 10^{-15}$. I/Os in the order of petascale are required to experimentally show this level of reliability. Lastly, while Equation 1 models flash memory errors as a function of history-dependent parameters, real flash memories nevertheless exhibit random and sporadic faults. These manifest as not only as chip and wordline failures, but also as runtime bad blocks.

5 Holistic Reliability Management

Our experiments on the effectiveness of existing reliability enhancements across a wide range of SSD states show that there is no *one size fits all* solution. Data re-read mechanism, even though optimistic in our model, not only causes long-tail latencies for that data, but also increase the error rate for other data in the same block. Intra-SSD redundancy, while relevant against random errors, does not offer significant advantages due to its high write amplification. Background data scrubbing, though relatively more robust than other techniques, accelerates wear, and the internal traffic generated by it negates the benefits of error prevention.

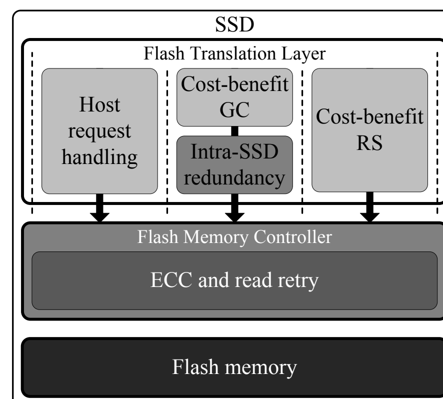


Figure 10: The overall SSD architecture for holistic reliability management. Data written by the garbage collector are protected through redundancy, and read scrubber selects data based on its cost (number of valid data) and benefit (re-read count).

Based on our observation, we propose that these existing techniques should be applied in a coordinated manner as shown in Figure 10. Redundancy should be selectively applied only to infrequently accessed *cold* data to reduce write amplification while providing protection against retention errors. Frequently read *read-hot* data should be relocated through scrubbing to reduce the data re-reads, but the benefit of scrubbing should be compared against the cost of data relocation. Update-frequent *write-hot* data require less attention as it is likely written to a fresh block due to the out-

Table 4: Trace workload characteristics. `Access footprint` is the size of the logical address space accessed, and `Data accessed` is the total amount of data transferred. `Hotness` is the percentage of data transferred in the top 20% of the frequently accessed address.

Workload	Application description	Duration (hrs)	Access footprint (GiB)		Data accessed (GiB)		Hotness (%)	
			Write	Read	Write	Read	Write	Read
DAP-DS	Advertisement caching tier	23.5	0.2	3.5	1.0	40.5	77.8	35.3
DAP-PS	Advertisement payload	23.5	35.1	35.1	42.9	35.2	34.6	20.3
LM-TBE	Map service backend	23.0	192.7	195.5	543.7	1760.0	34.4	45.1
MSN-BEFS	Storage backend file	5.9	30.8	45.8	102.3	193.7	56.9	58.7
MSN-CFS	Storage metadata	5.9	5.7	14.6	14.0	27.0	58.5	56.6
RAD-AS	Remote access authentication	15.3	4.8	1.2	18.7	2.4	63.3	53.1
RAD-BE	Remote access backend	17.0	14.7	8.3	53.3	97.0	49.0	32.7

of-place updates in SSDs. For data classification, we take advantage of SSD's existing mechanisms: data gathered by the read scrubber (RS) is read-hot, while the leftover data selected by the garbage collector (GC) is cold. Write-hot data will be naturally invalided in GC and RS's blocks, and will be re-written to new blocks allocated for host data. This approach for data classification is reactive and conservative as it relies on GC and RS's selection algorithm *after* the data is first written by the host request handler.

The background data scrubbing in § 4.3 used an oracle scrubber that knows the expected error rate for all data. This is impractical in implementation and was only used to illustrate the best-case usage of scrubbing. In the cost-benefit analysis for selecting victims to scrub, the number of valid data is used to represent the cost of relocation. The benefit is the reduction in re-reads after scrubbing, and we use the number of past re-reads for each block since its last erasure as a proxy. That is, if the number of re-reads for a block is large, the potential benefit of scrubbing that block is also large.

5.1 Workload and Test Settings

We use real-world I/O traces from Microsoft production servers [29] to evaluate the representative error-handling approaches and our proposed holistic reliability management (HRM). The traces are modified to fit into the 200GiB range, and all the accesses the aligned to 4KiB boundaries, the same as the mapping granularity for the SSD. Similarly to the synthetic workload evaluation, the logical address range is randomly written to pre-condition the SSD. Table 4 summarizes the trace workload characteristics with particular emphasis on data access pattern. `Access footprint` is the size of the logical address space accessed (of the total 200GiB), and `Data accessed` is the total amount of data transferred. `Hotness` is the percentage of data transferred in the top 20% of the frequently accessed addresses.

We evaluate the following four schemes. Intra-SSD redundancy is omitted as it performed badly due to high write amplification.

∞-bit ECC corrects all errors, and any performance degradation is caused by queueing delays and garbage collection. This represents the baseline performance.

ECC + re-read (§ 4.1) relies on the ECC engine of the flash memory controller, and repeatedly re-reads the data until the error is corrected. 4-bit ECC is used for the 3x-nm SSD, 10-bit ECC for the 2y-nm SSD, and 75-bit for the 72-layer.

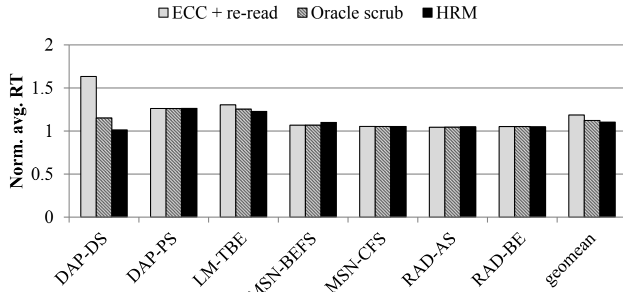
Oracle scrub (§ 4.3) knows the expected number of errors for all data and preventively relocates them before errors accumulate. In an unfortunate event of an ECC failure, it falls back to the `ECC + re-read` approach.

HRM (§ 5) selectively employs redundancy to data gathered by the garbage collector, conditionally re-reads data depending on its redundancy level, and judiciously manages data scrubbing through a cost-benefit analysis.

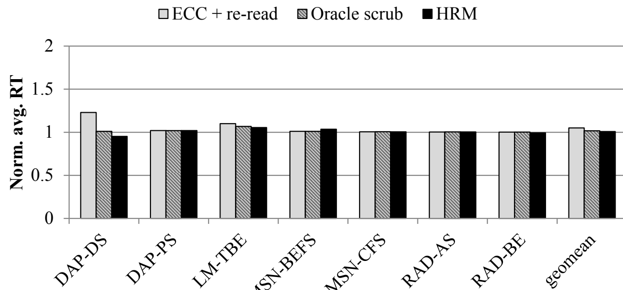
5.2 Experimental Results

Figure 11 shows the performance of ECC+re-read, Oracle scrub, and the proposed HRM, normalized to the performance of ∞-bit ECC on the three SSDs, each at its end-of-life phase. One of the most noticeable results is the performance under `DAP-DS`, which shows that repeated data re-reads severely degrade the performance. `DAP-DS` has a small write footprint (0.2GiB accessed), a high read/write ratio (40.5GiB read vs. 1.0GiB write), and a high write-hotness (77.8% of write data are to 20% of the address). This means that without preventive data relocation, only a small write-hot region will be frequently relocated during garbage collection, and a large region of read-only data suffers from read disturbance. In some cases, HRM even performs better than the baseline, as shown in `DAP-DS` of Figure 11b. This is due to data relocation (unexpectedly) improving parallelism.

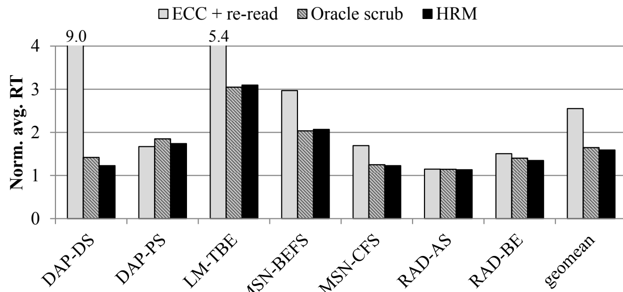
Though marginal, ECC+re-read achieves better performance under `DAP-PS` in Figure 11c. The lack of read skew in the workload (only 20.3% of reads from the top 20% of the address) reduces the effects of read disturbance, as accesses



(a) 3x-nm MLC at the 50K wear state.



(b) 2y-nm MLC at the 10K wear state.



(c) 72-layer TLC at the 5K wear state.

Figure 11: Average read response time on three SSDs, all at their end-of-life phase. In order to ensure data integrity, ECC+re-read adds 18.7%, 5.0%, and 155.1% overhead on average for the 3x-nm MLC, 2y-nm MLC, and 72-layer TLC, respectively. Preventive measures reduce this overhead in general: Oracle scrub adds 12.3%, 1.7%, and 64.5% overhead, and HRM adds 10.4%, 0.9%, and 59.0% overhead, respectively.

are not concentrated to a particular block. On the other hand, workloads with high read skew (LM-TBE, MSN-BEFS, and MSN-CFS) show that the performance degradation can be reduced by preventively relocating data when the read accesses are concentrated. In particular, Oracle scrub outperforms HRM under MSN-BEFS for all three SSDs. This performance gap between Oracle scrub and HRM, however, is very small: 3% at most.

Overall, ECC+re-read adds 18.7%, 5.0%, and 155.1% overhead on average for reliability to the 3x-nm MLC (Figure 11a), 2y-nm MLC (Figure 11b), and 72-layer TLC (Figure 11c), respectively. Oracle scrub that knows the error rate for all data reduces this overhead to 12.3%, 1.7%, and 64.5%, respectively. By judiciously selecting data to relocate, HRM further reduces the overhead to 10.4%, 0.9%, and 59.0%, respectively. In HRM, data not relocated by the scrubber are protected by selective redundancy. Even though Oracle scrub represents the upper-bound benefit of scrubbing, HRM achieves better performance overall by reducing the relocation traffic and delegating the responsibility of data protection for unaccessed data to redundancy.

6 Related Work

To the best of our knowledge, our work first presents a holistic study on the interactions between multiple reliability enhancement techniques and their overall impact on performance in modern SSDs. Our work builds upon a number of prior work from the reliability enhancement techniques to QoS-conscious SSD designs for large-scale deployments.

6.1 Reliability Enhancement

LDPC is widely used in the communications domain and is slowly gaining attention for storage due to its error correction capability. In the context of flash memory-based storages, LDPC-in-SSD [54] reduces the LDPC's long response time by speculatively starting the soft-decision decoding and progressively increasing the iterative memory-sensing level. However, its interaction with other reliability enhancement techniques is not examined.

A series of work exists on threshold voltage prediction for flash memory reads. HeatWatch [38] predicts the change in threshold voltage level caused by the self-recovery effect of flash memory, RDR [13] predicts the changes caused by read disturbance, and ROR [14], by retention error. While these techniques reduce the raw bit error rate by 36%–93.5%, the system-level implications (particularly for QoS performance) are not extensively covered.

Aside from threshold voltage tuning, other tunable voltages exist in flash memory, and several prior work study the performance and reliability tradeoff for these settings. Reducing the read pass-through voltage mitigates the effects of read disturbance [13, 22], and tuning the program voltages tradeoff the flash memory's wear and SSD's write performance [26, 36, 51]. These approaches complement our study and further diversify the system parameters in the performance-reliability spectrum.

Prior work on intra-SSD redundancy techniques focus on reducing the overhead of parity updates [25,34], dynamically managing the stripe size [31,33], and intra-block incremental redundancy [46]. While these approaches are relevant for enhancing the SSD's reliability, we focus on the interaction of these techniques with error correction and error prevention schemes.

In addition to considering multiple reliability enhancement techniques, our work borrows ideas from prior work

on judicious data placement that supplements data protection. LDPC-in-SSD [54] splits a single ECC codeword across multiple memory chips to guard against asymmetric wears; WARM [37] groups write-hot data together and relaxes the management overhead for preventing retention-induced errors; RedFTL [22] identifies frequently read pages and places them in reliable blocks to reduce the overhead of read reclaim. In our work, we holistically cover all causes of errors and study the interactions among multiple reliability enhancements.

Our investigation of SSD's multiple reliability enhancement schemes is inspired by HDD's sector error studies that evaluate intra-HDD redundancy schemes and disk scrubbing techniques [24, 40, 49]. However, we re-assess the effectiveness of these techniques in the context of SSDs for the following three reasons. First, the out-of-place update in SSDs makes intra-SSD redundancy techniques [31,33,46] to be fundamentally different from intra-HDD techniques [17] that allow in-place update of parity data. Second, the existing need for SSD-internal data relocations amortizes the overhead of implementing read reclaim/refresh inside the SSD, while disk scrubbing for HDDs requires external management [6, 45]. Lastly, error patterns in flash memory are history-dependent (number of erase, time-since-written, and number of reads), and can be monitored and controlled to manage the error rate; this is in contrast to errors in HDD that are mostly random events (though temporally and spatially bursty) [8, 45, 49].

6.2 QoS Performance

Improving the QoS performance of SSDs is of great interest in large-scale systems [16, 21, 23], and few recent work suggest several methods in designing SSDs with short tail latencies. AutoSSD [30] dynamically manages the intra-SSD housekeeping tasks (such as garbage collection and read scrubbing) using a control theoretic-approach for a stable performance state, and RLGC [28] schedules garbage collection by predicting the host's idle time using reinforcement learning. ttFlash [53] exploits the existing intra-SSD redundancy scheme and reconstructs data when blocked by SSD-internal tasks to improve QoS performance. We expect QoS-aware scheduling to become increasingly important as more flash memory quirks are introduced, and reliability management in conjunction with scheduling is a central design decision for reducing tail latencies.

Despite the various efforts at the SSD device-level for QoS performance, large-scale systems nevertheless replicate data across devices, servers, data centers for responsiveness and fault-tolerance [16]. Thus, *fail-fast* SSDs are desirable over *fail-slow* ones under such circumstances so that replicated data are instead retrieved. This has culminated in the proposal of *read recovery level* [4] that allows a configurable tradeoff between QoS performance and device error rate. Such tunable service-level agreement between the system and the device further necessitates an comprehensive reliability management.

The increasingly unreliable trend of flash memory incites large production environments to independently study the failure patterns of SSDs [39, 44, 50]. While these studies provide valuable insight on correlating SSD failures and monitored information (such as erase counts, number of reads, amount of data written), they do not directly address how SSD reliability enhancement techniques should be constructed internally.

7 Conclusion

In this work, we examine the design tradeoffs of the existing reliability enhancement techniques in SSDs across multiple dimensions such as performance, write amplification, and reliability. Our findings show that existing solutions exhibit both strengths and weaknesses, and based on our observations, we propose a reliability management scheme that selectively applies appropriate techniques to different data.

There are several research directions that need further investigation. First, the limitation of our study reveals the necessity to integrate the SSD-level design framework (FTL and flash controller) and memory cell-level models that accurately describe electron distributions. Second, there exists a need to mathematically model the effectiveness of data rereads and data scrubbing, so that device reliability can be demonstrated without petascale I/O workloads.

Acknowledgment

We thank our shepherd, Peter Desnoyers, and the anonymous reviewers for their constructive and insightful comments. This work was supported in part by SK Hynix and the Basic Research Laboratory Program through the National Research Foundation of Korea (NRF-2017R1A4A1015498). The Institute of Computer Technology at Seoul National University provided the research facilities for this study.

Notes

[1] The dataset for the `2y`-nm MLC comes from two different papers by the same author. The author informed us that the two papers used *different* chips of the *same* manufacturer. Due to lack of publically available RBER data of similar generation, however, we assume that the two chips are similar enough to create a representative `2y`-nm MLC. One paper provided error rate as a function of wear and time-since-written, while the other, as a function of wear and read disturbance.

[2] During ECC checks, errors are generated randomly using the binomial probability distribution, but the expected number of errors for the oracle scrubber is deterministically computed using the RBER at that moment. This means that ECC may fail even if the expected number of errors is below the `correction strength`, and *vice versa*.

References

[1] The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101). http://www.pdl.cmu.edu/PDL-FTP/DriveChar/CMU-PDL-08-101.pdf, 2008. Parallel Data Laboratory.

[2] Jedec ssd endurance workloads. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2011/20110810_T1B_Cox.pdf, 2011. Flash Memory Summit.

[3] Webplotdigitizer. https://automeris.io/WebPlotDigitizer/, 2011. Ankit Rohatgi.

[4] Solving latency challenges with NVM express SSDs at scale. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170809_SIT6_Petersen.pdf, 2017. Flash Memory Summit.

[5] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M. S., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *2008 USENIX Annual Technical Conference, USENIX ATC 2008, Boston, MA, USA, June 22-27, 2008.* (2008), pp. 57–70.

[6] AMVROSIADIS, G., OPREA, A., AND SCHROEDER, B. Practical scrubbing: Getting to the bad sector at the right time. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012* (2012), pp. 1–12.

[7] ARITOME, S. *NAND flash memory technologies*. John Wiley & Sons, 2015.

[8] BAIRAVASUNDARAM, L. N., GOODSON, G. R., PASUPATHY, S., AND SCHINDLER, J. 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, SIGMETRICS 2007, San Diego, California, USA, June 12-16, 2007* (2007), pp. 289–300.

[9] CAI, Y., GHOSE, S., HARATSCH, E. F., LUO, Y., AND MUTLU, O. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE 105*, 9 (2017), 1666–1704.

[10] CAI, Y., GHOSE, S., LUO, Y., MAI, K., MUTLU, O., AND HARATSCH, E. F. Vulnerabilities in MLC NAND flash memory programming: Experimental analysis, exploits, and mitigation techniques. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017* (2017), pp. 49–60.

[11] CAI, Y., HARATSCH, E. F., MUTLU, O., AND MAI, K. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012* (2012), pp. 521–526.

[12] CAI, Y., HARATSCH, E. F., MUTLU, O., AND MAI, K. Threshold voltage distribution in MLC NAND flash memory: characterization, analysis, and modeling. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013* (2013), pp. 1285–1290.

[13] CAI, Y., LUO, Y., GHOSE, S., AND MUTLU, O. Read disturb errors in MLC NAND flash memory: Characterization, mitigation, and recovery. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25, 2015* (2015), pp. 438–449.

[14] CAI, Y., LUO, Y., HARATSCH, E. F., MAI, K., AND MUTLU, O. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015* (2015), pp. 551–563.

[15] CAI, Y., YALCIN, G., MUTLU, O., HARATSCH, E. F., CRISTAL, A., ÜNSAL, O. S., AND MAI, K. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *30th International IEEE Conference on Computer Design, ICCD 2012, Montreal, QC, Canada, September 30 - Oct. 3, 2012* (2012), pp. 94–101.

[16] DEAN, J., AND BARROSO, L. A. The tail at scale. *Commun. ACM 56*, 2 (2013), 74–80.

[17] DHOLAKIA, A., ELEFTHERIOU, E., HU, X., ILIADIS, I., MENON, J., AND RAO, K. K. A new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. *TOS 4*, 1 (2008), 1:1–1:42.

[18] GALLAGER, R. G. Low-density parity-check codes. *IRE Trans. Information Theory 8*, 1 (1962), 21–28.

[19] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: anomalies, observations, and applications. In *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42) 2009, December 12-16, 2009, New York, New York, USA* (2009), pp. 24–33.

[20] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The bleak future of NAND flash memory. In *10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012* (2012), p. 2.

[21] GUNAWI, H. S., SUMINTO, R. O., SEARS, R., GOLLIHER, C., SUNDARARAMAN, S., LIN, X., EMAMI, T., SHENG, W., BIDOKHTI, N., MCCAFFREY, C., 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 *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018.* (2018), pp. 1–14.

[22] HA, K., JEONG, J., AND KIM, J. An integrated approach for managing read disturbs in high-density NAND flash memory. *IEEE Trans. on CAD of Integrated Circuits and Systems 35*, 7 (2016), 1079–1091.

[23] HAO, M., SOUNDARARAJAN, G., KENCHAMMANA-HOSEKOTE, D. R., CHIEN, A. A., AND GUNAWI, H. S. 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.* (2016), pp. 263–276.

[24] ILIADIS, I., HAAS, R., HU, X., AND ELEFTHERIOU, E. Disk scrubbing versus intra-disk redundancy for high-reliability raid storage systems. In *2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2008, Annapolis, MD, USA, June 2-6, 2008* (2008), pp. 241–252.

[25] IM, S., AND SHIN, D. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Trans. Computers 60*, 1 (2011), 80–92.

[26] JEONG, J., HAHN, S. S., LEE, S., AND KIM, J. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *12th USENIX conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA, February 17-20, 2014* (2014), pp. 61–74.

[27] JIMENEZ, X., NOVO, D., AND IENNE, P. Wear unleveling: improving NAND flash lifetime by balancing page endurance. In *12th USENIX conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA, February 17-20, 2014* (2014), pp. 47–59.

[28] KANG, W., SHIN, D., AND YOO, S. Reinforcement learning-assisted garbage collection to mitigate long-tail latency in SSD. *ACM Trans. Embedded Comput. Syst. 16*, 5 (2017), 134:1–134:20.

[29] KAVALANEKAR, S., WORTHINGTON, B. L., ZHANG, Q., AND SHARDA, V. Characterization of storage workload traces from production Windows servers. In *4th International Symposium on Workload Characterization (IISWC) 2008, Seattle, Washington, USA, September 14-16, 2008* (2008), pp. 119–128.

[30] KIM, B. S., YANG, H. S., AND MIN, S. L. AutoSSD: an autonomic SSD architecture. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.* (2018), pp. 677–690.

[31] KIM, J., LEE, E., CHOI, J., LEE, D., AND NOH, S. H. Chip-level RAID with flexible stripe size and parity placement for enhanced SSD reliability. *IEEE Trans. Computers 65*, 4 (2016), 1116–1130.

[32] KIM, J., LEE, J., CHOI, J., LEE, D., AND NOH, S. H. Improving SSD reliability with RAID via elastic striping and anywhere parity. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013* (2013), pp. 1–12.

[33] LEE, S., LEE, B., KOH, K., AND BAHN, H. A lifespan-aware reliability scheme for RAID-based flash storage. In *2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011* (2011), pp. 374–379.

[34] LEE, Y., JUNG, S., AND SONG, Y. H. FRA: a flash-aware redundancy array of flash storage devices. In *7th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2009, Grenoble, France, October 11-16, 2009* (2009), pp. 163–172.

[35] LIN, S., AND COSTELLO, D. J. *Error Control Coding*. Prentice-Hall, Inc., 2004.

[36] LIU, R., YANG, C., AND WU, W. Optimizing NAND flash-based SSDs via retention relaxation. In *10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012* (2012), p. 11.

[37] LUO, Y., CAI, Y., GHOSE, S., CHOI, J., AND MUTLU, O. WARM: Improving NAND flash memory lifetime with write-hotness aware retention management. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015* (2015), pp. 1–14.

[38] LUO, Y., GHOSE, S., CAI, Y., HARATSCH, E. F., AND MUTLU, O. HeatWatch: Improving 3D NAND flash memory device reliability by exploiting self-recovery and temperature awareness. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018* (2018), pp. 504–517.

[39] MEZA, J., WU, Q., KUMAR, S., AND MUTLU, O. A large-scale study of flash memory failures in the field. In *2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Portland, OR, USA, June 15-19, 2015* (2015), pp. 177–190.

[40] MI, N., RISKA, A., SMIRNI, E., AND RIEDEL, E. Enhancing data availability in disk drives through background activities. In *38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings* (2008), pp. 492–501.

[41] MICHELONI, R. *3D Flash Memories*. Springer, 2016.

[42] MICHELONI, R., CRIPPA, L., AND MARELLI, A. *Inside NAND Flash Memories*. Springer, 2010.

[43] MIELKE, N. R., FRICKEY, R. E., KALASTIRSKY, I., QUAN, M., USTINOV, D., AND VASUDEVAN, V. J. Reliability of solid-state drives based on NAND flash memory. *Proceedings of the IEEE 105*, 9 (2017), 1725–1750.

[44] NARAYANAN, I., WANG, D., JEON, M., SHARMA, B., CAULFIELD, L., SIVASUBRAMANIAM, A., CUTLER, B., LIU, J., KHESSIB, B. M., AND VAID, K. SSD failures in datacenters: What? when? and why? In *9th ACM International on Systems and Storage Conference, SYSTOR 2016, Haifa, Israel, June 6-8, 2016* (2016), pp. 7:1–7:11.

[45] OPREA, A., AND JUELS, A. A clean-slate look at disk scrubbing. In *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010* (2010), pp. 57–70.

[46] PARK, H., KIM, J., CHOI, J., LEE, D., AND NOH, S. H. Incremental redundancy to reduce data retention errors in flash-based SSDs. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015* (2015), pp. 1–13.

[47] PATTERSON, D. A., GIBSON, G. A., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (RAID). In *1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988.* (1988), pp. 109–116.

[48] PRINCE, B. *Vertical 3D memory technologies*. John Wiley & Sons, 2014.

[49] SCHROEDER, B., DAMOURAS, S., AND GILL, P. Understanding latent sector errors and how to protect against them. In *8th USENIX Conference on File and Storage Technologies, FAST 2010, San Jose, CA, USA, February 23-26, 2010* (2010), pp. 71–84.

[50] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.* (2016), pp. 67–80.

[51] SHI, L., QIU, K., ZHAO, M., AND XUE, C. J. Error model guided joint performance and endurance optimization for flash memory. *IEEE Trans. on CAD of Integrated Circuits and Systems 33*, 3 (2014), 343–355.

[52] SUN, H., GRAYSON, P., AND WOOD, B. Quantifying reliability of solid-state storage from multiple aspects. In *7th IEEE Storage Networking Architecture and Parallel I/O, SNAPI 2011, Denver, CO, USA, May 23-27, 2011.* (2011).

[53] YAN, S., LI, H., HAO, M., TONG, M. H., SUNDARARAMAN, S., CHIEN, A. A., AND GUNAWI, H. S. 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* (2017), pp. 15–28.

[54] ZHAO, K., ZHAO, W., SUN, H., ZHANG, T., ZHANG, X., AND ZHENG, N. LDPC-in-SSD: making advanced error correction codes work effectively in solid state drives. In *11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013* (2013), pp. 243–256.

Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts

Taejin Kim, Duwon Hong, Sangwook Shane Hahn[†], Myoungjun Chun,
Sungjin Lee[‡], Jooyoung Hwang[*], Jongyoul Lee[*], and Jihong Kim

Seoul National University, [†]Western Digital, [‡]DGIST, [*]Samsung Electronics

Abstract

Multi-streamed SSDs can significantly improve both the performance and lifetime of flash-based SSDs when their streams are properly managed. However, existing stream management solutions do not adequately support the multi-streamed SSDs for their wide adoption. No existing stream management technique works in a fully automatic fashion for general I/O workloads. Furthermore, the limited number of available streams makes it difficult to effectively manage streams when a large number of streams are required. In this paper, we propose a *fully automatic* stream management technique, PCStream, which can work efficiently for *general* I/O workloads with *heterogeneous* write characteristics. PCStream is based on the key insight that stream allocation decisions should be made on dominant I/O activities. By identifying dominant I/O activities using program contexts, PCStream fully automates the whole process of stream allocation within the kernel with no manual work. In order to overcome the limited number of supported streams, we propose a new type of streams, internal streams, which can be implemented at low cost. PCStream can effectively double the number of available streams using internal streams. Our evaluations on real multi-streamed SSDs show that PCStream achieves the same efficiency as highly-optimized manual allocations by experienced programmers. PCStream improves IOPS by up to 56% over the existing automatic technique by reducing the garbage collection overhead by up to 69%.

1 Introduction

In flash-based SSDs, garbage collection (GC) is inevitable because NAND flash memory does not support in-place updates. Since the efficiency of garbage collection significantly affects both the performance and lifetime of SSDs, garbage collection has been extensively investigated so that the garbage collection overhead can be reduced [1, 2, 3, 4, 5, 6]. For example, hot-cold separation techniques are commonly used inside an SSD so that quickly invalidated pages are not mixed with long-lived data in the same block. For more efficient garbage collection, many techniques also exploit host-level I/O access characteristics which can be used as useful hints on the efficient data separation inside the SSD [7, 8].

Multi-streamed SSDs provide a special interface mechanism for a host system, called streams[1]. With the stream interface, data separation decisions *on the host level* can be delivered to SSDs [9, 10]. When the host system assigns two data D_1 and D_2 to different streams S_1 and S_2, respectively, a multi-streamed SSD places D_1 and D_2 in different blocks, which belong to S_1 and S_2, respectively. When D_1 and D_2 have distinct update patterns, say, D_1 with a short lifetime and D_2 with a long lifetime, allocating D_1 and D_2 to different streams can be helpful in minimizing the copy cost of garbage collection by separating hot data from cold data. Since data separation decisions can be made more intelligently on the host level over on the SSD level, when streams are properly managed, they can significantly improve both the performance and lifetime of flash-based SSDs [10, 11, 12, 13, 14]. We assume that a multi-streamed SSD supports m+1 streams, S_0, ..., S_m.

In order to maximize the potential benefit of multi-streamed SSDs in practice, several requirements need to be satisfied both for stream management and for SSD stream implementation. First, stream management should be supported in a fully automatic fashion over general I/O workloads without any manual work. For example, if an application developer should manage stream allocations *manually* for a given SSD, multi-streamed SSDs are difficult to be widely employed in practice. Second, stream management techniques should have no dependency on the number of available streams. If stream allocation decisions have some dependence on the number of available streams, stream allocation should be modified whenever the number of streams in an SSD changes. Third, the number of streams supported in an SSD should be sufficient to work well with multiple concurrent I/O workloads. For example, with 4 streams, it

[1] In this paper, we use "streams" and "external streams" interchangeably.

would be difficult to support a large number of I/O-intensive concurrent tasks.

Unfortunately, to the best of our knowledge, no existing solutions for multi-streamed SSDs meet all these requirements. Most existing techniques [10, 11, 12, 13] require programmers to assign streams at the application level with manual code modifications. AutoStream [14] is the only known automatic technique that supports stream management in the kernel level without manual stream allocation. However, since AutoStream predicts data lifetimes using the update frequency of the logical block address (LBA), it does not work well with append-only workloads (such as RocksDB [15] or Cassandra [16]) and write-once workloads (such as a Linux kernel build). Unlike conventional in-place update workloads where data are written to the same LBAs often show strong update locality, append-only or write-once workloads make it impossible to predict data lifetimes from LBA characteristics such as the access frequency.

In this paper, we propose a *fully-automatic* stream management technique, called PCStream, which works efficiently over general I/O workloads including append-only, write-once as well as in-place update workloads. The key insight behind PCStream is that stream allocation decisions should be made at a higher abstraction level where *I/O activities*, not LBAs, can be meaningfully distinguished. For example, in RocksDB, if we can tell whether the current I/O is a part of logging activity or a compaction activity, stream allocation decisions can be made a lot more efficiently over when only LBAs of the current I/O is available.

In PCStream, we employ a write program context[2] as such a higher-level classification unit for representing I/O activity regardless of the type of I/O workloads. A program context (PC) [17, 18], which uniquely represents an execution path of a program up to a write system call, is known to be effective in representing dominant I/O activities [19]. Furthermore, most dominant I/O activities tend to show distinct data lifetime characteristics. By identifying dominant I/O activities using program contexts during run time, PCStream can automate the whole process of stream allocation within the kernel with no manual work. In order to seamlessly support various SSDs with different numbers of streams, PCStream groups program contexts with similar data lifetimes depending on the number of supported streams using the k-means clustering algorithm [20]. Since program contexts focus on the semantic aspect of I/O execution as a lifetime classifier, not on the low-level details such as LBAs and access patterns, PCStream easily supports different I/O workloads regardless of whether it is update-only or append-only.

Although many program contexts show that their data lifetimes are narrowly distributed, we observed that this is not necessarily true because of several reasons. For example, when a single program context handles multiple types of data with different lifetimes, data lifetime distributions of such program contexts have rather large variances. In PCStream, when such a program context PC_j is observed (which was mapped to a stream S_k), the long-lived data of PC_j are moved to a different stream $S_{k'}$ during GC. The stream $S_{k'}$ prevents the long-lived data of the stream S_k from being mixed with future short-lived data of the stream S_k.

When several program contexts have a large variance in their data lifetimes, the required number of total streams can quickly increase to distinguish data with different lifetimes. In order to effectively increase the number of streams, we propose a new stream type, called an internal stream, which can be used only for garbage collection. Unlike external streams, internal streams can be efficiently implemented at low cost without increasing the SSD resource budget. In the current version of PCStream, we create the same number of internal streams as the external streams, effectively doubling the number of available streams.

In order to evaluate the effectiveness of PCStream, we have implemented PCStream in the Linux kernel (ver. 4.5) and extended a Samsung PM963 SSD to support internal streams. Our experimental results show that PCStream can reduce the GC overhead as much as a manual stream management technique while requiring no code modification. Over AutoStream, PCStream improves the average IOPS by 28% while reducing the average write amplification factor (WAF) by 49%.

The rest of this paper is organized as follows. In Section 2, we review existing stream management techniques. Before describing PCStream, its two core components are presented in Sections 3 and 4. Section 5 describes PCStream in detail. Experimental results follow in Section 6, and related work is summarized in Section 7. Finally, we conclude with a summary and future work in Section 8.

2 Limitations of Current Practice in Multi-Streamed SSDs

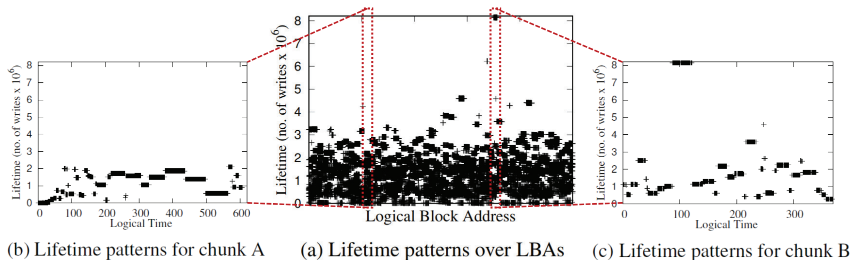
In this section, we review the key weaknesses of existing stream management techniques as well as stream implementation methods. PCStream was motivated to overcome these weaknesses so that multi-streamed SSDs can be widely employed in practice.

2.1 No Automatic Stream Management for General I/O Workloads

Most existing stream management techniques [10, 11, 12] require programmers to manually allocate streams for their applications. For example, in both ManualStream[3] [10] and [11], there is no systematic guideline on how to allocate

[2] Since we are interested in write-related system calls such as `write()` in Linux, we use *write program contexts* and *program contexts* interchangeable where no confusion arises.

[3] For brevity, we denote the manual stream allocation method used in [10] by ManualStream.



(b) Lifetime patterns for chunk A (a) Lifetime patterns over LBAs (c) Lifetime patterns for chunk B

Fig. 1: Lifetime distributions of append-only workload over addresses and times.

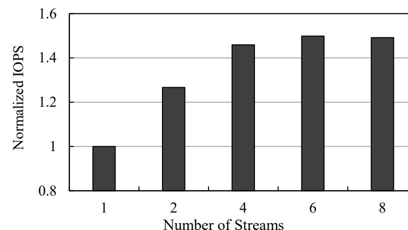


Fig. 2: IOPS changes over the number of streams.

streams for a given application. The efficiency of stream allocations largely depends on the programmer's understanding and expertise on data temperature (*i.e.*, frequency of updates) and internals of database systems. Furthermore, many techniques also assume that the number of streams is known *a priori*. Therefore, when an SSD with a different number of streams is used, these techniques need to re-allocate streams manually. vStream [12] is an exception to this restriction by allocating streams to virtual streams, not external streams. However, even in vStream, virtual stream allocations are left to programmer's decisions.

Although FStream [13] and AutoStream [14] may be considered as automatic stream management techniques, their applicability is quite limited. FStream [13] can be useful for separating file system metadata but it does not work for the user data separation. AutoStream [14] is the only known technique that works in a fully automatic fashion by making stream allocation decisions within the kernel. However, since AutoStream predicts data lifetimes using the access frequency of the same LBA, AutoStream does not work well when no apparent *locality* on LBA accesses exists in applications. For example, in recent data-intensive applications such as RocksDB [15] and Cassandra [16], the majority of data are written in an append-only manner, thus no LBA-level locality can be detected inside an SSD.

In order to illustrate a mismatch between an LBA-based data separation technique and append-only workloads, we analyzed the write pattern of RocksDB [15], which is a popular key-value store based on the LSM-tree algorithm [21]. Fig. 1(a) shows how LBAs may be related to data lifetimes in RocksDB. We define the lifetime of data as the interval length (in terms of the logical time based on the number of writes) between when the data is first written and when the data is invalidated by an overwrite or a TRIM command [22]. As shown in Fig. 1(a), there is no strong correlation between LBAs and their lifetimes in RocksDB.

We also analyzed if the lifetimes of LBAs change under some predictable patterns over time although the overall lifetime distribution shows large variances. Figs. 1(b) and 1(c) show scatter plots of data lifetimes over the logical time for two specific 1-MB chunks with 256 pages. As shown in Figs. 1(b) and 1(c), for the given chunk, the lifetime of data written to the chunk varies in an unpredictable fashion. For example, at the logical time 10 in Fig. 1(b), the lifetime was 1 but it increases about 2 million around the logical time 450 followed by a rapid drop around the logical time 500. Our workload analysis using RocksDB strongly suggests that under append-only workloads, LBAs are not useful in predicting data lifetimes reliably. In practice, the applicability of LBA-based data separation techniques is quite limited to a few cases only when the LBA access locality is obvious in I/O activities such as updating metadata files or log files. In order to support *general* I/O workloads in an automatic fashion, stream management decisions should be based on higher-level information which does not depend on lower-level details such as write patterns based on LBAs.

2.2 Limited Number of Supported Streams

One of the key performance parameters in multi-streamed SSDs is the number of available streams in SSDs. Since the main function of streams is to separate data with different lifetimes so that they are not mixed in the same block, it is clear that the higher the number of streams, the more efficient the performance of multi-streamed SSDs. For example, Fig. 2 shows how IOPS in RocksDB changes as the number of streams increases on a Samsung PM963 multi-streamed SSD with 9 streams. The db_bench benchmark was used for measuring IOPS values with streams manually allocated. As shown in Fig. 2, the IOPS is continuously improving until 6 streams are used when dominant I/O activities with different data lifetimes are sufficiently separated. In order to support a large number of streams, both the SBC-4 and NVMe revision 1.3, which define the multi-stream related specifications, allow up to 65,536 streams [9, 23]. However, the number of streams supported in commercial SSDs is quite limited, say, 4 to 16 [10, 11, 14], because of several implementation constraints on the backup power capacity and fast memory size.

These constraints are directly related to a write buffering mechanism that is commonly used in modern SSDs. In order to improve the write throughput while effectively hiding the size difference between the FTL mapping unit and the flash program unit, host writes are first buffered before they are written to flash pages in a highly parallel fashion for high performance. Buffering host writes temporarily inside

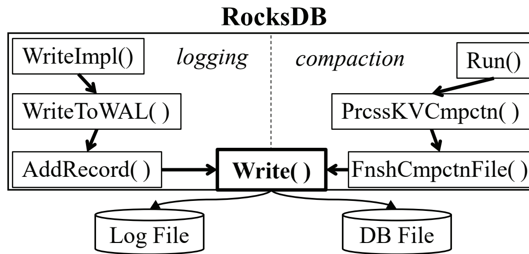


Fig. 3: An illustration of (simplified) execution paths of two dominant I/O activities in RocksDB.

SSDs, however, presents a serious data integrity risk for storage systems when a sudden power failure occurs. In order to avoid such critical failures, in data centers or storage servers where multi-streamed SSDs are used, SSDs use tantalum or electrolytic capacitors as a backup power source. When the main power is suddenly failed, the backup power is used to write back the buffered data reliably. Since the capacity of backup power is limited because of the limited PCB size and its cost, the maximum amount of buffered data is also limited. In multi-streamed SSDs where each stream needs its own buffered area, the amount of buffered data increases as the number of streams increases. The practical limit in the capacity of backup power, therefore, dictates the maximum number of streams as well.

The limited size of fast memory, such as TCM [24] or SRAM, is another main hurdle in increasing the number of streams in multi-streamed SSDs. Since multi-stream related metadata which includes data structures for the write buffering should be accessed quickly as well as frequently, most SSD controllers implement data structures for supporting streams on fast memory over more common DRAM. Since the buffered data is the most recent one for a given LBA, each read request needs to check if the read request should be served from the buffered data or not. In order to support a quick checkup of buffered data, probabilistic data structures such as a bloom filter can be used along with other efficient data structures, for accessing LBA addresses of buffered data and for locating buffer starting addresses. Since the latency of a read request depends on how fast these data structures can be accessed, most SSDs place the buffering-related data structure on fast memory. Similarly, in order to quickly store buffered data in flash chips, these data structure should be placed on fast memory as well. However, most SSD manufacturers are quite sensitive in increasing the size of fast memory because it may increase the overall SSD cost. The limited size of fast memory, unfortunately, restricts the number of supported streams quite severely.

3 Automatic I/O Activity Management

In developing an efficient data lifetime separator for general I/O workloads, our key insight was that in most applications, the overall I/O behavior of applications is decided by a few dominant I/O activities (*e.g.*, logging and flushing in RocksDB). Moreover, data written by dominant I/O activities tend to have distinct lifetime patterns. Therefore, if such dominant I/O activities of applications can be automatically detected and distinguished each other in an LBA-*oblivious* fashion, an automatic stream management technique can be developed for widely varying I/O workloads including append-only workloads.

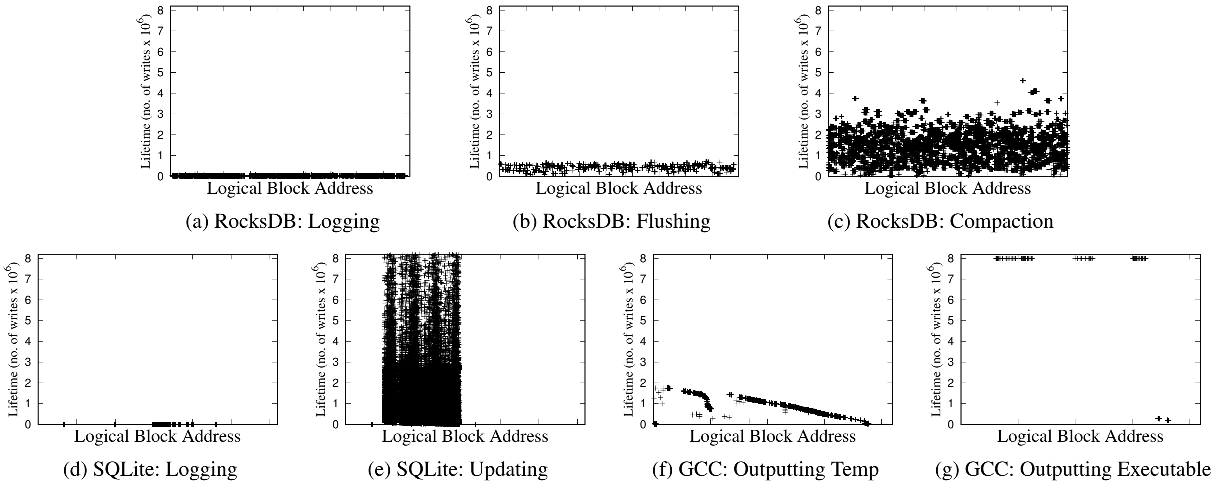
In this paper, we argue that a program context can be used to build an efficient general-purpose classifier of dominant I/O activities with different data lifetimes. Here, a PC represents an execution path of an application which invokes write-related system call functions such as `write()` and `writev()`. There could be various ways of extracting PCs, but the most common approach [17, 18] is to represent each PC with its PC signature which is computed by summing program counter values of all the functions along the execution path which leads to a write system call.

3.1 PC as a Unit of Lifetime Classification

In order to illustrate that using PCs is an effective way to distinguish I/O activities of an application and their data lifetime patterns, we measured data lifetime distributions of PCs from various applications with different I/O workloads. In this section, we report our evaluation results for three applications with distinct I/O activities: RocksDB [15], SQLite [25], and GCC [26]. RocksDB shows the append-only workload while SQLite shows a workload that updates in place. Both database workloads are expected to have distinct I/O activities for writing log files and data files. GCC represents an extensive compiler workload (*e.g.*, compiling a Linux kernel) that generates many short-lived temporary files (*e.g.*, `.s`, `.d`, and `.rc` files) as well as some long-lived files (*e.g.*, object files and kernel image files).

In RocksDB, dominant I/O activities include logging, flushing, and compaction. Since these I/O activities are invoked through different function-call paths, we can easily identify dominant I/O activities of RocksDB using PCs. For example, Fig. 3 shows (simplified) execution paths for logging and compaction in RocksDB. The sum of program counter values of the execution path `WriteImpl()`→`WriteToWAL()`→ `AddRecord()` is used to represent a PC for the logging activity while that of the execution path `Run()`→ `ProcessKeyValueCompaction()` → `FinishCompactionFile()` is used for the compaction activity. In SQLite, there exist two dominant I/O activities which are logging and managing database tables. Similar to the RocksDB, SQLite writes log files and database files using different execution paths. In GCC, there exist many dominant I/O activities of creating various types of temporal files and object files.

To confirm our hypothesis that data lifetimes can be distinguished by tracking dominant I/O activities and a PC is



(a) RocksDB: Logging (b) RocksDB: Flushing (c) RocksDB: Compaction

(d) SQLite: Logging (e) SQLite: Updating (f) GCC: Outputting Temp (g) GCC: Outputting Executable

Fig. 4: Data lifetime distributions of dominant I/O activities in RocksDB, SQLite and GCC.

a useful unit of classification for different I/O activities, we have analyzed how well PCs work for RocksDB, SQLite and GCC. Fig. 4 shows data lifetime distributions of dominant I/O activities which were distinguished by computed PC values. As expected, Fig. 4 validates that dominant I/O activities show distinct data lifetime distributions over the logical address space. For example, as shown in Figs. 4(a)∼4(c), the logging activity, the flushing activity and the compaction activity in RocksDB clearly exhibit quite different data lifetime distributions. While the logged data written by the logging activity have short lifetimes, the flushed data by the flushing activity have little bit longer lifetimes. Similarly, for SQLite and GCC, dominant I/O activities show quite distinct data lifetime characteristics as shown in Figs. 4(d)∼4(g). As shown in Fig. 4(d), the logging activity of SQLite generates short-lived data. This is because SQLite overwrites logging data in a small and fixed storage space and then removes them soon. Lifetimes of temporary files generated by GCC are also relatively short as shown in Fig. 4(f), because of the write-once pattern of temporary files. But, unlike the other graphs in Fig. 4, data lifetime distributions of Figs. 4(c) and 4(e), which correspond to the compaction activity of RocksDB and the updating activity of SQLite, respectively, show large variances. These *outlier I/O activities* need a special treatment, which will be described in Section 4.

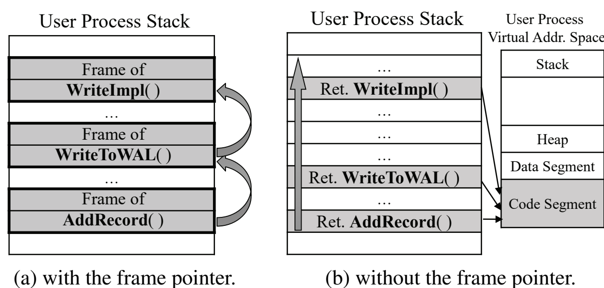
Note that if we used an LBA-based data separator instead of the proposed PC-based scheme, most of data lifetime characteristics shown in Fig. 4 could not have been known. Only the data lifetime distribution of the logging activity of SQLite, as shown in Fig. 4(d), can be accurately captured by the LBA-based data separator. For example, the LBA-based data separator cannot decide that the data lifetime of data produced from the outputting temp activity of GCC is short because temporary files are not overwritten each time they are generated during the compiling step.

3.2 Extracting PCs

As mentioned earlier, a PC signature, which is used as a unique ID of each program context, is defined to be the sum of program counters along the execution path of function calls that finally reaches a write-related system function. In theory, program counter values in the execution path can be extracted in a relatively straightforward manner. Except for inline functions, every function call involves pushing the address of the next instruction of a caller as a return address to the stack, followed by pushing a frame pointer value. By referring to frame pointers, we can back-track stack frames of a process and selectively get return addresses for generating a PC signature. Fig. 5(a) illustrates a stack of RocksDB corresponding to Fig. 3, where return addresses are pushed before calling `write()`, `AddRecord()` and `WriteToWAL()`. Since frame pointer values in the stack hold the addresses of previous frame pointers, we can easily obtain return addresses and accumulate them to compute a PC signature.

The frame pointer-based approach for computing a PC signature, however, is not always possible because modern C/C++ compilers often do not use a frame pointer for improving the efficiency of register allocation. One example is a `-fomit-frame-pointer` option of GCC [26]. This option enables to use a frame pointer as a general-purpose register for performance but makes it difficult for us to back-track return addresses along the call chains.

We employ a simple but effective workaround for back-tracking a call stack when a frame pointer is not available. When a write system call is made, we scan every word in the stack and check if it belongs to process's code segment. If the scanned stack word holds a value within the address range of the code segment, it assumes that it is a return address. Fig. 5(b) shows the scanning process. Since scanning the entire stack may take too long, we stop the scanning step once a sufficient number of return address candidates are found. The larger the return address candidates, the longer the com-



(a) with the frame pointer. (b) without the frame pointer.

Fig. 5: Examples of PC extraction methods.

putation time. On the other hand, if the number of return addresses is too small, two different paths can be regarded as the same path. When the minimum number of addresses that can distinguish the two paths of the program is found, the scanning should be stopped to minimize the scanning overhead. In our evaluation, five return addresses were enough to distinguish execution paths.

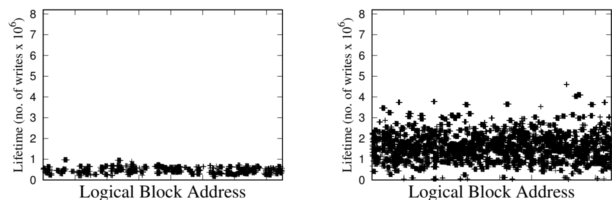
Even though it is quite ad-hoc, this restricted scan is quite effective in distinguishing different PCs because it is very unlikely that two different PCs reach the same `write()` system call through the same execution subpath that covers five proceeding function calls. In our evaluation on a PC with 3.4 GHz Intel CPU, the overhead of the restricted scan was almost negligible, taking only 300∼400 *n*sec per `write()` system call.

4 Support for Large Number of Streams

The number of streams is restricted to a small number because of the practical limits on the backup power capacity and the size of fast memory. Since the number of supported streams critically impacts the overall performance of multi-streamed SSDs, in this section, we propose a new type of streams, called *internal streams*, which can be supported without affecting the capacity of a backup power as well as the size of fast memory in SSDs. Internal streams, which are restricted to be used only for garbage collection, significantly improve the efficiency of PC-based stream allocation, especially when PCs show large lifetime variances in their data lifetime distributions.

4.1 PCs with Large Lifetime Variances

For most PCs, their lifetime distributions tend to have small variances (*e.g.*, Figs. 4(a), 4(d), and 4(f)). However, we observed that it is inevitable to have a few PCs with large lifetime variances because of several practical reasons. For example, when multiple I/O contexts are covered by the same execution path, the corresponding PC may represent several I/O contexts whose data lifetimes are quite different. Such a case occurs, for example, in the compaction job of RocksDB.



(a) RocksDB: L2 Compaction (b) RocksDB: L4 Compaction

Fig. 6: Lifetime distributions of the compaction activity at different levels.

RocksDB maintains several levels, L1, ..., L*n*, in the persistent storage, except for L0 (or a memtable) stored in DRAM. Once one level, say L2, becomes full, all the data in L2 is compacted to a lower level (*i.e.*, L3). It involves moving data from L2 to L3, along with the deletion of the old data in L2. In the LSM tree [21], a higher level is smaller than a lower level (*i.e.*, the size of (L2) $<$ the size of (L3)). Thus, data stored in a higher level is invalidated more frequently than those kept in lower levels, thereby having shorter lifetimes.

Unfortunately, in the current RocksDB implementation, the compaction step is supported by the same execution path (*i.e.*, the same PC) regardless of the level. Therefore, the PC for the compaction activity cannot effectively separate data with short lifetimes from one with long lifetimes. Fig. 6(a) and 6(b) show distinctly different lifetime distributions based on the level of compaction: data written from the level 4 have a large lifetime variance while data written from the level 2 have a small lifetime variance.

Similarly, in SQLite and GCC, program contexts with large lifetime variations are also observed. Fig. 4(e) shows large lifetime variances of data files in SQLite. Since client request patterns will decide how SQLite updates its tables, the lifetime of data from the updating activity of SQLite is distributed with a large variance. Similarly, the lifetime of data from the outputting temporary files of GCC can significantly fluctuate as well depending on when the next compile step starts. Fig. 4(g) shows long lifetimes of object files/executable files after a Linux build was completed (with no more re-compiling jobs). However, the lifetime of the same object files/executable files may become short when if we have to restart the same compile step right after the previous one is finished (*e.g.*, because of code changes).

For these *outlier* PCs with large lifetime variations, it is a challenge to allocate streams in an efficient fashion unless there are more application-specific hints (*e.g.*, the compaction level in RocksDB) are available. As an ad-hoc (but effective) solution, when a PC shows a large variance in its data lifetime, we allocate an additional stream, called an internal stream, to the PC so that the data written from the PC can be better separated between the original stream and its internal stream. In order to support internal streams, the total number of streams may need to be doubled so that each stream can be associated with its internal stream.

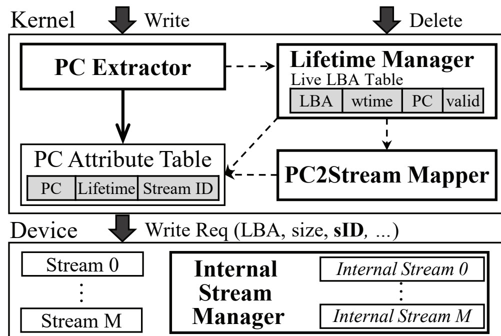


Fig. 7: An overall architecture of PCStream.

4.2 Implementation of Internal Streams

As described in Section 2.2, it is difficult to increase the number of (normal) streams. However, if we restrict that internal streams are used only for data movements during GC, they can be quite efficiently implemented without the constraints on the backup power capacity and fast memory size. The key difference in the implementation overhead between normal streams and internal streams comes from a simple observation that data copied during GC do not need the same reliability and performance support as for host writes. Unlike buffered data from host write requests, valid pages in the source block during garbage collection have no risk of losing their data from the sudden power-off conditions because the original valid pages are always available. Therefore, even if the number of internal streams increases, unlike normal streams, no higher-capacity backup capacitor is necessary for managing buffered data for internal streams.

The fast memory requirement is also not directly increased as the number of internal streams increases. Since internal streams are used only for GC and most GC can be handled as background tasks, internal streams have a less stringent performance requirement. Therefore, data structures for supporting internal streams can be placed on DRAM without much performance issues. Furthermore, for a read request, there is no need to check if a read request can be served by buffered data as in normal streams because the source block always has the most up-to-date data. This, in turn, allows data structures for internal streams to be located in slow memory. Once an SSD reaches the fully saturated condition where host writes and GC are concurrently performed, the performance of GC may degrade a little because of the slow DRAM used for internal streams. However, in our evaluation, such cases were rarely observed under a reasonable overprovisioning storage capacity.

5 Design and Implementation of PCStream

In this section, we explain the detailed implementation of PCStream. Fig. 7 shows an overall architecture of PCStream. The *PC extractor* is implemented as part of a kernel's system call handler as already described in Section 3, and is responsible for computing a PC signature from applications. The PC signature is used for deciding the corresponding stream ID[4] from the PC attribute table. PCStream maintains various per-PC attributes in the PC attribute table including PC signatures, expected data lifetimes, and stream IDs. In order to keep the PC attribute table updated over changing workloads, the computed PC signature with its LBA information is also sent to the *lifetime manager*, which estimates expected lifetimes of data belonging to given PCs. Since commercial multi-streamed SSDs only expose a limited number of streams to a host, the *PC2Stream mapper* groups PCs with similar lifetimes using a clustering policy, assigning PCs in the same group to the same stream. Whenever the lifetime manager or the PC2Stream mapper are invoked, the PC attribute table is updated with new outputs from these modules. Finally, the *internal stream manager*, which was implemented inside an SSD as firmware, is responsible for handling internal streams associated with external streams.

5.1 PC Lifetime Management

The responsibility of the lifetime manager is for estimating the lifetime of data associated with a PC. Except for outlier PCs, most data from the same PC tend to show similar data lifetimes with small variances.

Lifetime estimation: Whenever a new write request R arrives, the lifetime manager stores the write request time, the PC signature, PC_i, and the LBA list of R into the live LBA table. The live LBA table, indexed by an LBA, is used in computing the lifetime of data stored at a given LBA which belongs to PC_i. Upon receiving TRIM commands (that delete previously written LBAs) or overwrite requests (that update previously written LBAs), the lifetime manager searches the live LBA table for a PC signature PC_{found} with the LBA list which includes the deleted/updated LBAs. The new lifetime l_{new} of PC_{found} is estimated using the lifetime of the matched LBA from the live LBA table. The average of the existing lifetime l_{old} for PC_{found} and l_{new} is used to update the PC_{found} entry in the PC attribute table. Note that the written time entry of the live LBA table is updated differently depending on TRIM commands or overwrite requests. The written time entry becomes invalid for TRIM while it is updated by the current time for an overwrite request.

Maintaining the live LBA table, which is indexed by an LBA unit, in DRAM could be a serious burden owing to its huge size. In order to mitigate the DRAM memory requirement, the lifetime manager slightly sacrifices the accuracy of computing LBA lifetime by increasing the granularity of LBA lifetime prediction to 1 MB, instead of 4 KB. The live LBA table is indexed by 1 MB LBA, and each table entry holds PC signatures and written times over a 1 MB LBA range. For example, for a 256 GB SSD, 4 KB-granularity

[4] We call i the stream ID of S_i.

requires 4 billion entries while 1 MB-granularity requires 16 million entries. For a 9 byte-sized entry, LBA table requires about 144 MB memory. Due to the coarse-grained mapping, multiple requests within an address unit are considered as requests to the same address, which are updates. Therefore, the data lifetime can be recognized shorter than the real lifetime. However, even if long-lived data are misallocated to the short lifetime stream, the internal stream effectively suppresses the increase in WAF.

PC attribute table: The PC attribute table keeps PC signatures and its expected lifetimes. To quickly retrieve the expected lifetime of a requested PC signature, the PC attribute table is managed through a hash data structure. Each hash entry requires only 12 bytes: 64-bit for a PC signature and 32-bit for a predicted lifetime. The table size is thus small so that it can be entirely loaded in DRAM. From our evaluations, the maximum number of unique PCs was up to 30. So the DRAM size of the PC attribute table was sufficient with 360 KB.

In addition to the main function of the PC attribute table that maintains the data lifetime for a PC, the *memory – resident* PC attribute table has another interesting benefit for the efficient stream management. Since a PC signature of an I/O activity is virtually guaranteed to be *globally unique* across *all* applications (the uniqueness property), and a PC signature does not change over different executions of the same application (the consistency property), the PC attribute table can capture a long-term history of programs' I/O behaviors. Because of the uniqueness and consistency of a PC signature, `PCStream` can exploit the I/O behavior of even short-lived processes (*e.g.*, `cpp` and `cc1` for GCC) that are launched and terminated frequently. When short-lived processes are frequently executed, the PC attribute table can hold their PC attributes from their previous executions, thus enabling quick but accurate stream allocation for short-lived processes.

The consistency property is rather straightforward because each PC signature is determined by the sum of return addresses inside a process's virtual address space. Unless a program's binary is changed after recompilation, those return addresses remain the same, regardless of the program's execution. The uniqueness property is also somewhat obvious from the observation that the probability that distinct I/O activities that take different function-call paths have the same PC signature is extremely low. This is even true for multiple programs. Even though they are executed in the same virtual address space, it is very unlikely that I/O activities of diverged programs taking different function-call paths have the same PC. In our experiment, there was no alias for the PC value. Consequently, this immutable property of the PC signature for a given I/O activity makes it possible for us to characterize the given I/O activity in a long-term basis without risk of PC collisions.

5.2 Mapping PCs to SSD streams

After estimating expected lifetimes of PC signatures, the PC2Stream mapper attempts to group PCs with similar lifetimes into an SSD stream. This grouping process is necessary because while commercial SSDs only support a limited number of streams (*e.g.,* 9), the number of unique PCs can be larger (*e.g.*, 30). For grouping PCs with similar lifetimes, the PC2Stream mapper module uses the k-means algorithm [20] which is widely used for similar purposes. In `PCStream`, we use the difference in the data lifetime between two PCs as a clustering distance and generates m clusters of PCs for m streams. This algorithm is particularly well suited for our purpose because it is lightweight in terms of the CPU cycle and memory requirement. To quickly assign a proper stream to incoming data, we add an extra field to the PC attribute table which keeps a stream ID for each PC signature. More specifically, when a new write request comes, a designated SSD stream ID is obtained by referring to the PC attribute table using request's PC value as an index. If there is no such a PC in the table, or a PC does not have a designated stream ID, the request gets default stream ID, which is set to 0.

For adapting to changing workloads, re-clustering operations should be performed regularly. This re-clustering process is done in a straightforward manner. The PC2Stream mapper scans up-to-date lifetimes for all PCs in the PC attribute table. Note that PC's lifetimes are updated whenever the lifetime manager gets new lifetimes while handling overwrites or TRIM requests, as explained in Section 5.1. With the scanned information, the PC2Stream mapper recomputes stream IDs and updates stream fields of the PC attribute table. In order to minimize the unnecessary overhead of frequent re-clustering operations, re-clustering is triggered when 10% of the PC lifetime entries in the PC attribute table is changed.

5.3 Internal Stream Management

As explained in Section 4.1, there are a few outlier PCs with large lifetime variances. In order to treat these PCs in an efficient fashion, we devise a two-phase method that decides SSD streams in two levels: the main stream in the host level and its internal stream in the SSD level. Conceptually, long-lived data in the main stream are moved to its internal stream so that (future) short-lived data will not be mixed with long-lived data in the main stream. Although moving data to the internal stream may increase WAF, the overhead can be hidden if we restrict data copies to the internal stream during GC only. Since long-lived data (*i.e.*, valid pages) in a victim block are moved to a free block during GC, blocks belong to an internal stream tend to contain long-lived data. For instance, `PCStream` assigns the compaction-activity PC_1 to the main stream S_1 in the first phase. To separate the long-lived data of PC_1 (*e.g.*, L4 data) from future short-lived data of the

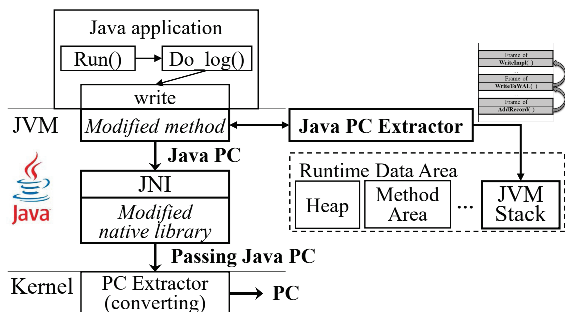


Fig. 8: Extracting PCs for JVM.

same PC_1 (*e.g.*, L1 data), valid pages of the S_1 are assigned to its internal stream for the second phase during GC.

We have implemented the internal stream manager with the two-phase method in Samsung's PM963 SSD [27]. To make it support the two-phase method, we have modified its internal FTL so that it manages internal streams while performing GC internally. Since the internal stream manager assigns blocks for an internal stream and reclaims them inside the SSD, no host interface changed is required.

5.4 PC Extraction for Indirect Writes

One limitation of using PCs to extract I/O characteristics is that it only works with C/C++ programs that *directly* call write-related system calls. Many programs, however, often invoke write system calls *indirectly* through intermediate layers, which makes it difficult to track program contexts.

The most representative example may be Java programs, such as Cassandra, that run inside a Java Virtual Machine (JVM). Java programs invoke write system calls via the Java Native Interface (JNI) [28] that enables Java programs to call a native I/O library written in C/C++. For Java programs, therefore, the PC extractor shown in Fig. 7 fails to capture Java-level I/O activities as it is unable to inspect the JVM stack from the native write system call which is indirectly called through the JNI. Another example is a program that maintains a write buffer that is dedicated to dealing with all the writes from an application. For example, in MySQL [29] and PostgreSQL [30], every write is first sent to a write buffer. Separate flush threads later materialize buffered data to persistent storage. In that case, the PC extractor only captures PCs of flush threads, not PCs of I/O activities that originally generate I/Os, because the I/O activities were executed in different threads using different execution stacks.

The problem of indirect writes can be addressed by collecting PC signatures *at the front-end interface* of an intermediate layer that accepts write requests from other parts of the program. In the case of Java programs, a native I/O library can be modified to capture write requests and computes their PC signatures. Once a native library is modified, PCStream can automatically gather PC signatures without modifying application programs. Fig. 8 illustrates how PCStream collects PC signatures from Java programs. We have modified the OpenJDK [31] source to extract PC signatures for most of the write methods in write related classes, such as `OutputStream`. The stack area in the `Runtime Data Areas` of JVM is used to calculate PC signatures. The calculated PC is then passed to the write system call of the kernel via the modified native I/O libraries. For the JIT compilation, the codes are dynamically compiled and optimized, so the computed PC value of the same path can be different. However, if the code cache space is sufficient, the compiled code is reused, so there is no problem in using the PC. In the experiment, there was enough space in the code cache.

Unlike Java, there is no straightforward way to collect PCs from applications with write buffers. This is because the implementation of write buffering is different depending on applications. Additional efforts to manually modify code are unavoidable. However, the scope of this manual modification is limited only to the write buffering code, and application logics themselves don't need to be edited or annotated. Moreover, in the virtual machine (VM) environment, modification of the VM itself is inevitable. PCStream can get PC of guest OS, but it is difficult to transfer directly to the device. We can transfer PC information to the system call layer of host OS through modification of virtualization layer.

6 Experimental Results

6.1 Experimental Settings

In order to evaluate PCStream, we have implemented it in the Linux kernel (version 4.5) on a PC host with Intel Core i7-2600 8-core processor and 16 GB DRAM. As a multi-streamed SSD, we used Samsung's PM963 480 GB SSD. The PM963 SSD supports up to 9 streams; 8 user-configurable streams and 1 default stream. When no stream is specified with a write request, the default stream is used. To support internal streams, we have modified the existing PM963 FTL firmware. For detailed performance analysis, we built a modified `nvme-cli` [32] tool that can retrieve the internal profiling data from PCStream-enabled SSDs. Using the modified `nvme-cli` tool, we can monitor WAF values and per-block data lifetimes from the extended PM963 SSD during run time.

We compared PCStream with three existing schemes: Baseline, ManualStream [10], and AutoStream [14]. Baseline indicates a legacy SSD that does not support multiple streams. ManualStream represents a multi-streamed SSD with manual stream allocation. AutoStream represents the LBA-based stream management technique proposed in [14].

We have carried out experiments with various benchmark programs which represent distinct write characteristics. RocksDB [15] and Cassandra [16] have append-only write patterns. SQLite [25] has in-place update write patterns and GCC [26] has write-once patterns. For more realistic evalu-

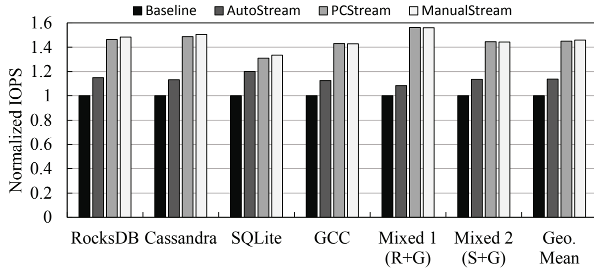


Fig. 9: A comparison of normalized IOPS.

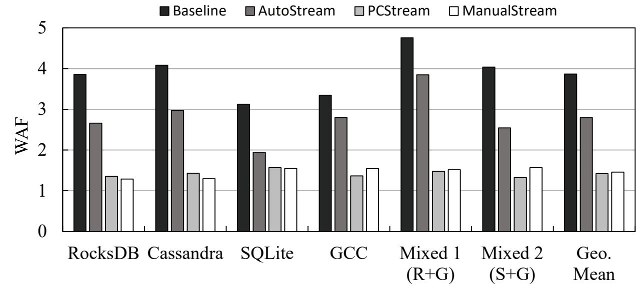


Fig. 10: A comparison of WAF under different schemes.

ations, we also used mixed workloads running two different benchmark programs simultaneously.

In both RocksDB and Cassandra experiments, Yahoo! Cloud Serving Benchmark (YCSB) [33] with 12-million keys was used to generate update-heavy workloads (workload type A) which consist of 50/50 reads and writes. Since both RocksDB and Cassandra are based on the append-only LSM-tree algorithm [21], they have three dominant I/O activities (such as logging, flushing, and compaction). Cassandra is written in Java, so its PC is extracted by the modified procedure described in Section 5.4. In SQLite evaluations, TPC-C [34] was used with 20 warehouses. SQLite has two dominant I/O activities such as logging and updating tables. In GCC experiments, a Linux kernel was built 30 times. For each build, 1/3 of source files, which were selected randomly, were modified and recompiled. Since GCC creates many temporary files (*e.g.*, `.s`, `.d`, and `.rc`) as well as long-lived files (*e.g.*, `.o`) from different compiler tools, there are more than 20 dominant PCs. To generate mixed workloads, we run RocksDB and GCC scenarios together (denoted by Mixed 1), and run SQLite and GCC scenarios at the same time (denoted by Mixed 2). In order to emulate an aged SSD in our experiments, 90% of the total SSD capacity was initially filled up with user files before benchmarks run.

6.2 Performance Evaluation

We compared the IOPS values of three existing techniques with PCStream. Fig. 9 shows normalized IOPS for six benchmarks with four different techniques. For all the measured IOPS values[5], PCStream improved the average IOPS by 45% and 28% over Baseline and AutoStream, respectively. PCStream outperformed AutoStream by up to 56% for complex workloads (*i.e.*, GCC, Mixed1 and Mixed 2) where the number of extracted PCs far exceeds the number of supported streams in PM963. The high efficiency of PCStream under complex workloads comes from two novel features of PCStream: (1) LBA-oblivious PC-centric data separation and (2) a large number of streams supported using internal streams. AutoStream, on the other hands, works poorly except for SQLite where the LBA-based separation can be effective. Even in SQLite, PCStream outperformed AutoStream by 10%.

[5]For RocksDB, Cassandra, and SQLite, the YCSB benchmark and TPC-C benchmark compute IOPS values as a part of the benchmark report. For GCC, where an IOPS value is not measured during run time, we computed the IOPS value as a ratio between the total number of write requests (measured at the block device layer) and the total elapsed time of running GCC.

6.3 WAF Comparison

Fig. 10 shows WAF values of four techniques for six benchmarks. Overall, PCStream was as efficient as ManualStream; Across all the benchmarks, PCStream showed similar WAF values as ManualStream. PCStream reduced the average WAF by 63% and 49% over Baseline and AutoStream, respectively.

As expected, Baseline showed the worst performance among all the techniques. Owing to the intrinsic limitation of LBA-based data separation, AutoStream performs poorly except for SQLite. Since PCStream (and ManualStream) did not depend upon LBAs for stream separations, they performed well consistently, regardless of write access patterns. As a result, PCStream reduced WAF by up to 69% over AutoStream.

One interesting observation in Fig. 10 is that PCStream achieved a lower WAF value than even ManualStream for GCC, Mixed 1, and Mixed 2 where more than the maximum number of streams in PM963 are needed. In ManualStream, DB applications and GCC were manually annotated at offline, so that write system calls were statically bound to specific streams during compile time. When multiple programs run together as in three complex workloads (*i.e.*, GCC, Mixed 1 and Mixed 2), static stream allocations are difficult to work efficiently because they cannot adjust to dynamically changing execution environments. However, unlike ManualStream, PCStream continuously adapts its stream allocations during run time, thus quickly responding to varying execution environments. For example, 10 PCs out of 25 PCs are remapped by 7 reclustering operations for Mixed 1 workload.

6.4 Per-stream Lifetime Distribution Analysis

To better understand the benefit of PCStream on the WAF reduction, we measured per-stream lifetime distributions for the Mixed 1 scenario. Fig. 11 shows a box plot of data lifetimes from the 25th to the 75th percentile. As shown in

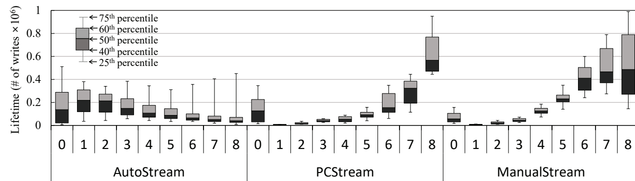


Fig. 11: A Comparison of per-stream lifetime distributions.

Fig. 11, streams in both PCStream and ManualStream are roughly categorized as two groups, $G1 = \{S_1, S_2, S_3, S_4, S_5\}$ and $G2 = \{S_6, S_7, S_8\}$, where $G1$ includes streams with short lifetimes and small variances (*i.e.*, S_1, S_2, S_3, S_4, and S_5) and $G2$ includes streams with large lifetimes and large variances (*i.e.*, S_6, S_7, and S_8). The S_0 does not belong to any groups as it is assigned to requests whose lifetimes are unknown. Even though the variance in the S_0 is wider than that in ManualStream, PCStream showed similar per-stream distributions as ManualStream. In particular, for the streams in $G2$, PCStream exhibited smaller variance than ManualStream, which means that PCStream separates cold data from hot data more efficiently. Since PCStream moves long-lived data of a stream to its internal stream, the variance of streams with large lifetimes tend to be smaller over ManualStream.

AutoStream was not able to achieve small per-stream variances as shown in Fig. 11 over PCStream and ManualStream. As shown in Fig. 11, all the streams have large variances in AutoStream because hot data are often mixed with cold data in the same stream. Since the LBA-based data separation technique of AutoStream does not work well with both RocksDB and GCC, all the streams include hot data as well as cold data, thus resulting in large lifetime variances.

6.5 Impact of Internal Streams

In order to understand the impact of internal streams on different stream management techniques, we compared the two versions of each technique, one with internal streams and the other without internal streams. Since internal streams are used only for GC, they can be combined with any existing stream management techniques. Fig. 12 shows WAF values for five benchmarks with four techniques. Overall, internal streams worked efficiently across the four techniques evaluated. When combined with internal streams, Baseline, AutoStream, PCStream, and ManualStream reduced the average WAF by 25%, 22%, 17%, and 12%, respectively. Since the quality of initial stream allocations in Baseline and AutoStream was relatively poor, their WAF improvement ratios with internal streams were higher over PCStream and ManualStream. Although internal streams were effective in separating short-lived data from long-lived data in both Baseline and AutoStream, the improvement from internal streams in these techniques is not sufficient to outperform PCStream and ManualStream. Poor initial stream allocations, which keep putting both hot and cold data to the same stream, unfortunately, offset a large portion of benefits from internal streams.

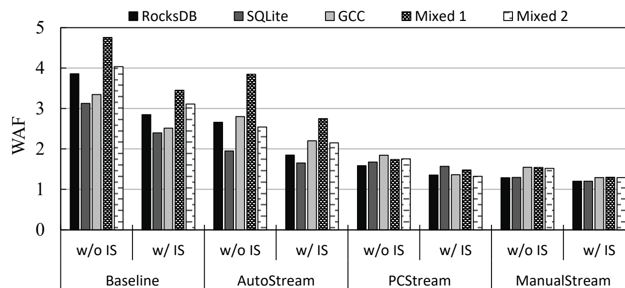


Fig. 12: The effect of internal streams on WAF.

6.6 Impact of the PC Attribute Table

As explained in Section 5, the PC attribute table is useful to maintain a long-term history of applications' I/O behavior by exploiting the uniqueness of a PC signature across different applications. To evaluate the effect of the PC attribute table on the efficiency of PCStream, we modified the implementation of the PC attribute table so that the PC attribute table can be selectively disabled on demands when a process terminates its execution. For example, in the kernel compilation scenario with GCC, the PC attribute table becomes empty after each kernel build is completed. That is, the next kernel build will start with no existing PC to stream mappings.

Fig. 13 show how many requests are assigned to the default S_0 stream over varying sizes of the PC attribute table. Since S_0 is used when no stream is assigned for an incoming write request, the higher the ratio of requests assigned to S_0, the less effective the PC attribute table. As shown in Fig. 13, in RocksDB, Cassandra, and SQLite, the PC attribute table did not affect much the ratio of writes on S_0. This is because these programs run continuously for a long time while performing the same dominant activities repeatedly. Therefore, although the PC attribute table is not maintained, they can quickly reconstruct it. On the other hand, the PC attribute table was effective for GCC, which frequently creates and terminates multiple processes (*e.g.*, `cc1`). When no PC attribute table was used, about 16% of write requests were assigned to S_0. With the 4-KB PC attribute table, this ratio was re-

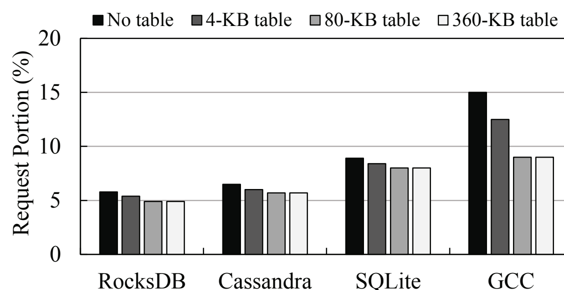


Fig. 13: The effect of the PC attribute table.

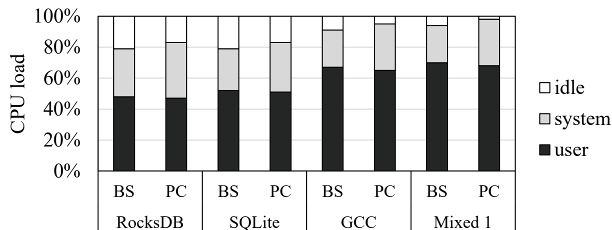


Fig. 14: A comparison of cpu load.

duced to 12%. With the 360-KB PC attribute table, only 9% of write requests were assigned to S_0. This reduction in the S_0 allocation ratio reduced the WAF value from 1.96 to 1.54.

6.7 CPU Overhead Evaluation

As described in Sections 3 and 5, PCStream requires additional CPU usage to compute and clustering PCs. We used the sar command in linux to evaluate the additional CPU load on PCStream. Fig. 14 shows the CPU utilization of the baseline (BS) and PCStream (PC) technique. The percentage of CPU utilization that occurred while executing at the user level and kernel level was represented by *user* and *system*, respectively. *idle* indicates the percentage of time that the CPU was idle. For all cases, the increased CPU load due to PCStream was less than 5%.

7 Related Work

There have been many studies for multi-streamed SSDs [10, 11, 12, 13, 14, 35]. Kang *et al.* first proposed a multi-streamed SSD that supported manual stream allocation for separating different types of data [10]. Yang *et al.* showed that a multi-streamed SSD was effective for separating data of append-only applications like RocksDB [11]. Yong *et al.* presented a virtual stream management technique that allows logical streams, not physical streams, to be allocated by applications. Unlike these studies that involve modifying the source code of target programs, PCStream automates the stream allocation with no manual code modification.

Yang *et al.* presented an automatic stream management technique at the block device layer [14]. Similar to hot-cold data separation technique used in FTLs, it approximates the data lifetime of data based on update frequencies of LBAs. The applicability of this technique is, however, quite limited to in-place update workloads only. PCStream has no such limitation on the workload characteristics, thus effectively working for general I/O workloads including append-only, write-once as well as in-place update workloads.

Ha *et al.* proposed an idea of using PCs to separate hot data from cold one in an FTL layer [19]. Kim *et al.* extended it for multi-streamed SSDs [35]. Unlike these works, our study treats the PC-based stream management problem in a more complete fashion by (1) pinpointing the key weaknesses of existing multi-streamed SSD solutions, (2) extending the effectiveness of PCs for more general I/O workloads including write-once patterns, and (3) introducing internal streams as an effective solution for outlier PCs. Furthermore, PCStream exploits the globally unique nature of a PC signature for supporting short-lived applications that run frequently.

8 Conclusions

We have presented a new stream management technique, PCStream, for multi-streamed SSDs. Unlike existing techniques, PCStream fully automates the process of mapping data to a stream based on PCs. Based on observations that most PCs are effective to distinguish lifetime characteristics of written data, PCStream allocates each PC to a different stream. When a PC has a large variance in their lifetimes, PCStream refines its stream allocation during GC and moves the long-lived data of the current stream to the corresponding internal stream. Our experimental results show that PCStream can improve the IOPS by up to 56% over the existing automatic technique while reducing WAF by up to 69%.

The current version of PCStream can be extended in several directions. First, PCStream does not support applications that rely on a write buffer (*e.g.,* MySQL). To address this, we plan to extend PCStream interfaces so that developers can easily incorporate PCStream into their write buffering modules with minimal efforts. Second, we have only considered write-related systems calls to collect PCs, but many applications (*e.g.,* MonetDB [36]) heavily access files with mmap-related functions (*e.g.*, `mmap()` [37] and `msync()`). We plan to extend PCStream to work with mmap-intensive applications.

Acknowledgments

We thank Youjip Won, our shepherd, and the anonymous reviewers for their valuable feedback and comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (NRF-2015M3C4A7065645 and NRF-2018R1A2B6006878). The ICT at Seoul National University provided research facilities for this study. Sungjin Lee was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2018R1A5A1060031, NRF-2017R1E1A1A01077410) (Corresponding Author: Jihong Kim)

References

[1] M. Chiang, P. Lee, R. and Chang, Using Data Clustering to Improve Cleaning Performance for Flash Memory, *Software-Practice & Experience, vol. 29, no. 3, pp. 267-290*, 1999.

[2] X. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, Write Amplification Analysis in Flash-Based Solid State Drives, In *Proceedings of the ACM International Systems and Storage Conference (SYSTOR)*, 2009

[3] W. Bux, and I. Iliadis, Performance of Greedy Garbage Collection in Flash-Based Solid-State Drives, *Performance Evaluation, vol. 67, no. 11, pp. 1172-1186*, 2010.

[4] C. Tsao, Y. Chang, and M. Yang, Performance Enhancement of Garbage Collection for Flash Storage Devices: An Efficient Victim Block Selection Design, In *Proceedings of the Annual Design Automation Conference (DAC)*, 2013.

[5] S. Yan, H. Li, M. Hao, M. Tong, S. Sundararaman, A. Chien, and H. Gunawi, Tiny-tail Flash: Near-perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs, In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[6] J. Hsieh, T. Kuo, and L. Chang, Efficient Identification of Hot Data for Flash Memory Storage Systems, *ACM Transactions on Storage, vol. 2, no. 1, pp. 22-40*, 2006.

[7] S. Hahn, S. Lee, and J. Kim, To Collect or Not to Collect: Just-in-Time Garbage Collection for High-Performance SSDs with Long Lifetimes, In *Proceedings of the Design Automation Conference (DAC)*, 2015.

[8] J. Cui, Y. Zhang, J. Huang, W. Wu, and J. Yang, ShadowGC: Cooperative Garbage Collection with Multi-Level Buffer for Performance Improvement in NAND flash-based SSDs, In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2018.

[9] SCSI Block Commnads-4 (SBC-4), `http://www.t10.org/cgi-bin/ac.pl?t=f&f=sbc4r15.pdf`.

[10] J. Kang, J. Hyun, H. Maeng, and S. Cho, The Multi-streamed Solid-State Drive, In *Proceedings of the Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[11] F. Yang, D. Dou, S. Chen, M. Hou, J. Kang, and S. Cho, Optimizing NoSQL DB on Flash: A Case Study of RocksDB, In *Proceedings of IEEE the International Conference on Scalable Computing and Communications (ScalCom)*, 2015.

[12] H. Yong, K. Jeong, J. Lee, J. Kim, vStream: Virtual Stream Management for Multi-streamed SSDs, In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[13] E. Rho, K. Joshi, S. Shin, N. Shetty, J. Hwang, S. Cho. and D. Lee, FStream: Managing Flash Streams in the File System, In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[14] J. Yang, R. Pandurangan, C. Chio, and V. Balakrishnan, AutoStream: Automatic Stream Management for Multi-streamed SSDs, In *Proceedings of the ACM International Systems and Storage Conference (SYSTOR)*, 2017.

[15] Facebook, `https://github.com/facebook/rocksdb`.

[16] Apache Cassandra, `http://cassandra.apache.org`.

[17] C. Gniady, A. Butt, and Y. Hu, Program-Counter-based Pattern Classification in Buffer Caching, In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[18] F. Zhou, J. Behren, and E. Brewer, Amp: Program Context Specific Buffer Caching, In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2005.

[19] K. Ha, and J. Kim, A Program Context-Aware Data Separation Technique for Reducing Garbage Collection Overhead in NAND Flash Memory, In *Proceedings of International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2011.

[20] J. Hartigan, and M. Wong, Algorithm as 136: A k-means Clustering Algorithm, *Journal of the Royal Statistical Society. Series C (Applied Statistics), vol. 28, no. 1, pp. 100-108*, 1979.

[21] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, The Log-Structured Merge-Tree (LSM-Tree), *Acta Informatica, vol. 33, no. 4, pp. 351-385*, 1996.

[22] J. Corbet, Block Layer Discard Requests, `https://lwn.net/Articles/293658/`.

[23] NVM Express Revision 1.3, `http://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf`.

[24] S. Frank, Tightly Coupled Multiprocessor System Speeds Memory-Access Times, *Electronics, vol. 57, no. 1*, 1984.

[25] SQLite, `https://www.sqlite.org/index.html`.

[26] R. Stallman, and GCC Developer Community, Using the GNU Compiler Collection for GCC version 7.3.0, https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc.pdf.

[27] Samsung, Samsung SSD PM963, https://www.compuram.de/documents/datasheet/Samsung_PM963-1.pdf

[28] S. Liang, Java Native Interface: Programmer's Guide and Specification, 1999.

[29] MySQL, https://www.mysql.com.

[30] PostgreSQL, https://www.postgresql.org.

[31] OpenJDK, http://openjdk.java.net/.

[32] NVM-Express user space tooling for Linux, https://github.com/linux-nvme/nvme-cli.

[33] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, Benchmarking Cloud Serving Systems with YCSB, In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010.

[34] The Transaction Processing Performance Council, Benchmark C, http://www.tpc.org/tpcc/default.asp.

[35] T. Kim, S. Hahn, S. Lee, J. Hwang, J. Lee and J. Kim, PCStream: Automatic Stream Allocation Using Program Contexts, In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[36] MonetDB, https://www.monetdb.org.

[37] Linux Programmer's Manual, mmap(2) - map files or devices into memory, http://man7.org/linux/man-pages/man2/mmap.2.html.

Large-Scale Graph Processing on Emerging Storage Devices

Nima Elyasi[†], Changho Choi[‡], Anand Sivasubramaniam[†]
[†]*The Pennsylvania State University,* [‡]*Samsung Semiconductor Inc.*

Abstract

Graph processing is becoming commonplace in many applications to analyze huge datasets. Much of the prior work in this area has assumed I/O devices with considerable latencies, especially for random accesses, using large amount of DRAM to trade-off additional computation for I/O accesses. However, emerging storage devices, including currently popular SSDs, provide fairly comparable sequential and random accesses, making these prior solutions inefficient. In this paper, we point out this inefficiency, and propose a new graph partitioning and processing framework to leverage these new device capabilities. We show experimentally on an actual platform that our proposal can give 2X better performance than a state-of-the-art solution.

1 Introduction

Graph processing is heavily employed as the fundamental computing platform for analyzing huge datasets in many applications such as social networks, web search, and machine learning. Processing large graphs leads to many random and fine-grained accesses to memory and secondary storage, which is detrimental to application performance. Prior work have attempted to develop optimized frameworks for graph processing either in a distributed system [11, 17, 19, 23, 25] or for a single machine [8, 9, 12, 13, 15, 16, 18, 20, 22, 26], by partially/completely storing the graph data in main memory (DRAM memory).

Recent efforts on single machine approaches aim at storing *Vertex data* in the main memory to serve their fine-grained accesses in the byte-addressable DRAM memory, while the *Edge data* which usually has coarser accesses, is stored in the secondary storage. With growing graph dataset size, even partially storing them on DRAM memory is not cost-effective. On the other hand, emerging storage devices, including currently popular Solid State Drives (SSDs), continue to scale and offer larger capacity with lower access latency, and can be used to accommodate voluminous graph datasets and deliver good performance. However, an SSD's large access granularity (several KB's) is an impediment towards exploiting its substantial bandwidth for graph processing.

Prior works [9, 15] attempt to alleviate this issue by either storing some part of graph data in the main memory or effectively partition the graph data. Such techniques are either designed for the conventional Hard Disk Drives (HDDs) and are not able to saturate an SSD's substantial bandwidth, or are not readily applicable when the vertex data is stored on secondary storage. GraFBoost [13] is a recent fully *external* graph processing framework that stores all graph data on the SSD, and tries to provide global sequentiality for I/O accesses. Despite yielding performance benefits, providing global sequentiality hinders its scalability as graph dataset sizes increase dramatically. On the other hand, since NVMe SSDs deliver comparable performance for random and sequential page-level I/O accesses [2–4], such perfect sequentiality may not be all that essential.

In this paper, we first study the performance issues of external graph processing, and propose a partitioning for vertex data to relax the global sequentiality constraint. More specifically, we address the performance and scalability issues of state-of-the-art external graph processing, where all graph data resides on the SSD. To this end, we devise a partitioning technique for vertex data such that, in each sub-iteration of graph algorithm execution, instead of randomly updating any vertices in the graph, updates occur to only a subset of vertices (which is sufficiently small to fit in main memory).

With our proposed partitioning, after transferring the vertex data associated with a partition into main memory from SSD, the subsequent information required to generate updates –to the vertices present in the memory– will be streamed from SSD. Thus, the fine-grained updates will be only applied to the set of vertices in the memory, eliminating the need for coalescing all the intermediate updates to provide perfect sequentiality. Our proposed enhancements can give more than 2X better performance than a state-of-the-art solution.

2 Background and Related Work

Graph Data Representation: Graphs are represented by (i) *Vertex data* that refers to a set of vertices with vertex attributes including its ID, value, and its neighboring information (i.e., byte offset of its neighbors), and (ii) *Edge data* that contains the set of edges connected to each vertex along with its properties. Edge data is usually stored in a compressed format. A common compressed representation of graph data is called Compressed Sparse Column (CSC) wherein vertex file stores the vertex information along with the byte offset of its neighbors in the edge file.

Programming Model: Due to its unique characteristics, large-scale graph processing is inherently not suited to the parallelism offered by previous parallel programming models. Among different models to facilitate processing of large graphs, *Vertex-Centric* programming model [19] has received much attention, as this iterative model is properly designed to distribute and parallelize large graph analytics. In this model, each vertex runs a *vertex program* which reads its attributes as well as its neighboring vertices, and generates updates to itself and/or its neighbors.

Graph Processing Frameworks: Numerous prior efforts incorporate vertex-centric model and disperse graph data amongst several machines, with each machine storing its portion on DRAM memory, to expedite the fine-grained and random accesses to the graph data. Distributing the graph data, on the other hand, necessitates frequent communications. Such approaches employ various partitioning techniques to minimize the communication overhead [11, 17, 19, 25], and balance the load.

Apart from distributed graph analytic frameworks, single-machine techniques have also been proposed [7–9, 12, 13, 15, 16, 18, 20–22, 26]. When a single machine is used, it may fully or partially store graph data on the secondary storage, and transfer it to main memory in fairly large chunks to achieve high I/O sequentiality. It is common in such techniques to store vertex data on main memory, and edge data on the secondary storage. GraphChi [15], specifically designed for HDDs, splits graph data into different partitions, where partitions are processed consecutively. Their enhancements has two consequences: (i) with increasing graph data size, the number of partitions can proportionally increase, resulting in high I/O costs, and (ii) when only a portion of graph data is required (e.g., when running a sparse graph algorithm), all graph data has to be transferred to main memory. FlashGraph [9] stores vertex data on DRAM memory while edge data resides on an array of SSDs. However, with graph data continuing to grow, even storing the vertex data–which is usually orders of magnitude smaller than edge data– requires considerable amount of expensive DRAM memory. Thus, it is important to consider completely external graph processing approaches.

External Graph Processing: Storing vertex data on SSD has performance and lifetime penalty due to fine-grained I/O accesses. For example, in push-style vertex-centric model, the value of different vertices (e.g., the rank in PageRank algorithm) needs be updated at the end of each iteration. Such updates are usually in the range of a few bytes (e.g., 4 byte integer), whereas the SSD page size is a few kilobytes (e.g., 4KB$\sim$16KB). Apart from its poor performance, an important consequence of the miss-match between the granularity of vertex updates and SSD page size, is its detrimental impact on SSD's endurance.

GraFBoost [13] proposes a sort-reduce scheme to coalesce all the fine-grained updates and submit large and sequential writes to the SSD. In each iteration of GraFBoost after running an *edge program* for the edges connected to a vertex v, a set of updates are generated for the neighbors of v. These updates are in the form of $< key, value >$ pairs, where *key* is the neighbor's ID, and *value* refers to the value of v (source vertex). The number of intermediate updates can be commensurate with the number of edges, denoted as $|E|$, with many duplicate *keys* generated for each destination vertex.

GraFBoost sorts and reduces the $< key, value >$ pairs to convert the fine-grained updates to large sequential SSD writes. Since the number of updates can reach well beyond the size of available DRAM memory, the graph data is streamed from SSD, processed and sorted in main memory in large parts (e.g., 512MB), and then logged on the SSD. Subsequently, these 512MB chunks are streamed from SSD, merge-reduced and written back to the SSD. Despite providing significant benefits, GraFBoost, or any external graph processing approach which tries to provide perfect sequentiality for all vertex updates, incurs high computation overhead. This computation could be avoided for SSDs which provide quite good page-level random access performance, unlike HDDs.

3 Motivation

In this section, we study the performance and scalability issues of GraFBoost, a state-of-the-art external graph processing framework. To investigate its performance, we run various graph algorithms, using different input graphs. For our experiments, we use a system with 48 Intel Xeon cores, 256 GB of DRAM, and two datacenter-scale Samsung NVMe SSDs with 3.2 TB capacity in total, which provide up to 6.4 GB/s sequential Read speed. We run two algorithms, *Breadth First Search* (BFS) and *PageRank* on various input graphs (details can be found in Table 1) including web crawl graph [6], twitter graph [5], and synthetic graphs generated based on Graph 500 benchmark [1]. This synthetic set of input graphs enables us to generate and examine graphs with various size.

Performance Analysis. We give the breakdown of normalized execution time for BFS and PageRank in Figure 1, running on three graphs: `web`, `twitter`, and `kron30`. The latency of different steps of GraFBoost, including (i) reading/writing vertex data, (ii) reading edge data and running edge program, and (ii) the sort-reduce phase, are reported in Figure 1. As

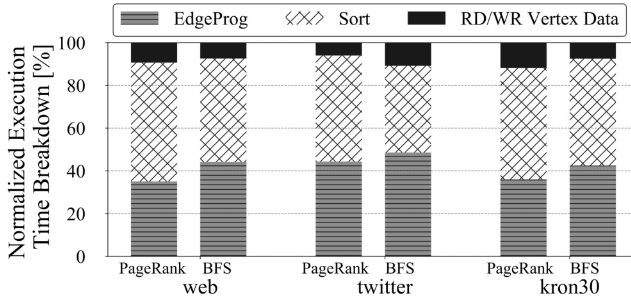


Figure 1: Execution time breakdown of GraFBoost.

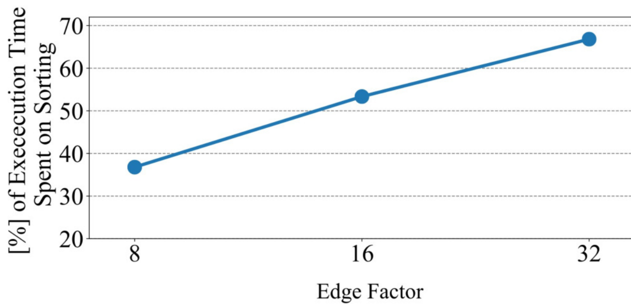


Figure 2: Percentage of execution time spent on sorting.

shown in this figure, the sort phase is the major contributor to the overall execution time, by accounting for nearly 61% of the total execution time for running PageRank on `web` graph. GraFBoost, despite effectively eliminating fine-grained I/O accesses, requires to expend considerable part of its execution time only for the *sorting* phase. In other words, it trades off the additional computation for I/O accesses. This is, in fact, very common in many graph processing frameworks, to minimize the communication/transfer overhead at the expense of adding more computation.

Scalability. To investigate the scalability of GraFBoost, we present a simplified analysis of its execution time. Assuming a graph with N edges, the latency of SSD accesses is linear with respect to N, i.e., $O(N)$. Moreover, Sorting in the memory takes $O(N * log(N))$ to complete, on average. With DRAM access speed k times faster than SSD, if the number of edges grows, such that $log(N) > k$, the sorting phase can dominate the total execution time and hinder its scalability. To quantitatively confirm our analysis, we run PageRank on a synthetic graph with different edge factors (ratio of number of edges to vertices). We report the percentage of time spent in the sort phase, in Figure 2, for `kron` graphs with 1 billion vertices and edge factors of 8∼32. As it is evident, increasing the number of edges results in larger sorting overheads, as more number of updates are generated, which in turn, takes longer time to sort.

Summary. Even though the computation cost that GraFBoost introduces may appear to be an acceptable trade-off for current systems and graph datasets, its benefits are expected to dramatically drop as the graph data sizes grow. Preserving comprehensive sequentiality and sorting of intermediate data, seems to be unnecessary with SSDs providing nearly identical page-level random and sequential access latencies. Instead, if graph vertices can be placed on SSD pages such that each page contains a set of vertices which are likely to be updated at almost the same time, the sorting phase could be eliminated altogether. However, perfectly clustering the graph vertices is known to be an NP-hard problem [14]. In this paper, we aim to provide a local sequentiality which, unlike prior works, does not require any sorting of intermediate updates to achieve lower execution times.

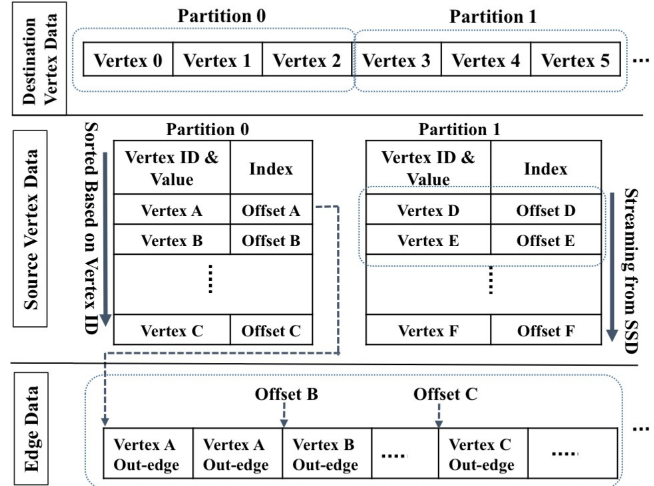


Figure 3: Data structures in our design.

4 Proposed Mechanism

In this section, we describe our proposed partitioning technique that re-organizes vertex data and splits them into several partitions, so that each can fit on a limited DRAM space. The high-level idea is to change the order in which graph vertices are updated, so that at each time, the updates are directed at a subset of vertices residing in main memory. Specifically, we propose to partition the vertex data and process each partition by reading its respective vertex data into main memory, followed by streaming the required edge data from the SSD. Figure 3 shows different data structures employed in our design. Since in each iteration, updates happen to the destination vertices, we (i) split the destination vertices and assign each subset of them to a partition (Destination Vertex Data in this figure), and (ii) store source vertices and their neighboring information –a pointer to the out-edges of each vertex[1]– for each partition, separately (Source Vertex Data in the figure). Lastly, we organize the edge data for each partition as shown in this

[1] e is called an out-edge of vertex u, if $e : u => v$.

figure (Edge Data). Note that, our proposed enhancements are based on the push-style vertex-centric graph processing model.

4.1 Partitioning Vertex Data

There has already been extensive prior work on partitioning graph data. However, they are not well suited for fully external graph processing, due to a number of reasons: (i) some of these studies [9, 15, 18] require all vertex data be present in the main memory when processing the graph, which as prior work [13] shows, they sometimes even fail to finish their execution when the available DRAM space is not enough to store the vertex data; (ii) some others [9, 11, 23, 24, 26] propose 2-D partitioning where graph data is assigned to each partition with the rows/columns corresponding to the source/destination vertices, respectively. These proposals typically do not decouple vertex data from edge data, needing the vertices and edges assigned to a partition be completely present in main memory, or cache, to process it. This constraint results in dramatic rise in the number of partitions which, in turn, accentuates the cross-partition communication overhead. Instead, we devise a mechanism that only requires the vertex data of a partition be present in main memory while edge data can be streamed from SSD, as needed. Based on our proposed greedy partitioning algorithm, destination vertices are uniquely assigned to each partition, whereas source vertices can have duplicates (mirrors) on different partitions. The goal of this greedy partitioning is to minimize the number of mirrors for source vertices while preserving the uniqueness of destination vertices. Based on this partitioning, for each edge $e : u => v$,

- *If* v is already assigned to a partition, u will be added to the same partition, if it does not already exist on that.
- *Else if*, v is not assigned to any partition yet,
  - *If* u is assigned to a set of partitions $\{P_1, P_2, ...\}$, we choose the partition with the least number of edges corresponding to it.
  - *Else*, we assign u and v to the partition with least number of edges corresponding to it.

This partitioning guarantees that each destination vertex is uniquely assigned to a partition and it does not have any mirrors. After this phase, the destination vertex IDs are updated with respect to their new order. These changes are also reflected on the respective source vertices and the edge data. The size of partitions are adjusted such that destination vertices for each partition can fit in main memory. Note that, partitioning is done off-line, as a pre-processing step, latency of which does not impact the execution time.

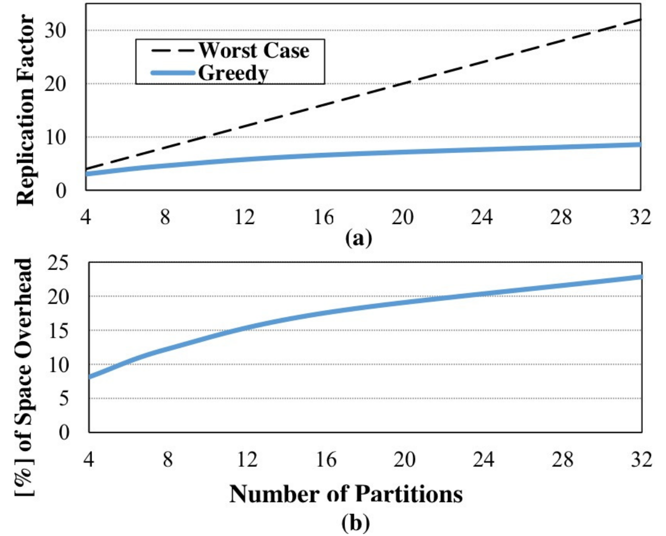


Figure 4: Overhead of the proposed partitioning.

4.1.1 Partitioning Overhead

We study the efficacy of our proposed partitioning, based on the *replication factor*, i.e., the average number of mirrors that each vertex has, and the space overhead. To this end, we run our partitioning algorithm on `twitter` graph, for different number of partitions, and report results in Figure 4. As shown in this figure, with increasing number of partitions, the replication factor increases sub-linearly according to the number of partitions, and it is fairly below the worst case. For instance, with 8 partitions, the replication factor and the space overhead are around 4.5 and 12%, respectively. These overheads happen to be smaller for other graphs listed in Table 1 (3.07 replication factor, on average).

4.2 Execution Model

Different partitions are processed consecutively. For each:
1. The destination vertex data associated with that partition is transferred to main memory from SSD.
2. Source vertex data (their attributes and neighboring information) for this partition, are streamed from SSD in 32 MB chunks. This can be done in parallel with each thread reading different chunks. Decisions regarding which vertex data is currently required to be processed (i.e., is active), can be made either on-the-fly or after the source vertex data is transferred into main memory.
3. After determining the set of active vertices (active vertex list), for each active vertex, byte offset of its neighbors on the edge data file is extracted and the required edges are transferred to main memory. Thus, for a chunk of source data, all the required information to run the graph algorithm exists in main memory, including the source vertex attributes, destination vertices in the current partition, and the neighboring

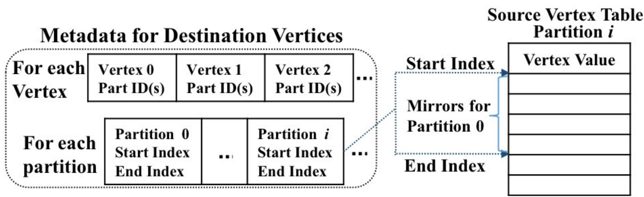


Figure 5: Meta-data for updating mirrors.

```
1.  Read Metadata for Partition P;
2.  For v in P.V:  //P.V: dest. vertex list for P
3.          For part in v.mirror_list_partition:
4.                  out_buf[part].append(v.value);
5.          End_For
6.  End_For
7.  For p_i in {0, Num_Partitions-1}:
8.          Write out_buf[p_i] to SourceTable p_i;
9.  End_For
```

Figure 6: Pseudo code for updating mirrors.

information. This implies that, all the updates generated by this source vertex chunk will happen to the set of destination vertices present in main memory.

4. The graph algorithm runs, and the updates are generated for the destination vertices. As an example, in PageRank, the rank (value) of each source vertex is sent to the destination vertices. The ranks are accumulated in each destination vertex and dampened by a factor specified for this algorithm (e.g., 0.15). In this step, multiple threads are attempting to update elements of the same vertex (destinations) list in memory, which can incur high synchronization cost. Instead, we perform the vertex data updates in two steps: (i) first, threads push updates (in large chunks, e.g., 1MB) to multiple buffers, each dedicated to a portion of the vertex list, and (ii) subsequently, writer threads pull data from these buffers and update their specified portion (similar to Map-Reduce [10] paradigm).

5. When processing for a partition completes, the meta-data (depicted in Figure 5) required for updating its mirrors on other partitions, is read from SSD. The meta-data includes the partition IDs of mirrors of each vertex[2], and the chunk offset for each partition. To minimize the overhead of mirror updates, all source vertex tables store the vertices in the order of their IDs to enables sequential updates to the mirrors.

6. The mirror updates are generated and written on SSD.

[2]We keep Partition IDs in a bitmap to minimize space overhead.

Table 1: Characteristics of the evaluated graph data.

Graph	webgraph	twitter	kron30	kron32
Num Vertices	3.5B	41M	1B	4B
Num Edges	128B	1.47B	17B	32B
Text Size	2.7TB	25GB	351GB	295GB
Rep. Factor	3.7	4.5	1.91	2.2
Space Overhead	10.5%	12%	10.3%	11.5%

4.3 Updating Mirrors

We give pseudo code for mirror updates, in Figure 6. For each vertex in destination vertices, we determine on which partitions its mirrors are located (line 3). In line 4, we insert the value of that vertex to a buffer assigned to destined partition. Lastly, the generated updates are written to the source vertex files on SSD (line 7∼9)[3]. Generating mirrors for different partitions is proportional to the number of destination vertices in each partition, resulting in overall running time of $O(|V|)$, with $|V|$ referring to the number of vertices.

5 Experimental Evaluation

5.1 Evaluation Environment

We evaluate the performance of our proposed mechanism against software version of GraFBoost. GraFBoost also has a hardware implementation, using hardware accelerators. Since the hardware implementation is not available to us, we extract its performance numbers from the original paper [13]. To make a fair comparison, we use the same configuration as GraFBoost: we use 32 processing cores (out of 48 available in our system), 128 GB of memory, and two Samsung NVMe SSDs, totalling 3.2 TB of capacity with nearly identical bandwidth as GraFBoost, i.e., 6.4 GB/s sequential read bandwidth. Similarly, we use the same set of graph data, details of which are reported in Table 1. In this table, we also present the replication factor and space overhead of our partitioning technique, for 8 partitions[4], as it is sufficient for the evaluated graph datasets.

5.2 Evaluation Results

Figure 7 shows the amount of reduction in total execution time (higher is better) for PageRank and BFS, for our proposal (**V-Part**), and software and hardware versions of GraFBoost (**GraFSoft** and **GraFHard**). All performance numbers are normalized to that of GraFSoft. We also show the execution time (in seconds) for PageRank and BFS algorithms, for **GraFSoft** and **V-Part** in Figure 8. As illustrated in these two figures, our proposed partitioning provides better performance

[3]This can be done in parallel for mirror updates on different partitions.
[4]We fix the memory size assigned to a partition's vertex data (e.g., 2GB), and find the proper number of partitions, accordingly.

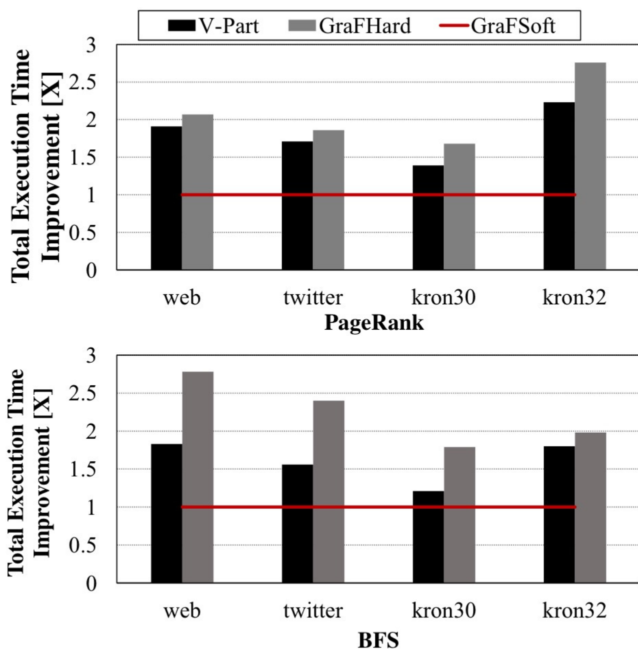


Figure 7: Execution time improvement results.

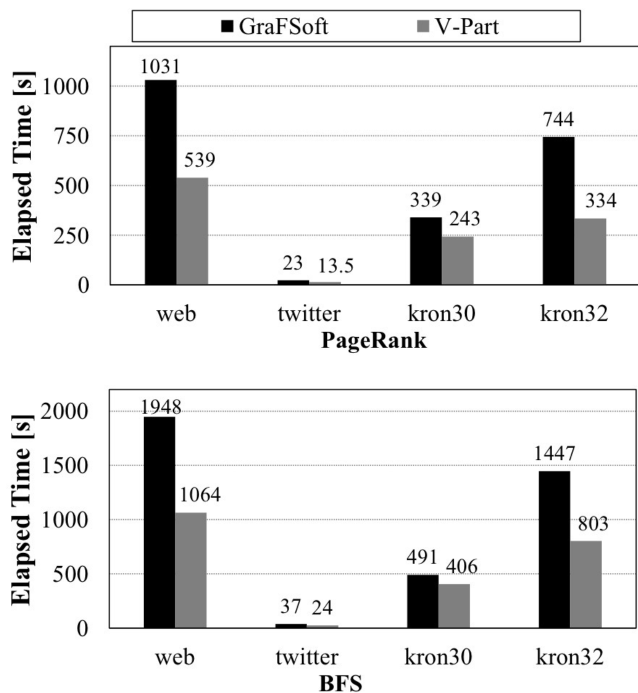


Figure 8: Execution time for (i) a PageRank iteration, and (ii) BFS.

than GraFSoft by around 2.2X (when running PageRank on `kron32`), and 1.8X and 1.6X, on average, as a result of eliminating the burdensome sorting phase of GraFBoost when running PageRank and BFS algorithms, respectively. Moreover, our proposed approach reaps higher benefits when the graph

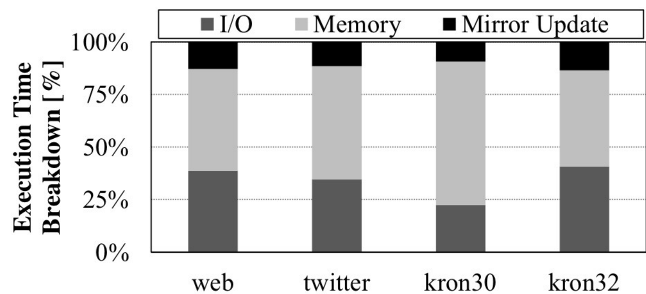


Figure 9: Execution time breakdown for PageRank.

size is larger (`web` and `kron32`). As shown in Figure 7, our optimizations can get very close to GraFHard performance in some cases (e.g., for PageRank on `web`), without incurring any of its hardware and implementation costs. In fact, our mechanism can also use the hardware accelerators to off-load some computation to provide even higher benefits, which we leave it to future work.

In Figure 9 we present the breakdown of execution time for PageRank algorithm. This figure reveals the contribution of each part to the total execution time, including SSD accesses (**I/O**), processing the in-memory graph data (**Memory**), and the time spent on updating mirrors (**Mirror Update**). The **I/O** part does not include SSD accesses for updating mirrors (this part is calculated in Mirror Update). As shown in this figure, the extra work that is introduced to the system for updating mirrors, is less than 15% across the evaluated graphs (even less than 10% in some cases such as `kron30`). This figure also demonstrates that, despite common wisdom, I/O is not the main contributor to the total execution time in graph processing. In some cases, memory accesses delays the processing time more than I/O. Incorporating more efficient caching and pre-fetching techniques, can help lower the memory overhead.

6 Conclusion

In this paper, we study the performance and scalability issues of external graph processing, and devise a mechanism to partition graph vertices to alleviate extra computation overhead of state-of-the-art external graph processing. Our optimizations yield significant performance benefits compared to the state-of-the-art, with more than 2X reduction in total execution time.

Acknowledgements

We thank Tim Harris, our shepherd, and the anonymous reviewers for their constructive feedback. This work has been funded in part by NSF grants 1763681, 1714389, 1629915, 1526750, 1439021, 1302557, and a DARPA/SRC JUMP grant.

References

[1] Graph500 benchmarks. https://graph500.org/.

[2] Intel nvme ssd. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/dc-p4511-series.html.

[3] Micron nvme ssd. https://www.micron.com/products/solid-state-storage/bus-interfaces/nvme-ssds#/.

[4] Samsung enterprise nvme ssd. http://www.samsung.com/semiconductor/products/flash-storage/enterprise-ssd/.

[5] Twitter graph dataset. http://law.di.unimi.it/webdata/twitter-2010/.

[6] Web data commons. http://webdatacommons.org/hyperlinkgraph/.

[7] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco. Heterogeneous memory subsystem for natural graph analytics. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 134–145, Sep. 2018.

[8] Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John CS Lui, and Cheng He. Venus: Vertex-centric streamlined graph computation on a single pc. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1131–1142. IEEE, 2015.

[9] Disa Mhembere Da Zheng, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 45–58, 2015.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[12] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 77–85, New York, NY, USA, 2013. ACM.

[13] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. Grafboost: Using accelerated flash storage for external graph analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 411–424. IEEE, 2018.

[14] David G. Kirkpatrick and Pavol Hell. On the completeness of a generalized matching problem. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 240–245, New York, NY, USA, 1978. ACM.

[15] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[16] Hang Liu and H Howie Huang. Graphene: Fine-grained io management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300. USENIX Association.

[17] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, pages 340–349, Arlington, Virginia, United States, 2010. AUAI Press.

[18] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 527–543, New York, NY, USA, 2017. ACM.

[19] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[20] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*,

SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[21] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[22] Keval Vora, Guoqing (Harry) Xu, and Rajiv Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *USENIX Annual Technical Conference*, pages 507–522, 2016.

[23] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[24] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 544–557. IEEE, 2018.

[25] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.

[26] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Grid-graph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annual Technical Conference*, pages 375–386, 2015.

Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques

Tianli Zhou and Chao Tian
Texas A&M University

Abstract

Various techniques have been proposed in the literature to improve erasure code computation efficiency, including optimizing bitmatrix design, optimizing computation schedule, common XOR operation reduction, caching management techniques, and vectorization techniques. These techniques were largely proposed individually previously, and in this work, we seek to use them jointly. In order to accomplish this task, these techniques need to be thoroughly evaluated individually, and their relation better understood. Building on extensive test results, we develop methods to systematically optimize the computation chain together with the underlying bitmatrix. This led to a simple design approach of optimizing the bitmatrix by minimizing a weighted cost function, and also a straightforward erasure coding procedure: use the given bitmatrix to produce the computation schedule, which utilizes both the XOR reduction and caching management techniques, and apply XOR level vectorization. This procedure can provide a better performance than most existing techniques, and even compete against well-known codes such as EVENODD, RDP, and STAR codes. Moreover, the result suggests that vectorizing the XOR operation is a better choice than directly vectorizing finite field operations, not only because of the better encoding throughput, but also its minimal migration efforts onto newer CPUs.

1 Introduction

A leading technique to achieve strong fault-tolerance in data storage systems is to utilize erasure codes. Erasure codes have been widely used in various data storage systems, ranging from disk array systems [5], peer-to-peer storage systems [22], to distributed storage systems [9, 11], and cloud storage systems [4]. The root of erasure codes can be traced back to the well-known Reed-Solomon codes [20], or more generally, maximum distance separable codes [10]. Roughly speaking, such erasure codes allow a fixed number of component failures in the overall system, and it has the lowest storage overhead (i.e., redundancy) among all strategies that can tolerate the same number of failures. One example is Quantcast File System (QFS) [13], which is an implementation of the data storage backend for the MapReduce framework; it can save 50% of storage space over the original HDFS which uses 3-replication, while maintaining the same failure-tolerance capability.

It has long been recognized that encoding data into its erasure-coded form will incur a much heavier computation load than simple data replication [21], thus more time-consuming. In order to complete the coding computation more efficiently, various techniques have been proposed in the literature to either directly reduce this computation load [2, 5–8, 18], or to accelerate the computation by better utilizing the resources in modern CPUs [12, 16].

Erasure codes rely on finite field operations, and in computer systems, the fields are usually chosen to be $GF(2^w)$, that is, an extension field of the binary field. Using the fact that such finite field operations can be effectively performed using binary XOR between the underlying binary vectors and matrices [3], Plank et al. [18] proposed efficient methods to encode using the "bitmatrix" representation. Several techniques were introduced in the same work to reduce the number of the XOR operations in the computation, and the overall encoding procedure can be viewed as a sequence of such XOR operations, i.e., organized in a computation schedule. Huang et al. [7] (see also the Liberation codes [14] where a similar idea was mentioned) made the observation that some chains of XORs to compute different parity bits may have common parts, and thus by computing the common parts first, the overall computation can be reduced. A matching strategy was proposed to identify such common parts, which leads to more efficient computation schedules. Further heuristic methods to reduce the number of XOR's along these lines were investigated by Plank et al. [17], and lower bounds on the total number of XOR's have also been found [15].

Though with the same goal of reducing the computation load in mind, the coding theory community addresses the is-

sue from another perspective, where specific code constructions have been proposed. Several notable examples of such codes can be found in [2, 5, 6, 8]. These codes usually allow only two or three parities, instead of the flexible choices seen in generic erasure codes.

In contrast to the approaches discussed above where the computation load can be fundamentally reduced, a different approach to improve the encoding speed is to better utilize the existing computation resources in modern computers, i.e., hardware acceleration. Particularly, since modern CPUs are typically built with the capability of "single-instruction-multiple-data" (SIMD), sometimes referred to as vectorization, it was proposed that instead of using the bitmatrix implementation, erasure coding can be efficiently performed by vectorizing finite field operations directly [16]. It was shown that this approach can provide significant improvements over the approach based on the afore-mentioned bitmatrix representation without vectorization. Also related to this approach of optimizing resource utilization, Luo et al. [12] noted that the order of operations in the computation schedule of the bitmatrix-based approach can affect the performance, due to CPU cache miss penalty, and thus steps can be taken to optimize the cache access efficiency.

Although these existing works have improved the coding efficiency of erasure codes to more acceptable levels, the sheer amount of data in modern data storage systems implies that even a small improvement of the coding efficiency may provide significant cost saving and be an important performance differentiator. Particularly, virtualization has been widely adopted for cloud computing, and erasure coding on such cloud platform will be more resource-constrained than on the native platform, thus reducing the computation load is very meaningful. Against this general backdrop, in this work we seek to answer the following questions:

1. Which methods are the most effective, i.e., can provide the most significant improvement? Particularly, how to make a fair comparison of the two distinct approaches of optimizing bitmatrix schedules and vectorization?

2. Can and should these techniques be utilized together, in order to maximize the encoding throughput?

3. If these techniques can be utilized together, which component should be optimized and how to optimize them?

In the process of answering these question, we discovered a particularly effective approach to accelerate erasure encoding: selecting bitmatrices optimized for the weight sum of the number of XOR and copying operations, taking into consideration of the reduction from the common XOR chains, then using XOR-level vectorization for hardware acceleration. The resulting encoding process we propose can provide significant improved encoding throughput compared to [12, 16, 18], ranging from 20% to 500% for various parameters. Moreover, in most cases, the proposed approach can compete with the well-known EVENODD code [2], RAID-6 code [14], RDP code [6], STAR code [8], and triple-parity Reed-Solomon code in Quantcast-QFS [13], which were specifically designed for fast encoding and only for restricted parameters.

Our result also suggests that instead of vectorizing the finite field operation directly, we should vectorize the XOR operations based on the bitmatrix representation. In addition to the throughput advantage, this approach in fact has an important practical advantage: vectorizing general finite field operation involves software implementation of a larger set of relevant operations using the CPU-specific instructions (for different finite field sizes and different finite field operations), while vectorizing XOR operations essentially involves only a single such instruction. As newer versions of CPUs and instruction sets are introduced, the proposed approach only requires minimal migration effort, since most of the bitmatrix implementation is hardware agnostic.

2 Background and Review

2.1 Erasure Codes and Reed-Solomon Codes

Erasure codes are usually specified by two parameters: the number of data symbols k to be encoded, and the number of coded symbols n to be produced. The data symbols and the coded symbols are usually assumed to be in finite field $GF(2^w)$ in computer systems. Such erasure codes are usually referred to as the (n,k) erasure codes.

To be more precise, let k linearly independent vectors $g_0, g_1, \ldots, g_{k-1}$ (of length n each) be given, whose components are in the finite field $GF(2^w)$. Denote the data (sometimes referred to as the message) as $u = (u_0, u_1, \ldots, u_{k-1})$, whose components are also represented as finite field elements in $GF(2^w)$. The codeword for the message u is then

$$v = u_0 g_0 + u_1 g_1 + \cdots + u_{k-1} g_{k-1}.$$

This encoding process can alternatively be represented using the *generator matrix* G of dimension $k \times n$ as

$$v = u \cdot G, \tag{1}$$

where

$$G = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix}.$$

In most data storage applications, the erasure codes have the maxmium distance separable (MDS) property, meaning that the data can be recovered from any k coded symbols in the vector v. In other words, it can tolerate loss of any $m = n - k$ symbols. This property can be guaranteed, as long as any k-by-k submatrix of G, which is created by deleting any m columns from G, is invertible.

2.1.1 Reed-Solomon Code

The original Reed-Solomon code relies on a Vandermonde matrix to guarantee that this invertibility condition is satisfied, i.e.,

$$G = \begin{bmatrix} 1 & \cdots & 1 & \cdots & 1 \\ a_0 & \cdots & a_i & \cdots & a_{n-1} \\ a_0^2 & \cdots & a_i^2 & \cdots & a_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_0^{k-1} & \cdots & a_i^{k-1} & \cdots & a_{n-1}^{k-1} \end{bmatrix} \quad (2)$$

where a_i's are distinct symbols in $GF(2^w)$.

Using a generator matrix of the Vandermonde form will produce a non-systematic form of the message, i.e., the message u is not an explicit part of the codeword v. We can convert G through elementary row operations (see e.g., [10]) to obtain an equivalent generator matrix G'

$$G' = [I,P] = \begin{bmatrix} 1 & 0 & \cdots & 0 & p_{0,0} & \cdots & p_{0,m-1} \\ 0 & 1 & \cdots & 0 & p_{1,0} & \cdots & p_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & p_{k-1,0} & \cdots & p_{k-1,m-1} \end{bmatrix}$$

where the left portion is the identity matrix I_k of dimension k-by-k, and the right portion is the "parity coding matrix" P. As a consequence, we have

$$v = u \cdot G' = (u_0, u_1, \cdots, u_{k-1}, p_0, p_1, \cdots, p_{m-1}), \quad (3)$$

where

$$(p_0, p_1, \cdots, p_{m-1}) = (u_0, u_1, \cdots, u_{k-1}) \cdot P. \quad (4)$$

The matrix P is sometimes also referred to as the coding distribution matrix [19].

2.1.2 Cauchy Reed-Solomon Codes

Instead of reducing from a Vandermonde generator matrix, we can also directly assign the matrix P such that the invertible condition can be satisfied. One well-known choice is to let P be a Cauchy matrix, and the corresponding erasure code is often referred to as Cauchy Reed-Solomon (GRS) codes [3].

More precisely, denote $X = (x_1, \ldots, x_k)$ and $Y = (y_1, \ldots, y_m)$, where x_i's and y_i's are distinct elements of $GF(2^w)$. Then the element in row-i column-j in the Cauchy matrix is $1/(x_i + y_j)$. It is clear that any submatrix of a Cauchy matrix is still a Cauchy matrix. Particularly, let C_ℓ be an order-ℓ square submatrix of a Cauchy matrix:

$$C_n = \begin{bmatrix} \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \cdots & \frac{1}{x_1+y_\ell} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \cdots & \frac{1}{x_2+y_\ell} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_\ell+y_1} & \frac{1}{x_\ell+y_2} & \cdots & \frac{1}{x_n+y_\ell} \end{bmatrix},$$

then C_ℓ is invertible, and the elements of the inverse of the Cauchy matrix C_ℓ^{-1} have an explicit analytical form [3].

One advantage of using Cauchy Reed-Solomon code instead of the classical Reed-Solomon code based on Vandermonde matrix is that inverting an order-n Vandermonde-based matrix is of time complexity $O(n^3)$, while inverting a Cauchy matrix has a time complexity $O(n^2)$. Following [18], we adopt Cauchy Reed-Solomon codes in this work, instead of the Vandermonde matrix based approach.

2.2 Encoding by Bitmatrix Presentation

Finite field operations in $GF(2^w)$ can be implemented using the underlying bit vectors and matrices [3], and thus all the computations can be conducted using direct copy or binary XOR. Based on this representation, reducing erasure code computation is equivalent to reducing the number of XOR and copying in the computation schedule. Various techniques to optimize this metric have been proposed in the literature, which we also briefly review in this subsection.

2.2.1 Convert Parity Matrix to Bitmatrix

Each element e in $GF(2^w)$ can be represented as a row vector $V(e)$ of $1 \times w$ or a matrix $M(e)$ of $w \times w$, where each element in the new representation are in $GF(2)$. $V(e)$ will be identical to the binary representation of e, and the ith row in $M(e)$ is $V(e^{2^{i-1}})$. If we apply this representation, the parity coding matrix of size $k \times m$ will be converted to a new parity coding matrix of size $wk \times wm$ in $GF(2)$, i.e., a binary matrix. Using the bitmatrix representation, erasure coding can be accomplished by XOR operations, together with an initial copying operation. A simple example of bitmatrix encoding is shown in Figure 1, where the matrix multiplications are now converted to XORs of data bits corresponding to the ones in the binary parity coding matrix, together with some copying operations.

The number of 1's in the bitmatrix is the number of XOR operations in encoding. Choosing different $X = (x_1, \ldots, x_k)$ and $Y = (y_1, \ldots, y_m)$ vectors will produce different encoding bitmarices, which have different numbers of 1's and thus different numbers of XOR operations in the computation schedule. In [19], exhaustive search and several other heuristics were used to find better assignments of the (X,Y) vector such that the number of 1's in the bitmatrix can be reduced. It was shown that these techniques can lead to encoding throughput improvement ranging from 10%-17% for different (n,k,w) parameters.

2.2.2 Normalization of the Parity Coding Matrix

A simple procedure to reduce the encoding computations is to multiply each row and each column of $G = [I,P]$ by certain non-zero values, such that some of the coefficients are more

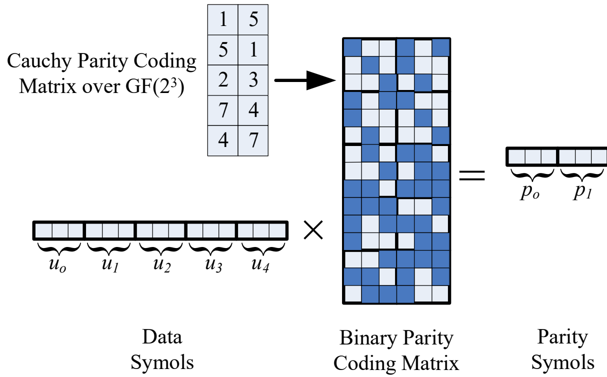


Figure 1: Encoding in bitmatrix representation for $k=5, m=2, w=3$: the parity coding matrix is first converted to its bitmatrix form (blue as bit 1, and gray as 0), and the encoding is done as a multiplication of the length-15 binary vector and the 15×6 binary matrix.

suitable for computation (e.g., to make some elements be the multiplicative unit 1), which is referred to as bitmatrix normalization [18]. Clearly this does not change the invertibility property of the original generator matrix. More precisely, for any parity coding matrix P, we can use the following procedure:

1. For each row-i in P, divide each element by $p_{i,0}$, after which all elements in column-0 will be the multiplicative unit in the finite field.

2. For each column-j except the first:

   (a) Count the number of ones in the column (in the bitmatrix representation of this column).

   (b) Divide column-j by $p_{i,j}$ for each i, and count the number of ones in this column (in the bitmatrix representation).

   (c) Using the $p_{i,j}$ which yields the minimum number of ones from the previous two steps, let the new column-j be the values after the division of $p_{i,j}$. In other words, we normalize column-j with the element in the column which induces the minimum number of ones in the bitmatrix.

An example given by Plank et al. [18] shows that for $m=3$, this procedure can reduce the number of ones in the bitmatrix to 34 ones from the original 46. This method is rather straightforward to implement and does not require any additional optimization.

2.2.3 Smart Scheduling

The idea of reusing some parity computation to reduce overall computation can be found in the code construction proposed by Plank [14], and this idea materialized as the smart scheduling component in the software package [18]. The underlying idea is as follows: if a parity bit can be written as the XOR of another parity bit and a small number of data bits, then it can be computed more efficiently. The following example should make this idea clear. Suppose the two parities are given as

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4, \quad p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5. \qquad (5)$$

A direct implementation to generate p_1 will use 3 XOR operations, but by utilizing p_0, p_1 can be computed as

$$p_1 = p_0 \oplus u_4 \oplus u_5,$$

which requires only two XORs. This technique requires slightly more effort to implement and optimize than the previous technique, however the computation schedule can essentially be generated offline and thus it does not contribute significantly to the encoding computation.

2.2.4 Matching

The idea of smart scheduling in fact has a related form. Huang et al. [7] recognized that instead of restricting to reusing computed parity bits, any common parts of the XOR chains in computing the parity bits can be reused, which reduces the total number of XOR computations. A grouping strategy was consequently proposed to optimize the number of necessary XORs. The proposed method focuses only on common XOR operations involving a pair of data bits, but not common operations involving three or more data bits. This is because common operations of three or more data bits are scarce in practical codes, and at the same time identifying them can be time consuming.

The core idea of the proposed approach by Huang et al. (common operation first) is to represent the demand data pair of parities in a graph, each vertex of which corresponds to an input data bit, and the weight of edge between vertex i and j represents the number of parities demands $u_i \oplus u_j$. A greedy procedure is used to extract a sub-graph with the edges of the largest weights, then the *maximum cardinality matching* algorithm can be used on the sub-graph to find a set of edges, where none of them have shared vertices. Each edge such found indicates a pair of input data bits whose XOR is common in some XOR chains. The algorithm then removes these edges and vertices from the original graph, and repeat this subgraph extraction and matching procedure on the remaining graph. This technique requires further effort to implement and optimize than smart scheduling, but the computation schedule can also be generated offline.

In the matching phase on the sub-graphs, two different strategies were introduced:

1. Unweighted matching. This method views all edges in the graph as having the same weight.

2. Weighted matching. This method uses the heuristic of making the matching covers as few dense nodes in the sub-graph (defined as degrees of nodes) as possible, by adjusting the assignments of the weights on the edges according to the sum of degrees of both ends in the original graph.

In our work, an implementation of Edmond's blossom shrinking algorithm in the LEMON graph library [1] is utilized to implement these matching algorithms.

Generalizing the matching technique, a few more heuristic methods to reduce the number of XOR's were investigated by Plank et al. [17]. However, these methods themselves can be extremely time-consuming, and the observed improvements in encoding throughput appear marginal. Therefore, we do not pursue these heuristic methods in this work.

2.3 Optimizing Utilization of CPU Resources

Speeding-up erasure coding computation can also be obtained through more efficient utilization of CPU resources, such as vectorization and avoiding cache misses.

2.3.1 Vectorization for Hardware Acceleration

Modern CPUs typically have SIMD capability, which can dramatically improve the computation speed. Most of the computation in erasure coding can be done in parallel, including XOR operations and more general finite field operations, since the same operations need to be applied on an array of data.

Directly vectorizing finite field operations has been previously investigated and implemented for finite fields $GF(2^8)$, $GF(2^{16})$, and $GF(2^{32})$ [16]. This was accomplished through invoking the 128-bit SSE vectorization instruction set for INTEL and AMD CPUs. More recently, 256-bit AVX2 vectorization instructions and 512-bit AVX-512 vectorization instructions are becoming more common in newer generations of CPUs. In this work, we only report results based on the 128-bit SSE instruction set in order to make the comparison fair, however, our implementation can indeed utilize 256-bit AVX2 vectorization instructions and 512-bit AVX-512 vectorization instructions without any essential change to the software program, exactly because of the reason mentioned at the end of Section 1.

2.3.2 Reducing Cache Misses

The sequence of operations can affect the coding performance due to cache misses, and more efficient cache-in and cache-out can be accomplished by choosing a streamlined computation order. A detailed analysis of the CPU-cache handling and the effects of different operation orders was given by Luo et al. [12]. The conclusion is that increased spatial data locality can help to reduce cache miss penalty. Consequently, a computation schedule was proposed where a data chunk is accessed only once sequentially, each of which is then used to update all related parities. More precisely, this strategy will read the data symbol u_0 first, update all parities which involves u_0, then $u_1, u_2, ..., u_{k-1}$. In contrast, the naive strategy of computing the parity symbols $p_0, p_1, ..., p_{m-1}$ sequentially suffers a performance loss, which was reported to be roughly 23%$\sim$36%.

3 Effects of Individual Techniques and Possible Combinations

As mentioned earlier, the first step of our work is to better understand the effects of the existing techniques in speeding up the erasure coding computation. For this purpose, we first conduct tests on encoding procedures with each individual component enabled. The relation of different techniques will be discussed later, which allows us to utilize them together in the proposed design procedure.

3.1 Analyzing Individual Techniques

The existing techniques we consider individually are: XOR bitmatrix normalization (BN), XOR operation smart scheduling (SS), common XOR reduction using unweighted matching (UM), common XOR reduction using weighted matching (WM), scheduling for caching optimization (S-CO), and direct vectorization of XOR operation (V-XOR). The first four techniques can be viewed as optimization on the bitmatrix such that the total number of XOR (and copy) operations is reduced, as discussed in Section 2.2. The last technique, though has not been systematically investigated in the literature, is in fact a rather natural choice and is thus included in our test. The latter two methods aim to better utilize the CPU resources such that the computation can be done more efficiently without reducing fundamentally the computation load, and they are more hardware platform dependent.

We conducted encoding throughput tests for a range of (n,k,w) parameters most relevant for data storage applications, the results of which are reported in Table 1, in terms of the improvement over the baseline approach of taking a simple Cauchy Reed-Solomon code without any additional optimization. For the first four techniques, the improvement is measured in terms of the reduction of the number of 1's in the bitmatrix, while for the latter two, the improvement is measured in terms of the encoding throughput increase. Multiple tests are performed for each parameter, and we report the average over them. In the last row of Table 1, the average encoding throughput over all the tested parameters is included. All tests in this work are conducted on a workstation with an AMD Ryzen 1700X CPU (8 cores) running at 3.4GHz, 16GB DDR4 memory, which runs the Ubuntu 18.04 64-bit operating system and the compiler is GCC 7.3.0

Table 1: Performance improvements by individual techniques

(n,k,w)	The number of XOR's in the baseline code	Reduction in the number of XOR's				Throughput increase	
		BN	SS	UM	WM	S-CO	V-XOR
(8,6,4)	104	42.31%	17.31%	31.73%	31.73%	2.14%	98.44%
(8,6,8)	362	53.31%	33.70%	34.53%	34.81%	0.52%	126.36%
(9,6,4)	152	32.89%	19.74%	32.89%	32.89%	3.54%	90.48%
(9,6,8)	549	44.63%	29.14%	38.25%	38.25%	4.51%	152.87%
(10,6,4)	200	27.50%	22.00%	36.00%	36.00%	0.60%	91.30%
(10,6,8)	736	40.90%	28.80%	38.59%	38.59%	1.23%	130.71%
(12,8,4)	256	23.44%	23.44%	35.94%	35.94%	9.21%	118.26%
(12,8,8)	1028	36.38%	24.81%	39.79%	39.79%	0.10%	156.88%
(16,10,4)	496	18.95%	21.77%	37.70%	37.70%	8.28%	138.93%
(16,10,8)	1920	30.16%	21.98%	40.89%	40.89%	5.00%	179.00%
Average over all tested cases		35.05%	24.27%	36.63%	36.66%	4.81%	130.4%

which is the default compiler of the OS. During compilation, O3 optimization, SSE4 and AVX2 instruction sets are enabled. Using different compilers and different compiler options may yield slightly different coding throughputs, but will not change the relative relationship among different coding methods, when the same compiler and compiler options are used across them.

It can be seen that among the first four techniques, BN usually provides roughly 35% improvement over the baseline on average. The variation among different (n,k,w) parameter settings is not negligible, which is likely caused by the specific field chosen and the number of possible choices of (X,Y) coefficients in the Cauchy Reed-Solomon codes. In comparison, smart scheduling can provide a more modest $\sim 24\%$ improvement. Both versions of matching algorithms can also provide significant improvements over the baseline, however, there is not a clear winner between the two versions of the matching algorithms.

Between the latter two techniques, S-CO can provide a gain of $\sim$5%, while V-XOR is able to improve the coding speed by roughly $100\% - 200\%$. The improvement observed in our work by S-CO is considerably less significant compared to the 23-36% improvement reported by Luo et al. [12], which we suspect is due to the improvement in cache size and cache prediction algorithm in modern CPUs. Indeed, when we test the same approach on different operating systems and CPUs, different (sometimes significantly different) amounts of improvement can be observed. Among all the techniques, V-XOR appears to be able to provide the most significant performance improvement.

The performance of directly vectorizing finite field operations [16] is not included in the set of tests above, because it belongs to a completely different computation chain. It does not utilize the bitmatrix representation at all, and thus completely bypasses all other techniques. This observation in fact raises the following question: can vectorizing XORs

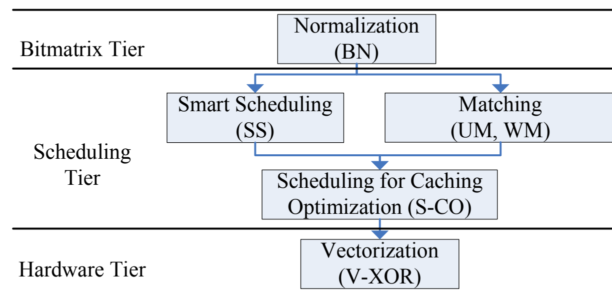


Figure 2: Operation tiers of the individual techniques

within the bitmatrix framework be a better choice than vectoring finite field operations directly? Surprisingly, the question has not been answered in the existing literature. As we shall discuss in the next section, our result suggests that the answer to this question indeed appears to be positive.

3.2 Combining the Individual Techniques

Equipped with individual improvements reported above, we are interested in whether and how these techniques can be combined to achieve more efficient erasure encoding. The techniques we test can be categorized into three tiers: the bitmatrix tier, the scheduling tier, and the hardware-related tier, as shown in Figure 2.

Since these techniques mostly reside in different tiers, they can be applied in tandem. The only exception is among the SS, UM, and WM techniques, since they are essentially optimizing the same component in the computation chain. As such, we need choose the technique or techniques to adopt. Additionally, although BN is able to provide improvement and can be applied together with other techniques, it is essentially also a procedure to optimize the bitmatrix, and the set of tests does not indicate whether it is still going to be effective when combined with SS, UM, or WM. The S-CO

Table 2: Combination of individual strategies

i	j
BN disabled (0)	no XOR reuse (0)
	SS (1)
BN enabled (1)	UM (2)
	WM (3)

technique is basically independent of the other techniques, and thus we can always invoke it for any combined strategies. Similarly, V-XOR can be applied directly together with all the other techniques. In fact, because of the significance of the improvement offered by V-XOR, including it should be a priority when using the techniques jointly.

We use a pair of indices (i, j) to enumerate the eight possible combinations of bitmatrix-based techniques, where $i \in \{0,1\}$ and $j \in \{0,1,2,3\}$, as shown in Table 3.2. For example, strategy-$(1,3)$ means both BN and WM are used. These combinations are the candidate strategies, within the bitmatrix framework, that we need to select from.

4 Selecting Coding Strategy under Optimized Bitmatrices

Our next task is to select one of eight strategies which can offer the best performance. The complication here is that since different choices of the (X,Y) vector in Cauchy Reed-Solomon code can lead to different computation load during erasure coding computation (see [19]), the (X,Y) coefficients need to be optimized. In other words, the strategies should be evaluated with such optimized bitmatrices. For this purpose, we first conduct heuristic optimizations to minimize the cost function of the total number of XOR and copy operations, under these eight strategies, the result of which is used to determine the best strategy. At the end of the section, we discuss possible improvement to the cost function.

4.1 Bitmatrix Optimization Algorithms

For a fixed choice of (X,Y) which determines the Cauchy parity coding matrix P, a given (i, j)-strategy will induce a given number of total computation operations, including XORs and copyings. Let us denote the function mapping from (X,Y) to this cost function as $c_{i,j}(X,Y)$. To compute, for example, $c_{1,2}(X,Y)$, we first conduct bitmatrix normalziation on the Cauchy matrix induced by (X,Y), then apply the unweighted matching algorithm to obtain the number of XOR operations and the number of copying operations in the encoding computation, and finally compute the total number of the operations. Our goal here is thus to find the choice of (X,Y) vector that minimizes this cost function. Due to the complex relation between the choice of (X,Y) vector and the cost function value, it is not possible to find the optimal solution using standard optimization techniques. Instead, we adopt two heuristic optimization procedures: simulated annealing (SA) and genetic algorithm (GA).

In the simulated annealing, there are several parameters that need to be set, however, we found that the results in this application is not sensitive to them. The only parameter of material importance is the annealing factor Δ, which control the rate of cooling.

For the genetic algorithm, we defined the population as a set of (X,Y) vectors. There is also a set of standard parameters in genetic algorithm (such as the crossover rate and the mutation rate), but the most important factor appears to be the crossover procedure in this setting. We considered two crossover methods to generate a child Cauchy matrix.

1. Random crossover: From the set of finite field elements which appear in the parent vectors (X,Y), choose $k+m$ elements at random and produce a random assignment of the new (X,Y) as the child.

2. Common elements first crossover: The finite field elements which appear in both parents are selected first as the element of child, and the others are chosen at random from the other elements which appear only in one of the parents.

The second approach tends to provide better new bitmatrices, which appears to match our intuition that some assignments are better than others, and keeping the good trait in the children may produce even better assignments.

In Table 3, we include a subset of (n,k,w) parameter choices we have attempted using the two optimization approaches together with the case when (X,Y) vector is assigned according to the sequential order in the finite field; the other test results are omitted due to space constraint. It can be seen that the genetic algorithm provides better solutions than simulated annealing in most cases, and for this reason we shall adopt GA in the subsequent discussion.

Due to the heuristic nature of the two algorithms, the readers may question whether the optimized (X,Y) choice can indeed provide any performance gain. It can be seen in Table 3, that both SA and GA can provide significant improvement on the total number of operations by finding good bitmatrices. In Figure 3, we further plot the amounts of cost reduction for different (n,k,w) parameters, from the baseline approach of without any optimization. It can be seen for most (n,k,w) parameters, meaningful (sometime significant) gains of $\sim 5\%$ to $\sim 25\%$ can be obtained. Thus, although the two heuristic optimization methods cannot guarantee finding the optimal solutions, they do lead to considerably improved bitmatrix choices.

4.2 Choosing the Best (i, j)-Strategy

We can now select the best (i, j)-strategy, using the optimized bitmatrices obtained by the genetic algorithm. In Figure 4, the cost function values of different (i, j)-strategies

Table 3: Comparison of the total number of XOR and copy operations for all (i,j)-strategies, when the bitmatrices are obtained without optimization, by simulated annealing, and by the genetic algorithm, respectively, as the three columns in each box.

(i,j)	$(n,k,w)=(8,6,4)$			$(n,k,w)=(9,6,4)$			$(n,k,w)=(10,6,4)$			$(n,k,w)=(12,8,4)$			$(n,k,w)=(16,10,4)$		
(0,0)	112	**77**	**77**	164	122	**117**	216	173	**162**	272	**232**	**232**	520	**462**	464
(0,1)	94	70	**68**	134	102	**100**	172	137	**134**	212	190	**188**	412	**368**	**368**
(0,2)	90	72	**68**	127	103	**101**	164	134	**132**	204	186	**180**	376	**344**	345
(0,3)	90	72	**68**	127	102	**101**	164	**132**	**132**	204	184	**182**	376	**343**	345
(1,0)	68	**58**	**58**	114	99	**98**	161	142	**141**	212	**198**	**198**	426	**419**	**419**
(1,1)	64	**58**	**58**	99	90	**89**	138	**120**	**120**	189	174	**171**	365	**352**	**352**
(1,2)	64	58	**57**	98	**87**	**87**	133	**118**	**118**	176	165	**164**	326	317	**316**
(1,3)	64	58	**57**	98	90	**87**	132	**118**	**118**	175	**164**	**164**	326	**316**	**316**

(i,j)	$(n,k,w)=(8,6,8)$			$(n,k,w)=(9,6,8)$			$(n,k,w)=(10,6,8)$			$(n,k,w)=(12,8,8)$			$(n,k,w)=(16,10,8)$		
(0,0)	378	291	**247**	573	467	**425**	768	611	**582**	1060	880	**841**	1968	1685	**1681**
(0,1)	256	234	**225**	413	393	**349**	556	518	**479**	805	740	**684**	1546	1432	**1374**
(0,2)	286	227	**217**	408	346	**321**	532	474	**450**	726	636	**591**	1304	1180	**1170**
(0,3)	286	216	**197**	408	357	**329**	532	460	**450**	726	636	**627**	1304	1180	**1170**
(1,0)	185	151	**142**	328	286	**262**	467	434	**419**	686	613	**563**	1389	1293	**1246**
(1,1)	164	155	**133**	285	270	**230**	411	383	**371**	593	557	**544**	1264	1184	**1133**
(1,2)	167	143	**134**	272	244	**225**	377	343	**335**	520	487	**462**	998	951	**922**
(1,3)	167	144	**130**	273	249	**233**	377	**337**	**337**	520	473	**467**	995	941	**932**

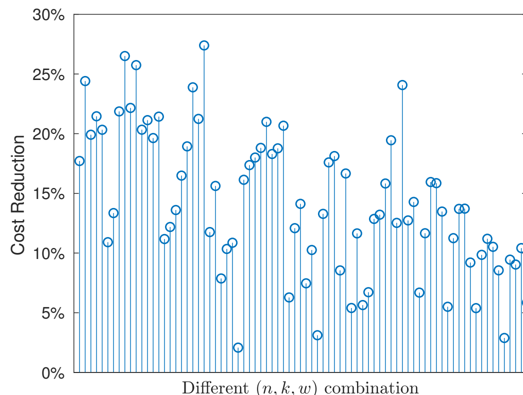


Figure 3: Cost reductions obtained by the genetic algorithm for different (n,k,w) parameters (sorted by n).

are shown, under various (n,k,w) parameters. Here again we have chosen a subset of representative test results, and omit others due to space constraint. It can be seen that the strategies $(1,2)$ and $(1,3)$ are the best among all the possibilities, and they do not show any significant difference between themselves.

4.3 Refining the Cost Function

We have so far used the total number of XORs and copying operations as the cost function in the optimization of bitmatrices. However, this choice may not accurately capture all the computation operations, and as such, we next consider three possible cost functions: 1) the total number of XORs, 2) the total number of operations, including XORs and copyings, and 3) a weighted combination of the number of XORs and that of copying operations. In the last option, we set the weight according to the empirical testing result of these operation on the target workstation: the time taken for copying (memcpy) and that for XORing the same amount of data are measured. On our platform, the weight given to XOR is roughly 1.5 the weight given to memory copying. To distinguish from the cost function $c_{i,j}(X,Y)$, we write this last cost function as $c'_{i,j}(X,Y)$.

The effectiveness of these three cost functions is evaluated and shown in Table 4 by using the genetic algorithm to find the optimized bitmatrices. The resulting bitmatrices obtained under the three cost functions are used to encode the data with the $(1,3)$-strategy, and we compare the encoding throughput values. It can be seen that the third cost function is able to most accurately capture the encoding computation cost in practice. The improvements obtained by the refined cost function $c'_{i,j}(X,Y)$, in most cases, are not extremely large, ranging from $0\% - 10\%$, and occasionally it does cause a minor performance degradation than the cost function $c_{i,j}(X,Y)$.

5 The Proposed Design and Coding Procedure, and Performance Evaluation

From the previous discussion, the proposed bitmatrix design procedure is quite clear: perform a suitable optimization algorithm (the genetic algorithm is used in this work) with the weighted cost function $c'_{1,2}(X,Y)$ or $c'_{1,3}(X,Y)$. The proposed erasure coding procedure then naturally involves the

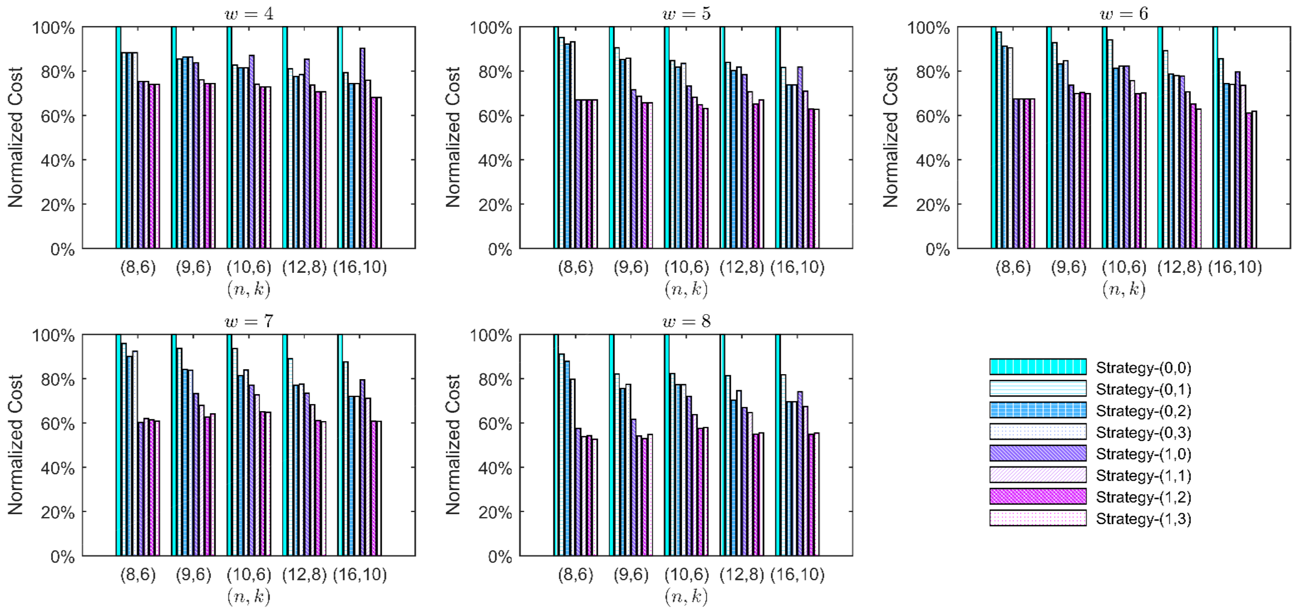


Figure 4: Total number of XOR and copying operations for all (i, j) strategies with optimized bitmatrices

Table 4: Encoding throughput (GB/s) using bitmatrices obtained by the genetic algorithm under different cost functions

(n,k,w)	Cost Function		
	# of XOR	# of XOR and copying	Weighted
(8,6,4)	4.64	4.66	**4.68**
(8,6,8)	4.30	**4.35**	4.32
(9,6,4)	3.72	3.73	**3.80**
(9,6,8)	3.28	3.28	**3.44**
(10,6,4)	2.33	2.51	**2.52**
(10,6,8)	1.99	1.97	**2.11**
(12,8,4)	2.96	3.11	**3.16**
(12,8,8)	2.54	2.56	**2.58**
(16,10,4)	2.29	2.29	**2.32**
(16,10,8)	1.71	1.72	**1.74**

corresponding components: from the selected (X,Y), produce the corresponding bitmatrix by bitmatrix normalization, then generate the computation schedule from the produced bitmatrix using the selected matching algorithm and following the cache-friendly order, and perform the vectorized XOR operations using the necessary CPU instructions.

In the sequel, we discuss a few details in integrating these techniques, and then provide performance evaluation in comparison to the existing approaches.

5.1 Integrating XOR Matching and S-CO

As described in 2.3.2, ordering the encoding operation sequence according to the data order can increase spatial data locality, which helps reduce cache miss penalty. The schedules in [12] contain only data to parity operations

$$u_i \to p_j,$$

where the arrow indicates the direction of data flow for either a memory copy or an XOR operation. Because of the UM procedure or the WM procedure in the computation chain, the common XOR pairs need to be computed and stored as intermediate results, denoted as int_l. As such, in the proposed procedure, the schedule contains three types of operations

$$u_i \to p_j, \quad u_i \to int_l, \quad int_l \to p_j.$$

To reduce cache misses, we need to extend the ordering method to handle these three cases. This can be accomplished by following the sequential order of (XORing and copying) the data bits to the parity bits or intermediate bits first, then the intermediate bits to the parity bits, e.g.,

$$\begin{bmatrix} u_0 \to p_0 \\ u_0 \to int_0 \\ \cdots \\ u_1 \to p_3 \\ \cdots \\ int_0 \to p_4 \\ \cdots \end{bmatrix}.$$

This order ensures that each data bit u_i will be read exactly once, which maintains the spatial data locality.

5.2 Vectorizing XOR Operations

Each step in the computation schedule is either a copying operation, or an XOR operation. In practice, instead of performing them in a bit-wise manner, a set of bits is processed at a time, i.e., using an extended word format. The smallest such extension is a byte (8bits) in computer systems, and similarly a long long word has 64 bits. For CPUs with an SIMD instruction set, the extended word can be 128 bits, 256 bits, or 512 bits. A single instruction thus completes the operation on 8 bits, 64 bits, 128 bits, 256 bits, or 512 bits, respectively. The instructions we used in this work are included in Intel® Intrinsics instruction set, which has also been adopted in AMD CPUs.

The C code to perform the XOR operation is as follows:

```
#include <x86intrin.h>
void fast_xor( char *r1,   /* region 1 */
               char *r2,   /* region 2 */
               char *r3,   /* r3 = r1 ^ r2 */
               int size)   /* bytes of region */
{
    __m128i *b1, *b2, *b3;
    int vec_width = 16,j;
    int loops = size / vec_width;
    for(int j = 0;j<loops;j++)
    {
        b1 = (__m128i *)(r1 + j*vec_width);
        b2 = (__m128i *)(r2 + j*vec_width);
        b3 = (__m128i *)(r3 + j*vec_width);
        *b3 = _mm_xor_si128(*b1, *b2);
    }
}
```

The SSE data type `__m128i` is a vector of 128 bits, which can be easily converted from any common data types such as `char`, `int`, or `long`. The instruction `_mm_xor_si128` computes the bitwise XOR of 128 bits. To utilize the 256 bits AVX2 instructions or the 512 bits AVX-512 instructions, the migration is rather straightforward in the proposed computation procedure as follows:

1. Update the data format:
   AVX2 : `__m128i` → `__m256i`
   AVX-512: `__m128i` → `__m512i`
2. Update the bitwidth parameter:
   AVX2 : `vec_width = 16;` → `vec_width = 32;`
   AVX-512: `vec_width = 16;` → `vec_width = 64;`
3. Update the instruction:
   AVX2 : `_mm_xor_si128` → `_mm256_xor_si256`
   AVX-512: `_mm_xor_si128` → `_mm512_xor_epi32`

In our experience, performing the XOR operation on the same amount of data with 64 bitwidth (long long format) is 30% slower than `_mm_xor_si128`, and it is 50% slower than `_mm256_xor_si256`. Note that our tests are performed on an AMD CPU, however INTEL CPUs may have somewhat different characteristics. The instruction `_mm512_xor_epi32` is expected to be even faster, however we currently do not have such a platform for testing.

Table 5: Encoding throughput (GB/s) for methods that allow general (n,k) parameters and $w = 8$

(n,k)	Proposed	Vectorized XOR-based CRS code	Vectorized GF-based RS code [16]
(7,5)	4.64	**4.73**	2.52
(8,6)	5.21	**5.22**	2.70
(9,7)	5.32	**5.45**	2.74
(10,8)	5.36	**5.59**	2.77
(12,10)	5.72	**5.88**	2.81
(8,5)	**3.19**	2.75	1.76
(9,6)	**3.49**	2.84	1.77
(10,7)	**3.67**	2.79	1.80
(11,8)	**3.72**	2.92	1.82
(13,10)	**3.82**	3.10	1.84
(10,6)	**2.55**	2.15	1.31
(11,7)	**2.75**	2.17	1.32
(12,8)	**2.86**	2.20	1.35
(14,10)	**2.86**	2.19	1.40
(15,10)	**2.30**	1.79	1.11
(16,10)	**1.96**	1.48	0.92

5.3 Encoding Performance Evaluation

Here we provide comprehensive encoding throughput test results between the proposed approach and several well-known efficient erasure coding methods in the literature, as well as the erasure array codes designed for high throughput. The latter class includes EVENODD code [2], RDP code [6], Linux Raid-6 [14], STAR code [8], and Quantcast-QFS [13] code. EVENODD code, RDP code, Raid-6 are specially designed to have two parities, and STAR code and Quantcast-QFS are specially designed to have only three parities; in order to make the comparison fair, we use 128-bit vectorized XOR discussed in Section 5.2 for these codes as well. Since open source implementations for these codes are not available, we have implemented these coding procedures, with and without vectorization, to use in our comparison. The former class of codes includes several efficient Cauchy Reed-Solomon code implementations based on bitmatrices (XOR-based CRS) [12, 16, 18], and finite field vectorized Reed-Solomon code (GF-based RS code) [16]; the source code for them can be found in the Jerasure library 2.0 [16] publicly available online, which is used in our comparison. The implementation in Jerasure library 2.0 is based on vectorizing (through 128-bit instruction) finite field operation in $GF(2^8)$, $GF(2^{16})$, and $GF(2^{32})$. The Cauchy Reed-Solomon code implementation in Jerasure library 1.2 [18] can be adapted to utilize with XOR-level vectorization, however, it would

Table 6: Encoding throughputs (GB/s): Three parities

(n,k,w)	Proposed	Vectorized XOR-based CRS code	STAR code [8]	Quancast QFS [13]
(8,5,4)	**3.59**	3.25	2.97	2.92
(8,5,8)	**3.19**	2.75	2.97	2.92
(9,6,4)	3.52	**3.72**	3.42	3.04
(9,6,8)	**3.49**	2.84	3.42	3.04
(10,7,4)	**4.15**	3.86	3.76	3.25
(10,7,8)	3.67	2.79	**3.76**	3.25
(11,8,4)	**4.36**	4.13	3.94	3.27
(11,8,8)	3.72	2.92	**3.94**	3.27
(13,10,4)	**4.51**	4.08	4.37	3.41
(13,10,8)	3.82	3.10	**4.37**	3.41

Table 7: Encoding throughputs (GB/s): Two parities

(n,k,w)	Proposed	Vectorized XOR-based CRS code	Vectorized Raid-6	EVEN ODD [2]	RDP [6]
(7,5,4)	**5.16**	4.85	n/a	4.28	4.37
(7,5,8)	4.64	**4.73**	2.18	4.28	4.37
(8,6,4)	4.67	**5.22**	n/a	4.83	4.95
(8,6,8)	5.21	**5.22**	2.15	4.83	4.95
(9,7,4)	**5.77**	5.59	n/a	5.20	5.32
(9,7,8)	5.32	**5.45**	2.16	5.20	5.32
(10,8,4)	**5.90**	5.23	n/a	5.50	5.69
(10,8,8)	5.36	5.59	2.17	5.50	**5.69**
(12,10,4)	6.23	6.00	n/a	6.02	**6.24**
(12,10,8)	5.72	5.88	2.17	6.02	**6.24**

not include the UM (or WM) component and the refinement of the cost function discussed in Section 4.3.

In the tests reported below, the parameters k varies from 5 to 10, m from 2 to 6, and w from 4 to 8. Some of these parameters do not apply for some of the reference codes and coding methods, which will be indicated as n/a in the result tables. The comparison is first presented in three groups.

- In Table 5, the proposed approach is compared with vectorized XOR-based Cauchy Reed-Solomon code, and vectorized finite field Reed-Solomon code, when $w = 8$. All three approaches are applicable for general (n,k) coding parameters, however the implementation of vectorized finite field operations in [16] can only use $w = 8, w = 16$ or $w = 32$; in contrast, the other two approaches can use other w values. Here we choose $w = 8$ for a fair comparison. When $m = n-k = 2$, it is seen that vectorized XOR-based Cauchy Reed-Solomon code is slightly faster than the proposed approach, because the SS technique in these cases in fact provides a slighter better scheduling than WM. When m is larger than 2, the proposed procedure can provide a more significant throughput advantage. Vectorizing finite field operations is always the worse choice among the three by a large margin.
- In Table 6, the proposed approach is compared with well-known codes with three parities. It is seen that the proposed approach is able to compete with these coding theory based techniques. It should be noted that STAR code and Quantcast QFS code do not rely on the parameter w, and thus the throughput performances for $w = 4$ and $w = 8$ are the same for each (n,k) parameter.
- In Table 7, the proposed approach is compared with well-known codes with two parities. It is again seen that the proposed approach is able to compete with these established coding techniques. EVENODD code and RDP code do not rely on the parameter w, and thus the throughput performances for $w = 4$ and $w = 8$ are the same.

Table 8: Encoding throughput improvements over references

Reference codes or methods	Improvement by proposed code
General (n,k) Codes	
GF-based RS code w/o vectorization	552.27%
XOR-based CRS code w/o vectorization	53.65%
Vectorized GF-based RS code [16]	99.82%
Vectorized XOR-based CRS code	14.98%
Three Parities Codes	
STAR [8]	5.59%
Quancast-QFS [13]	21.68%
Two Parities Codes	
Raid-6 w/o vectorization	206.88%
Vectorized Raid-6	142.07%
RDP [6]	5.85%
EVENODD [2]	8.79%

In Table 8, we list the amounts of improvements of the proposed approach over other reference approaches or codes, averaged over all tested (n,k,w) parameters. It is seen that the proposed approach can provide improvements over all existing techniques, some by a large margin. The result in this table is included here to provide a summary on the performance by various techniques, however for individual (n,k,w) parameter, the performance may vary as indicated by the previous three tables.

5.4 Decoding Performance Evaluation

In practical systems, data is usually read out directly without using the parity symbols, unless the device storing the data symbols becomes unavailable, i.e., in the situation of degraded read. Therefore, the most time consuming computation in erasure code decoding is in fact invoked much less often, which implies that the decoding performance should be viewed as of secondary importance. However, it is still

Table 9: Decoding throughput (GB/s) for methods that allow general (n,k) parameters and $w=8$

(n,k)	Proposed	Vectorized XOR-based CRS code	Vectorized GF-based RS code [16]
(7,5)	3.87	**4.56**	2.58
(8,6)	**5.45**	4.86	2.67
(9,7)	4.46	**5.06**	2.70
(10,8)	4.89	**5.11**	2.75
(12,10)	4.45	**5.52**	2.79
(8,5)	**3.04**	2.11	1.71
(9,6)	**2.94**	2.20	1.74
(10,7)	**3.28**	2.29	1.76
(11,8)	**3.08**	2.31	1.71
(13,10)	**3.21**	2.37	1.88
(10,6)	**2.38**	1.80	1.31
(11,7)	**2.35**	1.85	1.32
(12,8)	**2.54**	1.87	1.33
(14,10)	**2.47**	1.89	1.38
(15,10)	**2.00**	1.48	1.09
(16,10)	**1.77**	1.30	0.91

Table 10: Decoding throughputs (GB/s): Three parities

(n,k,w)	Proposed	Vectorized XOR-based CRS code	STAR code [8]	Quancast QFS [13]
(8,5,4)	**4.28**	3.08	3.20	1.77
(8,5,8)	**3.04**	2.11	3.20	1.77
(9,6,4)	**4.13**	3.41	3.23	1.74
(9,6,8)	**2.94**	2.20	3.23	1.74
(10,7,4)	**4.55**	3.53	3.52	1.77
(10,7,8)	**3.28**	2.29	3.52	1.77
(11,8,4)	**4.70**	3.78	3.13	1.68
(11,8,8)	**3.08**	2.31	3.13	1.68
(13,10,4)	**4.86**	3.71	3.50	1.71
(13,10,8)	3.21	2.37	**3.50**	1.71

important to understand the impact of optimizing the encoding bitmatrix and procedure, which was our main focus. In this section, we present the decoding performance of various methods, along the similar manner as for the encoding performance. Only the performance for the worst case failure pattern (the most computationally expensive case) is reported, when m data symbols are lost.

As seen in Table 9, the proposed approach can provide better decoding throughput comparing to vectorized XOR-based Cauchy Reed-Solomon code and vectorized GF-based RS code, except for some cases when $m=2$. For codes with three parities, it can be seen from Table 10 that the decoding throughput of the proposed approach still outperforms well-known codes in the literature specifically designed for this case. For codes with two parities, as shown in Table 7, the

Table 11: Decoding throughputs (GB/s): Two parities

(n,k,w)	Proposed	Vectorized XOR-based CRS code	Vectorized Raid-6	EVEN ODD [2]	RDP [6]
(7,5,4)	5.52	4.85	n/a	6.66	**7.28**
(7,5,8)	3.87	4.65	2.64	6.66	**7.28**
(8,6,4)	5.43	5.14	n/a	7.42	**8.00**
(8,6,8)	5.45	4.86	2.67	7.42	**8.00**
(9,7,4)	6.03	5.37	n/a	7.65	**8.13**
(9,7,8)	4.46	5.06	2.73	7.65	**8.13**
(10,8,4)	5.88	5.73	n/a	7.93	**8.44**
(10,8,8)	4.89	5.11	2.77	7.93	**8.44**
(12,10,4)	6.23	5.89	n/a	7.49	**9.10**
(12,10,8)	4.45	5.52	2.81	7.49	**9.10**

decoding throughput of proposed approach is usually lower than EVEN-ODD and RDP codes.

In summary, the optimized encoding procedure we propose does not appear to significantly impact the performance of the decoding performance in most cases (when $m \geq 3$), which itself is a less important performance measure in practice than the encoding performance that we focus on in this work.

6 Conclusion

We performed a comprehensive study of the erasure coding acceleration techniques in the literature. A set of tests was conducted to understand the improvements and the relation among these techniques. Based on these tests, we consider combining the existing techniques and jointly optimize the bitmatrix. The study led us to a simple procedure: produce a computation schedule based on an optimized bitmatrix (using a cost function matching the computation strategy and workstation characteristic), together with the BN and WM (or UM) technique, then use vectorized XOR operation in the computation schedule. The proposed approach is able to provide improvement over most existing approaches, particularly when the number of parity is greater than two. One particularly important insight of our work is that vectorization at the XOR-level using the bitmatrix framework is a much better approach than vectorization of the finite field operations in erasure coding, not only because of the better throughput performance, but also because of the simplicity in migration to new generation CPUs.

References

[1] Library for efficient modeling and optimization in networks. `http://lemon.cs.elte.hu/trac/lemon`, 2003.

[2] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. EVENODD: An efficient scheme for tolerat-

ing double disk failures in RAID architectures. *IEEE Transactions on computers*, 44(2):192–202, 1995.

[3] Johannes Blomer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, University of California at Berkeley, 1995.

[4] Kevin D. Bowers, Ari Juels, and Alina Oprea. Hail: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 187–198. ACM, 2009.

[5] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[6] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.

[7] Cheng Huang, Jin Li, and Minghua Chen. On optimizing XOR-based codes for fault-tolerant storage applications. In *Proceedings of Information Theory Workshop*, pages 218–223, 2007.

[8] Cheng Huang and Lihao Xu. STAR: An efficient coding scheme for correcting triple storage node failures. *IEEE Transactions on Computers*, 57(7):889–901, 2008.

[9] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 190–201. ACM, 2000.

[10] Shu Lin and Daniel J. Costello. *Error control coding*. Pearson Education India, 2001.

[11] Witold Litwin and Thomas Schwarz. LH*RS: A high-availability scalable distributed data structure using Reed Solomon codes. In *Proceedings of the ACM SIGMOD Record*, volume 29, pages 237–248, 2000.

[12] Jianqiang Luo, Mochan Shrestha, Lihao Xu, and James S. Plank. Efficient encoding schedules for XOR-based erasure codes. *IEEE Transactions on Computers*, 63(9):2259–2272, 2014.

[13] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The Quantcast file system. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.

[14] James S. Plank. The RAID-6 liberation code. In *Proceedings of the 6th Usenix Conference on File and Storage Technologies*, pages 97–110, 2008.

[15] James S. Plank. XOR's, lower bounds and MDS codes for storage. In *2011 IEEE Information Theory Workshop (ITW)*, pages 503–507, 2011.

[16] James S. Plank, Kevin M. Greenan, and Ethan L. Miller. Screaming fast Galois field arithmetic using intel SIMD instructions. In *Proceedings of the 11st Usenix Conference on File and Storage Technologies*, pages 299–306, 2013.

[17] James S. Plank, Catherine D. Schuman, and B. Devin Robison. Heuristics for optimizing matrix-based erasure codes for fault-tolerant storage systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, 2012.

[18] James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

[19] James S. Plank and Lihao Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *Proceedings of Fifth IEEE International Symposium on Network Computing and Applications*, pages 173–180, 2006.

[20] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[21] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proceedings of the International Workshop on Peer-to-Peer Systems*, pages 328–337, 2002.

[22] Zooko Wilcox-O'Hearn and Brian Warner. Tahoe–The least-authority filesystem. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 21–26, 2008.

OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems

Xiaolu Li[†], Runhui Li[†], Patrick P. C. Lee[†], and Yuchong Hu[‡]
[†]*The Chinese University of Hong Kong* [‡]*Huazhong University of Science and Technology*

Abstract

Erasure coding becomes a practical redundancy technique for distributed storage systems to achieve fault tolerance with low storage overhead. Given its popularity, research studies have proposed theoretically proven erasure codes or efficient repair algorithms to make erasure coding more viable. However, integrating new erasure coding solutions into existing distributed storage systems is a challenging task and requires non-trivial re-engineering of the underlying storage workflows. We present OpenEC, a unified and configurable framework for readily deploying a variety of erasure coding solutions into existing distributed storage systems. OpenEC decouples erasure coding management from the storage workflows of distributed storage systems, and provides erasure coding designers with configurable controls of erasure coding operations through a directed-acyclic-graph-based programming abstraction. We prototype OpenEC on two versions of HDFS with limited code modifications. Experiments on a local cluster and Amazon EC2 show that OpenEC preserves both the operational performance and the properties of erasure coding solutions; OpenEC can also automatically optimize erasure coding operations to improve repair performance.

1 Introduction

Erasure coding provides a low-cost redundancy mechanism for fault-tolerant storage, and is now widely deployed in today's distributed storage systems (DSSs). Examples include enterprise-level DSSs [15,21,30] and many open-source DSSs [1,3,7,31,54,55]. Unlike replication that simply creates identical data copies for redundancy protection, erasure coding introduces much less storage overhead through the coding operations of data copies, while preserving the same degree of fault tolerance [53]. Modern DSSs mostly realize erasure coding based on the classical Reed-Solomon (RS) codes [43], yet RS codes have high performance penalty, especially in repairing lost data when failures happen. Thus, research studies have proposed new erasure coding solutions with improved performance, such as erasure codes with theoretical guarantees and efficient repair algorithms that are applicable to general erasure-coding-based storage (§6).

However, deploying new erasure coding solutions in DSSs is a daunting task. Existing studies often integrate new erasure coding solutions into specific DSSs by re-engineering the DSS workflows (e.g., the read/write paths). The tight coupling between erasure coding management and the DSS workflows makes new erasure coding solutions hard to be generalized for other DSSs and further enhanced. Some DSSs with built-in erasure coding features (e.g., HDFS with erasure coding [1,5], Ceph [54], and Swift [7]) provide certain configuration capabilities, such as interfaces for implementing various erasure codes and controlling erasure-coded data placement, yet the interfaces are rather limited and it is non-trivial to extend the DSSs with more advanced erasure codes and repair algorithms (§2.2). How to fully realize the power of erasure coding in DSSs remains a challenging issue.

We present OpenEC, a unified and configurable framework for erasure coding management in DSSs, with the primary goal of bridging the gap between designing new erasure coding solutions and enabling the feasible deployment of such new solutions in DSSs. Inspired by software-defined storage [16,48,51], which aims for configurable storage management without being constrained by the underlying storage architecture, we apply this concept into erasure coding management. Our main idea is to decouple erasure coding management from the DSS workflows. Specifically, OpenEC runs as a middleware system between upper-layer applications and the underlying DSS, and is responsible for performing all erasure coding operations on behalf of the DSS. Such a design relaxes the stringent dependence on the erasure coding support of DSSs. More importantly, OpenEC takes the full responsibility of erasure coding management, and hence provides flexibility for erasure coding designers to (i) incorporate a variety of erasure coding solutions, (ii) configure the workflows of erasure coding operations, and (iii) decide the placement of both erasure-coded data and erasure coding operations across storage nodes. Our contributions are summarized as follows:

- We propose a new programming model for erasure coding implementation and deployment. Our model builds on an abstraction called an ECDAG, a directed acyclic graph that defines the workflows of erasure coding operations. We show how we feasibly realize a general erasure coding solution through the ECDAG abstraction.
- We design OpenEC, which translates an ECDAG into erasure coding operations atop a DSS. OpenEC supports encoding operations on or off the write path as well as various state-of-the-art repair operations. In particular, it can automatically optimize an ECDAG for hierarchical topologies to improve repair performance.

- We implement a prototype of OpenEC on HDFS-RAID [5] and Hadoop 3.0 HDFS (HDFS-3) [1]. Its integrations into HDFS-RAID and HDFS-3 only require limited code changes (with no more than 450 LoC).
- We evaluate OpenEC on a local cluster and Amazon EC2. OpenEC incurs negligible performance overhead in DSS operations, supports various state-of-the-art erasure codes and repair algorithms, and increases the repair throughput by at least 82% through automatically customizing an ECDAG for a hierarchical topology.

The source code of our OpenEC prototype is available at: **http://adslab.cse.cuhk.edu.hk/software/openec**.

2 Background and Motivation

2.1 Erasure Coding Basics

Consider a DSS that comprises multiple *storage nodes* and organizes data in units of *blocks*. We construct erasure coding as an (n,k) code with two configurable parameters n and k, where $k < n$. For every k fixed-size original blocks (called *data blocks*), an (n,k) code encodes them into $n-k$ redundant blocks of the same size (called *parity blocks*), such that any k out of the n erasure-coded blocks (including both data and parity blocks) can decode the k data blocks; that is, any $n-k$ block failures can be tolerated. We call the collection of n erasure-coded blocks a *coding group*. A DSS encodes different sets of k data blocks independently, and distributes the n erasure-coded blocks of each coding group across n storage nodes to protect against any $n-k$ storage node failures. In this paper, our discussion focuses on the coding operations (i.e., encoding or decoding) of a single coding group.

For performance reasons, a DSS implements coding operations in small-size units called *packets*, while the read/write units are in blocks; for example, our experiments set the default packet and block sizes as 128 KiB and 64 MiB, respectively). It divides a block into multiple packets, and encodes the packets at the same block offsets in a coding group together. Thus, instead of first reading the whole blocks to start coding operations, a DSS can perform packet-level coding operations, while reading the whole blocks, in a pipelined manner. To simplify our discussion, we use blocks as the units of coding operations, and only differentiate packets and blocks in our implementation (§4.5).

Given the prevalence of failures, repairs are frequent operations in DSSs [40]. We consider two types of repairs: (i) *degraded reads*, which decode the unavailable data blocks that are being requested, and (ii) *full-node recovery*, which decodes all lost blocks of a failed storage node. Since repairs trigger substantial traffic [40], achieving high repair performance is important in erasure coding deployment. RS codes [43] are the most popular erasure codes that are widely used in production [5, 7, 15, 31, 54, 55], but they incur high repair costs. Thus, many repair-friendly erasure codes have been proposed. Since single-failure repairs (i.e., repairing a single lost block of a coding group in degraded reads or a single failed node in full-node recovery) are the most common repair scenarios [21, 40], existing repair-friendly erasure codes aim to minimize the repair bandwidth or I/O in single-failure repairs. Examples are regenerating codes [14], including minimum-storage regenerating (MSR) and minimum-bandwidth regenerating (MBR) codes, as well as locally repairable codes (LRCs) [21, 23, 44, 49].

Our work focuses on practical erasure codes. In particular, we target *linear* codes, which include RS codes, MSR and MBR codes, as well as LRCs. Linear codes perform linear coding operations based on the Galois field arithmetic [17]. Mathematically, for an (n,k) code, let $d_0, \cdots, d_{k-1}$ be the k data blocks, and $p_0, \cdots, p_{n-k-1}$ be the $n-k$ parity blocks. Each parity block p_j $(0 \leq j \leq n-k-1)$ can be expressed as $p_j = \sum_{i=0}^{k-1} \gamma_{ji} d_i$, where γ_{ji} is some coding coefficient for computing p_j. Note that the linear operations are *additive associative* (i.e., independent of how additions are grouped).

Also, our work addresses *sub-packetization*, which is used in various designs of MSR and MBR codes [14, 18, 32, 39, 42, 45, 50, 52]. Sub-packetization divides each block into smaller-size sub-blocks, so that repairs can be done by retrieving sub-blocks rather than whole blocks.

Most DSSs assume that all erasure-coded blocks are *immutable* and do not support in-place updates. Thus, we focus on four basic operations: writes, normal reads, degraded reads, and full-node recovery (§4.2), while we address in-place updates in future work.

2.2 Limitations of Erasure Coding Management

Modern DSSs now support erasure coding, yet existing erasure coding management in such DSSs remains stringent and still faces practical limitations. To motivate our study, we review six state-of-the-art DSSs that currently realize erasure-coded storage: HDFS-RAID [5], HDFS-3 [1], QFS [31], Tahoe-LAFS [55], Ceph [54], and Swift [7]. HDFS-RAID is the erasure coding extension of HDFS [46] in the earlier version of Hadoop. Here, we focus on Facebook's HDFS-RAID implementation [3], which builds on Hadoop version 0.20. HDFS-3 builds on the newer Hadoop version 3.0, which includes erasure coding by design. QFS resembles HDFS and includes erasure coding by design. All HDFS-RAID, HDFS-3, and QFS organize data in fixed-size blocks. In contrast, Tahoe-LAFS, Ceph, and Swift organize data in variable-size objects and partition each object into equal-size data blocks for erasure coding.

(L1) Limited support for adding advanced erasure codes: Existing DSSs provide encoding/decoding interfaces for implementing new erasure codes. However, most DSSs do not provide interfaces for adding erasure codes with sub-packetization (e.g., MSR and MBR codes [14, 18, 32, 39, 42, 45, 50, 52]) and handling erasure-coded blocks at the granularity of sub-blocks, while only recently Ceph includes the sub-packetization feature in its master codebase [52]. Also,

recent erasure codes [19,38] address the hierarchical nature of DSSs to reduce cross-rack [19] (or cross-cluster [38]) repair traffic, yet realizing such hierarchy-aware erasure codes needs modifications to the DSS workflows.

(L2) Limited configurability for workflows of coding operations: Enabling configurable workflows of coding operations allows better resource usage within a DSS. Take repairs (degraded reads or full-node recovery) as an example. DSSs execute repairs at different entities upon the detection of failures. For a degraded read, it is executed at the client (in HDFS-RAID, HDFS-3, QFS, and Tahoe-LAFS), the proxy (in Swift), or a storage node (in Ceph); for full-node recovery, it is executed at either storage nodes (in HDFS-RAID, HDFS-3, QFS, Ceph, and Swift) or the client (in Tahoe-LAFS). Both degraded reads and full-node recovery operate in a *fetch-and-compute* manner, in which the entity that executes the repair will retrieve available blocks from other non-failed storage nodes and reconstruct the lost blocks. On the other hand, besides the fetch-and-compute approach, we cannot configure a DSS to adopt different repair workflows or distribute the repair loads across storage nodes. For example, recent repair algorithms [25,29] decompose a single-block repair operation into partial sub-block repair operations that are parallelized across storage nodes for better bandwidth usage, but existing DSSs do not support this feature by design.

(L3) Limited configurability for placement of coding operations: All DSSs we consider ensure that the n erasure-coded blocks of each coding group are stored in n distinct storage nodes, and most of them additionally allow configurable block placement. For example, both HDFS-RAID and HDFS-3 provide a base class for configuring block placement policies; QFS provides an *in-rack* placement option to store multiple blocks in a rack; Ceph uses *placement groups*, while Swift uses *object rings*, to control how erasure-coded blocks are placed in different storage nodes.

However, existing DSSs focus on how erasure-coded blocks are placed after encoding, but do not specify where to perform the coding operations. For example, in encoding operations, we may want to co-locate the computations of parity blocks at one storage node (rather than distribute the computations across different storage nodes) to limit the I/Os of retrieving data blocks. Also, the repair algorithms in [25,29] require some storage nodes that store available data blocks to first compute partially decoded blocks and send the results to other storage nodes for further decoding. In this case, we need to place the partial decoding operations at specific storage nodes. Such fine-grained placement of coding operations is currently not supported in existing DSSs.

2.3 Lessons Learned and Goals

The root cause of the limitations in §2.2 is that the current erasure coding management is tightly coupled with the DSS workflows. Realizing erasure coding in DSSs needs to address how coding operations are performed (i.e., the control flow) and how erasure-coded blocks are stored and accessed (i.e., the data flow). The current practice is that erasure coding designers only define an erasure code and its coding operations (e.g., the coding coefficients used in coding operations), while DSS developers require dedicated engineering efforts to integrate the coding operations into the read/write paths of DSSs without compromising the correctness of upper-layer applications. Such tight coupling makes the extensions of erasure coding features inflexible.

OpenEC decouples erasure coding management from the underlying DSS by providing a unified and configurable framework for erasure coding management, such that erasure coding designers can leverage OpenEC to realize new erasure coding solutions and configure the workflows of coding operations, without worrying how they are integrated into the DSS workflows. Specifically, OpenEC addresses the limitations in §2.2 with the following goals: (i) extensibility of new erasure codes; (ii) configurable workflows of coding operations; and (iii) configurable placement of both erasure-coded blocks and coding operations. To achieve these goals, OpenEC builds on a programming model for erasure coding management, as elaborated in the following sections.

3 Programming Model

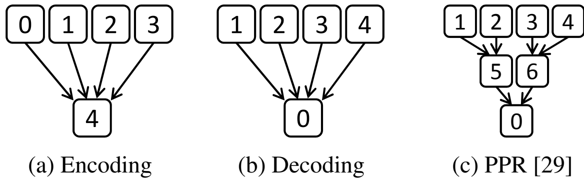
We propose a programming model that allows erasure coding designers to not only define an erasure code structure and its coding operations, but also configure how coding operations are performed in a DSS. We present a new erasure coding abstraction called an ECDAG (§3.1), followed by three primitives for constructing an ECDAG (§3.2). We then propose a programming interface for realizing an erasure code based on the ECDAG abstraction (§3.3).

3.1 ECDAG Overview

At a high level, an ECDAG is a directed acyclic graph that describes the workflows of coding operations of a coding group of an erasure code. Each vertex represents a block in the coding group, and the connections among vertices describe how vertices are related by linear combinations. To address the limitations in §2.2, we design ECDAGs to work for general linear codes (L1 addressed). Also, we can construct different ECDAGs to configure *how* and *where* coding operations are performed (L2 and L3 addressed, respectively).

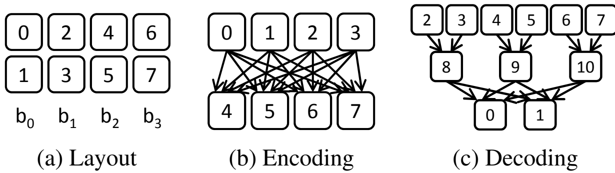
Consider a coding group of an (n,k) code with n erasure-coded blocks; to simplify our discussion, we do not consider sub-packetization first. We index the blocks from 0 to $n-1$, and let b_i denote the block with index i. Without loss of generality, we refer to $b_0,\cdots,b_{k-1}$ as k data blocks, and $b_k,\cdots,b_{n-1}$ as $n-k$ parity blocks. In some cases (see below), the coding operations may generate some intermediately computed blocks that will not be finally stored (as opposed to blocks $b_0,b_1,\cdots,b_{n-1}$). We call such blocks *virtual blocks*, and denote a virtual block by $b_{i'}$ for some $i' \geq n$.

In an ECDAG, let v_i ($i \geq 0$) be a vertex that maps to block



(a) Encoding (b) Decoding (c) PPR [29]

Figure 1: Example of an ECDAG for a $(5,4)$ code.



(a) Layout (b) Encoding (c) Decoding

Figure 2: Example for an ECDAG for a (4,2) code with $w = 2$.

b_i; we call a vertex $v_{i'}$ ($i' \geq n$) that maps to a virtual block $b_{i'}$ a *virtual vertex*. Let $e_{i,j}$ ($i, j \geq 0$) be a directed edge from v_i to v_j indicating that b_i is an input to the linear combination for computing b_j. Each edge is associated with a coding coefficient for the linear combination. If there exists an edge $e_{i,j}$, we say that v_j is the *parent* of v_i, while v_i is a *child* of v_j. A vertex can have any number of parents and children.

Both encoding and decoding operations are each associated with an ECDAG. The ECDAG for encoding is constructed at the beginning of the encoding operation to describe how data blocks are linearly combined to form each parity block. In contrast, the ECDAG for decoding is constructed on demand depending on what blocks are currently available.

For example, consider a $(5,4)$ code (i.e., (4+1)-RAID-5). Figure 1(a) shows the ECDAG for the encoding operation, which states that the parity block b_4 is a linear combination of the four data blocks b_0, b_1, b_2, and b_3. Suppose now that block b_0 is lost. Figure 1(b) shows the ECDAG for the decoding operation for b_0, which can be computed from other available blocks b_1, b_2, b_3, and b_4.

We can parallelize partial decoding operations as in PPR [29] by constructing another ECDAG for decoding b_0 (see Figure 1(c)), in which we first compute in parallel the partially decoded blocks b_5 and b_6 (both of which are virtual blocks) from b_1 and b_2 and from b_3 and b_4, respectively, followed by computing b_0 from b_5 and b_6. This shows that we can flexibly configure coding operations by constructing different ECDAGs. Note that PPR needs to compute b_5 and b_6 at the storage nodes where data blocks (e.g., b_2 and b_4, respectively) are stored (see [29] for details). We address this issue in §3.2.

We can also construct an ECDAG for erasure codes with sub-packetization. Let w be the number of sub-blocks per block ($w = 1$ means no sub-packetization). We index the sub-blocks of block b_0 from 0 to $w-1$, those of b_1 from w to $2w-1$, and so on. Each vertex v_i ($i \geq 0$) now corresponds to the sub-block with index i, while any vertex $v_{i'}$ for $i' \geq nw$ is a virtual vertex. For example, consider the $(4,2)$ MISER code [45] (an MSR code based on interference alignment), where $w = 2$. Figure 2(a) shows how the sub-blocks are indexed. Figure 2(b) shows the ECDAG for the encoding operation, in which the sub-blocks of parity blocks b_2 and b_3 are computed from the sub-blocks of data blocks b_0 and b_1. Suppose that block b_0 is lost. Figure 2(c) shows the ECDAG for decoding b_0 based on MISER codes [45], in which we first compute an encoded sub-block from each of other available blocks b_1, b_2, and b_3 (represented by the virtual vertices v_8, v_9, and v_{10}, respectively), followed by using the encoded sub-blocks to decode the lost sub-blocks of b_0.

```
void Join(int pidx, vector<int> cidxs, vector<int> coefs);
int BindX(vector<int> idxs);
void BindY(int pidx, int cidx);
```

Listing 1: Primitives for ECDAG construction.

3.2 ECDAG Primitives

An ECDAG can be constructed from three primitives: `Join`, `BindX`, and `BindY`. `Join` is used for constructing an ECDAG, while `BindX` and `BindY` control the placement of coding operations. Listing 1 shows their definitions in C++ format.

Join: It specifies how a parent vertex (with index `pidx`) is formed by the linear combinations of a list of child vertices (with indices in `cidxs`) and the corresponding coding coefficients (in `coefs`). For example, we deploy the $(6,4)$ RS code and encode four data blocks b_0, b_1, b_2, and b_3 into two new parity blocks b_4 and b_5. We can construct an ECDAG with `Join` as follows (see Figure 3(a)):

```
ECDAG* ecdag = new ECDAG();
ecdag->Join(4, {0,1,2,3}, {1,1,1,1});
ecdag->Join(5, {0,1,2,3}, {1,2,4,8});
```

BindX: It co-locates the coding operations of multiple vertices (with indices in `idxs`) that reside at the same level of an ECDAG (i.e., in the x-direction), so as to reduce I/O in coding operations. For example, in Figure 3(a), suppose that the data blocks being encoded are stored in different storage nodes. Without `BindX`, we need to compute b_4 and b_5 separately and retrieve each data block twice. Instead, we can call `BindX` on vertices v_4 and v_5 to create a new virtual vertex v_6 as follows (see Figure 3(b)):

```
int vidx = ecdag->BindX({4,5});
```

This indicates that blocks b_4 and b_5 are first computed together at the same storage node before being distributed to different storage nodes. Now we only need to retrieve each data block once. Note that the index of v_6 (i.e., 6) is generated randomly and returned as `vidx` by `BindX`.

BindY: It co-locates the coding operations of a parent vertex (with index `pidx`) and its child vertex (with index `cidx`) at different levels (i.e., in the y-direction). Consider the same example in Figure 3(b) after we call `BindX`. We can call `BindY` on vertices v_0 and v_6 as follows (see Figure 3(c)):

```
ecdag->BindY(vidx, 0);
```

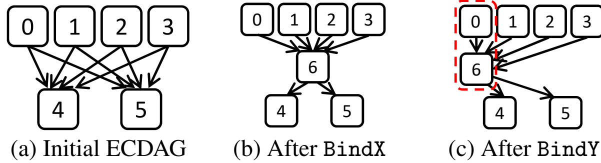


Figure 3: Construction of ECDAGs for the $(6,4)$ RS code.

```
class ECBase {
  int n, k, w;  // coding parameters
  vector<int> ecoefs; // encoding coefficients
public:
  ECDAG* Encode();
  ECDAG* Decode(vector<int> from, vector<int> to);
  vector<vector<int>> Place();
};
```

Listing 2: Erasure coding programming interface.

Thus, we compute parity blocks b_4 and b_5 at the same storage node that stores b_0, thereby saving the I/Os of retrieving b_0.

Note that `BindY` enables us to implement the repair algorithms (e.g., PPR [29] and repair pipelining [25]) that need to compute partially decoded blocks at the storage nodes that store the data blocks. For example, referring to Figure 1(c) for PPR, we can call `BindY` on v_2 and v_5, and on v_4 and v_6, to co-locate the computations of the partially decoded blocks b_4 and b_5 at the storage nodes that store b_2 and b_4, respectively.

Remarks: We provide flexibility for erasure coding designers to construct any ECDAG using the above three primitives, yet this also puts burdens on erasure coding designers to configure coding operations. Nevertheless, OpenEC can also automatically call `BindX` and `BindY` on some specific subgraph structures of an ECDAG (§4.4).

3.3 Erasure Coding Interfaces

We provide a programming interface for realizing an erasure code. Unlike the traditional approach that takes data blocks as input and generates parity blocks, we program an erasure code through the construction of ECDAGs. OpenEC then parses the ECDAGs to perform the actual coding operations and store the erasure-coded blocks.

Listing 2 shows the erasure coding programming interface as a base class `ECBase`. To realize an erasure code, we (as erasure coding designers) inherit `ECBase` and first define all necessary member variables (e.g., n, k, w, and encoding coefficients) in the constructor method as in traditional erasure code programming. Note that we can store encoding coefficients in a *generator matrix* [35] and compute decoding coefficients later based on the available blocks. We then implement three functions, namely `Encode`, `Decode`, and `Place`.

Encode: It constructs an ECDAG that describes the encoding operation. For example, to encode the $(6,4)$ RS code based on Figure 3(c), we can construct an ECDAG as in Listing 3.

Decode: It constructs an ECDAG that takes the available blocks (with indices in `from`) as input and decodes any lost

```
ECDAG* Encode() {
  ECDAG* ecdag = new ECDAG();
  ecdag->Join(4, {0,1,2,3}, {1,1,1,1});
  ecdag->Join(5, {0,1,2,3}, {1,2,4,8});
  int vidx = ecdag->BindX({4,5});
  ecdag->BindY(vidx, 0);
  return ecdag;
}
```

Listing 3: Encode function.

```
ECDAG* Decode(vector<int> from, vector<int> to) {
  ECDAG* ecdag = new ECDAG();
  vector<int> dcoefs;  // decoding coefficients
  // compute dcoefs based on the available blocks
  ecdag->Join(to[0], from, dcoefs);
  return ecdag;
}
```

Listing 4: Decode function.

```
vector<vector<int>> Place() {
  vector<vector<int>> groups;
  for (int i=0; i<n/2; ++i) groups[0].push_back(i);
  for (int i=n/2; i<n; ++i) groups[1].push_back(i);
  return groups;
}
```

Listing 5: Place function.

blocks (with indices in `to`). For example, we can implement `Decode` for a single lost block as in Listing 4, in which the decoding coefficients are computed based on the available blocks in `from`. In general, `Decode` constructs an ECDAG for one of the two scenarios: (i) decoding one lost block, in which we can choose an efficient single-failure repair approach (e.g., see Figure 2(c) for the $(4,2)$ MISER code); or (ii) decoding multiple lost blocks, in which we can choose any k available blocks (e.g., the first k blocks in `from`) to compute the decoding coefficients and decode all lost blocks.

Place: It configures how erasure-coded blocks are placed with hierarchy awareness. In addition to storing erasure-coded blocks in different storage nodes, we can configure how the blocks are grouped (e.g., in the same rack in rack-based DSSs). This supports fine-grained block placement configurations as in existing DSSs (§2.2), and allows the realization of hierarchy-aware erasure codes [19,38]. For example, we can divide n erasure-coded blocks into two groups via `Place` as in Listing 5. Note that `BindX` and `BindY` in ECDAG construction (§3.2) address the placement of coding operations, while `Place` addresses the placement of erasure-coded blocks.

4 OpenEC Design

We design OpenEC to provide erasure coding management for a DSS. We show its architecture (§4.1) and supported basic operations (§4.2). We then describe how it parses ECDAGs to realize coding operations (§4.3). We further show how it automatically optimizes coding operations (§4.4). We conclude this section with the implementation details (§4.5).

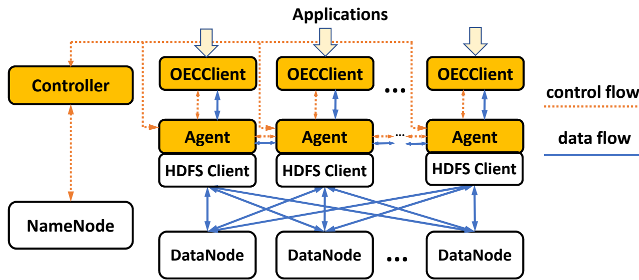


Figure 4: OpenEC architecture based on HDFS.

4.1 Architectural Overview

OpenEC runs as a middleware system atop a DSS. As a proof of concept, we design OpenEC atop two implementations of HDFS [46]: HDFS-RAID [5] and HDFS-3 [1][1]. HDFS (including both HDFS-RAID and HDFS-3) comprises a *NameNode* that coordinates the storage in units of *blocks* across multiple *DataNodes* (storage nodes). Figure 4 shows how OpenEC is integrated into HDFS. OpenEC comprises a centralized *controller*, which coordinates multiple *agents*. An application interacts with OpenEC via an *OECClient*.

Controller: The controller parses ECDAGs and instructs all agents how to perform coding operations and store erasure-coded blocks. It keeps erasure coding metadata and all ECDAGs in local disk for persistence. There are three types of metadata: (i) the information of blocks associated with each file; (ii) the information of blocks associated with each coding group; and (iii) the block locations. The controller interacts with the NameNode in two aspects. First, it accesses or updates the block locations of the NameNode to configure the placement of blocks. Second, it receives the reports of lost blocks from the NameNode and coordinates the repair operations among the agents.

We assume that the controller is reliable (i.e., no single-point-of-failure). Our measurements show that the controller can serve a request of parsing an ECDAG for coding operations in less than 0.3 ms in our local cluster (§5), and hence it incurs limited overhead to basic operations.

Agent: Each agent performs coding operations as instructed by the controller. It accesses the erasure-coded blocks in HDFS through the HDFS client interface. Note that agents can communicate among themselves to perform coding operations and exchange erasure-coded blocks. We currently deploy each agent at a DataNode, so that the agent can access the local storage of the DataNode without network transfers.

OECClient: Each OECClient is associated with an agent, and serves as an interface between an upper-layer application and the agent. It connects to the agent via Redis-based communication (§4.5). An application now accesses HDFS through an OECClient instead of an HDFS client.

[1] We also implement OpenEC atop QFS [31]. See [27] for details.

4.2 Basic Operations

OpenEC supports four basic operations: (i) writes; (ii) normal reads; (iii) degraded reads; and (iv) full-node recovery.

Writes: Note that HDFS-3 supports *online encoding* (i.e., clients perform encoding on the write path), while HDFS-RAID supports *offline encoding* (i.e., clients first write the data blocks in uncoded form, and the data blocks are later encoded in the background). OpenEC is currently designed to support both online and offline encoding. An OECClient specifies which encoding mode to use in a write request. For online encoding, OpenEC encodes data on a per-file basis. When an OECClient writes a file, its agent encodes every k data blocks into $n-k$ parity blocks and writes the n erasure-coded blocks to n DataNodes through the HDFS client. For offline encoding, an OECClient first writes file data via its agent to HDFS. When OpenEC receives an encoding request, the controller parses the specified ECDAG (§4.3) and instructs all agents to perform encoding, such that every k blocks are encoded into n erasure-coded blocks as a coding group.

Normal reads: An OECClient issues normal reads (under no failures) via its agent, which connects to the DataNodes that store the uncoded data blocks and retrieves the data blocks from the DataNodes.

Degraded reads: An OECClient issues degraded reads (under failures) via its agent, which connects to non-failed DataNodes and retrieves the available blocks for decoding the lost blocks based on the ECDAG specification.

Full-node recovery: The controller coordinates the full-node recovery operation. When it receives a report of lost blocks from the NameNode, it informs the agents to repair the lost blocks based on the ECDAG specification.

4.3 Parsing an ECDAG

OpenEC parses ECDAGs to perform coding operations in writes (online or offline encoding), degraded reads, and full-node recovery. Given an ECDAG, OpenEC decomposes a coding operation into multiple *tasks*, each of which is executed by an agent. Each task operates in blocks (or sub-blocks in sub-packetization). There are four types of tasks:

- **Load:** It loads a block into memory from the agent's input stream, which could be either the OECClient if the block is from upper-layer applications, or the HDFS client if the block is from HDFS.
- **Fetch:** It retrieves blocks from other agents.
- **Compute:** It computes a block based on the linear combination of blocks and coding coefficients.
- **Persist:** It either writes a block to HDFS via the HDFS client, or returns the block to an OECClient.

Parsing procedure: OpenEC performs *topological sorting* of an ECDAG (based on depth-first search) to identify the vertex sequence of coding operations. It then assigns tasks to each vertex based on the ECDAG structure. Depending on the

Vertices	Nodes	Tasks
v_0	C	**Load** b_0
v_1	C	**Load** b_1
v_2	C	**Load** b_2
v_3	C	**Load** b_3
v_6	C	**Compute** b_4 from $\{b_0, b_1, b_2, b_3\}$ with coding coefficients $\{1,1,1,1\}$; **Compute** b_5 from $\{b_0, b_1, b_2, b_3\}$ with coding coefficients $\{1,2,4,8\}$
v_4	C	–
v_5	C	–
–	C	**Persist** b_0; **Persist** b_1; **Persist** b_2; **Persist** b_3; **Persist** b_4; **Persist** b_5

(a) Online encoding

Vertices	Nodes	Tasks
v_0	N_0	**Load** b_0
v_1	N_1	**Load** b_1
v_2	N_2	**Load** b_2
v_3	N_3	**Load** b_3
v_6	N_0	**Fetch** b_1 from N_1; **Fetch** b_2 from N_2; **Fetch** b_3 from N_3; **Compute** b_4 from $\{b_0, b_1, b_2, b_3\}$ with coding coefficients $\{1,1,1,1\}$; **Compute** b_5 from $\{b_0, b_1, b_2, b_3\}$ with coding coefficients $\{1,2,4,8\}$
v_4	N_4	**Fetch** b_4 from N_0; **Persist** b_4
v_5	N_5	**Fetch** b_5 from N_0; **Persist** b_5

(b) Offline encoding

Table 1: Vertex sequence of coding operations, including the nodes that are responsible for processing the vertices as well as the tasks that are performed.

types of basic operations, OpenEC may perform coding operations on the client side (for online encoding and degraded reads) or distribute the coding operations across storage nodes (for offline encoding and full-node recovery).

OpenEC associates tasks with different types of vertices. At a high level, the Load task is associated with a vertex without any child; the Fetch task is associated with a parent vertex that has a child vertex; the Compute task is associated with a vertex with more than one child for the linear combination; the Persist task is associated with a vertex without any parent, while it is also associated with a vertex without any child in the case of online encoding (see the example below).

Example: We show the parsing procedure via an example. Suppose that we encode four data blocks (i.e., b_0, b_1, b_2, and b_3) to generate two parity blocks (i.e., b_4 and b_5) using the $(6,4)$ RS code, based on the ECDAG in Figure 3(c) and the `Encode` function in Listing 3. Table 1 shows the vertex sequence of tasks for both online and offline encoding.

For online encoding (see Table 1(a)), the client-side agent (denoted by C) performs all coding operations. It finally persists all data blocks and parity blocks into HDFS.

For offline encoding (see Table 1(b)), OpenEC distributes the coding operations across storage nodes. To elaborate, suppose that b_i is stored in storage node N_i, for $0 \leq i \leq 5$. First, since v_0, v_1, v_2, and v_3 have no child, OpenEC creates tasks for the agents in storage nodes N_0, N_1, N_2, and N_3 to load the blocks b_0, b_1, b_2, and b_3, respectively, from HDFS (via HDFS clients) into memory. Second, since vertex v_6 is created from `BindX` on vertices v_4 and v_5, OpenEC computes both b_4 and b_5 from the blocks in the child vertices (i.e., b_0, b_1, b_2, and b_3). Also, since `BindY` is called on v_6 and v_0, OpenEC assigns the tasks of v_6 to the agent in N_0. Finally, v_4 and v_5 retrieve blocks b_4 and b_5 from v_6, respectively. Since v_4 and v_5 have no parent and are the last vertices in the topological order, they persist the blocks to HDFS.

Note that OpenEC can parallelize the coding operations on the vertices that have no dependencies on others. For example, OpenEC can simultaneously execute the tasks for v_0, v_1, v_2, and v_3, and similarly the tasks for v_4 and v_5.

4.4 Automated Optimizations

In addition to letting erasure coding designers construct ECDAGs, OpenEC can automatically customize ECDAGs for performance optimizations to save manual configuration efforts. We address this in two aspects.

Automated BindX and BindY: OpenEC can automatically call `BindX` and `BindY` for some specific subgraph structures of an ECDAG. For `BindX`, OpenEC examines all parent vertices that have more than one child vertex in an ECDAG. If multiple parent vertices have the same set of child vertices, OpenEC calls `BindX` on those parent vertices (e.g., v_4 and v_5 in Figure 3(b)). For `BindY`, for any parent vertex (with one or more child vertices), OpenEC calls `BindY` on the parent vertex and any one of the child vertices (e.g., the parent vertex v_6 and the child vertex v_0 in Figure 3(c)).

Hierarchy awareness: OpenEC can further enhance the repair performance based on the physical DSS topology. One scenario is that a DSS hierarchically organizes storage nodes in racks [19] (or clusters [38]), such that the cross-rack bandwidth is much more constrained than the inner-rack bandwidth. OpenEC can transform an ECDAG into a *pipelined ECDAG*, so as to mitigate the cross-rack traffic. Our idea is based on repair pipelining [25], which pipelines partial coding operations across multiple storage nodes. We additionally perform all partial coding operations within a rack before sending the partial coding results to another rack. To illustrate, suppose that we deploy an (n,k) RS code with $k = 6$. We want to repair a lost block b_0 from six other available blocks b_1, b_2, b_3, b_4, b_5, and b_6, such that blocks b_1, b_3, and b_5 are in one rack, while blocks b_2, b_4, and b_6 are in another rack. We also want to store the reconstructed block b_0 at the same rack as b_2, b_4, and b_6. The conventional repair approach is to retrieve all six available blocks and construct an ECDAG as in Figure 5(a). Then we need to transfer three blocks (i.e., b_1, b_3, and b_5) across racks. Instead, OpenEC can automatically construct another ECDAG as in Figure 5(b), in which it first

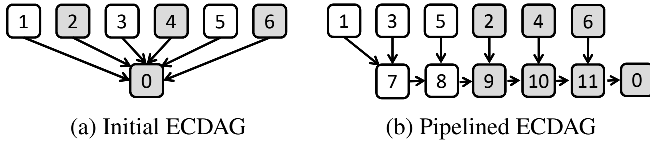


Figure 5: Example of constructing a pipelined ECDAG; vertices of the same color mean that their blocks are in the same rack.

computes the partially decoded block b_8 (corresponding to vertex v_8) based on b_1, b_3, and b_5 in the same rack, followed by combining b_8 with b_0, b_2, and b_4 in another rack to reconstruct block b_0. In this case, we only need to transfer one block (i.e., b_8) across racks.

4.5 Implementation

We implement an OpenEC prototype in C++ with around 7K LoC. We use Intel's Intelligent Storage Acceleration Library (ISA-L) [6] to implement erasure coding functionalities. Here, we highlight several implementation details of OpenEC.

From blocks to packets: OpenEC performs coding operations in units of packets to improve performance, while the read/write operations are still in units of blocks (§2.1). By default, the packet size is 128 KiB. For encoding (both online and offline), OpenEC writes n erasure-coded packets to n DataNodes; in the case of sub-packetization, each packet is divided into sub-packets. If a DataNode receives an amount of packet data equal to the HDFS block size (64 MiB by default), it *seals* the block and stores additional packets in a different block. The n sealed erasure-coded blocks then form a coding group. Note that while OpenEC is sending packets to DataNodes, it can start encoding for the next group of packets. Thus, both the sending and encoding operations can be done in parallel. Similarly, OpenEC performs decoding (for degraded reads and full-node recovery) at the packet level.

As OpenEC performs packet-level coding operations, the block layouts differ in online and offline encoding. For online encoding, OpenEC adopts a *striped* layout as in HDFS-3 [4], as it stripes file data across blocks at the granularities of packets. For offline encoding, OpenEC adopts a *contiguous* layout, as the file data is first stored in a block before encoding. Figure 6 depicts both block layouts.

Internal communication: OpenEC uses Redis [8] for internal communications among the controller, agents, and OECClient. Each agent maintains a local in-memory key-value Redis store. The controller sends the task instructions of coding operations to an agent via the Redis client, and the task instructions are buffered at the agent for subsequent processing. Agent-to-agent communications are *pull-based* via the Fetch tasks (§4.3), such that the sender agent buffers the blocks to be sent in its local Redis store, and the receiver fetches the buffer via the Redis client. Each OECClient also communicates with its associated agent via Redis.

Integration: We integrate OpenEC into HDFS-RAID and HDFS-3 as follows. We realize a new block placement policy

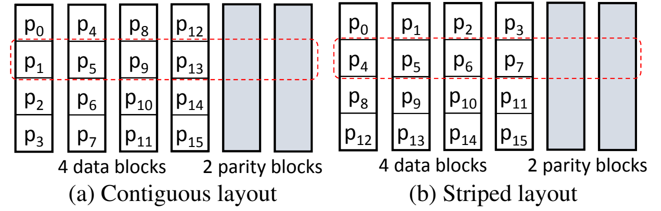


Figure 6: Block layouts for the $(6,4)$ RS code. Suppose that each block stores four packets. We partition file data into 16 packets, ordered as p_0, p_1, $\cdots$, and p_{15}. We place the packets across $k=4$ blocks. Packets at the same offset (e.g., in dashed boxes) are encoded together. In sub-packetization, each packet is further divided into sub-packets for encoding.

called `BlockPlacementPolicyOEC`, which redirects block placement requests to the controller to manage erasure-coded block placement. We also modify the `FSNamesystem` class in HDFS-RAID and the `BlockManager` class in HDFS-3 to redirect any lost block information to the controller so that OpenEC manages repair operations. Note that our integrations into HDFS-RAID and HDFS-3 only require limited modifications to their codebases, with around 300 LoC and 450 LoC, respectively[2].

5 Evaluation

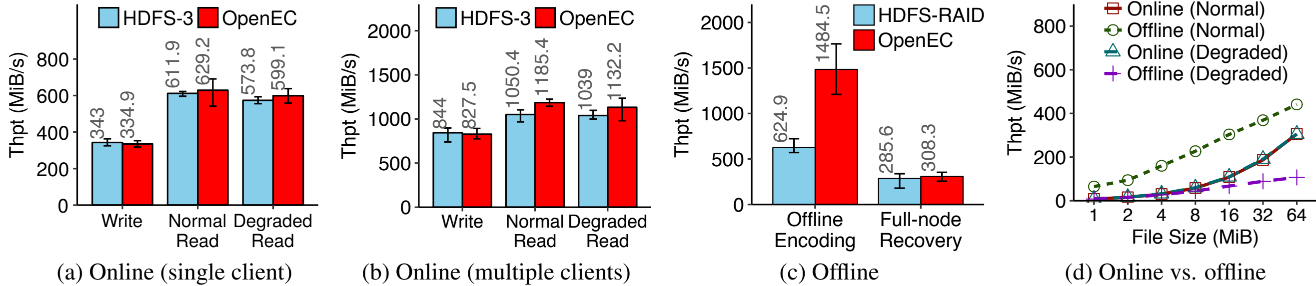
We conduct testbed experiments on OpenEC. We summarize our major findings on OpenEC: (i) it preserves the performance of HDFS-RAID and HDFS-3 in erasure coding deployment (§5.2); (ii) it supports various state-of-the-art erasure coding solutions and preserves their properties, especially in network-bound environments (§5.3); (iii) it can automatically optimize the repair performance for a hierarchical topology (§5.4); and (iv) it achieves scalable performance in real cloud environments (§5.5).

5.1 Setup

Testbeds: We evaluate OpenEC on both a local cluster (§5.2-§5.4) and Amazon EC2 (§5.5). Our local cluster testbed comprises 16 machines, each of which has a quad-core 3.4 GHz Intel Core i5-3570, 16 GiB RAM, and a Seagate ST1000DM003 7200 RPM 1 TiB SATA hard disk. All machines are interconnected via a 10 Gb/s Ethernet switch. On the other hand, our Amazon EC2 testbed comprises 30 instances of type `m5.xlarge`, connected via a 10 Gb/s network, in the US East (North Virginia) region. Each instance has four vCPUs with Intel AVX-512 instruction sets and 16 GiB RAM. Both testbeds support optimized coding operations based on ISA-L.

Default setup: We set the HDFS block size as 64 MiB and the packet size for erasure coding as 128 KiB. We set HDFS-3

[2]We compare the amounts of code changes in OpenEC with those in our previously built prototypes CORE [26] and DoubleR [19], both of which modify HDFS-RAID to realize new erasure codes. Excluding the implementation of erasure codes (e.g., coding operations), CORE and DoubleR make around 2,300 LoC and 4,100 LoC of changes to the HDFS-RAID codebase for the integration of erasure codes, respectively.



(a) Online (single client) (b) Online (multiple clients) (c) Offline (d) Online vs. offline

Figure 7: Performance of basic operations in online and offline encoding.

as the default DSS for OpenEC, except when we compare OpenEC with HDFS-RAID. Regarding the automated optimization features (§4.4), our experiments enable automated `BindX` and `BindY`, except when we evaluate the original performance of erasure codes without OpenEC optimization in §5.3 and when we evaluate `BindX` and `BindY` in §5.4. We also disable hierarchy-aware repairs until we evaluate this feature in §5.4. We assign a dedicated machine to serve both the OpenEC controller and the HDFS NameNode, while each remaining machine serves an OECClient, an OpenEC agent, an HDFS client, and an HDFS DataNode. We plot the average results over 10 runs, including the error bars showing the maximum and minimum of the 10 runs.

5.2 Performance of Basic Operations

We compare OpenEC with HDFS-RAID and HDFS-3 in terms of basic operations using our local cluster. As OpenEC adds another software layer between upper-layer applications and the underlying DSS, it may incur extra overhead. We show that such overhead (if any) is limited; in some cases, OpenEC even significantly improves performance. We also compare OpenEC with native coding performance and evaluate its performance for different block and packet sizes.

Single-client performance in online encoding: We first compare the single-client performance between HDFS-3 and OpenEC, both of which are configured with online encoding to generate erasure-coded data. Here, we use the $(9,6)$ RS code (as in QFS [31]). We first write a file of size 384 MiB (i.e., six times the block size), and issue a normal read to the file without failures. We also issue a degraded read to the file with one data block deleted. Figure 7(a) shows the throughput results of writes, normal reads, and degraded reads. Both OpenEC and HDFS-3 have similar performance: OpenEC's throughput is slightly less than HDFS-3's by 2.36% in writes, and is slightly higher than HDFS-3's by 2.83% and 4.41% in normal reads and degraded reads, respectively.

Multi-client performance in online encoding: We compare the multi-client performance between HDFS-3 and OpenEC. We run a total of five clients, each of which writes a file of size 384 MiB under the $(9,6)$ RS code. Figure 7(b) shows the aggregate throughput of all five clients in writes, normal reads, and degraded reads. OpenEC has lower aggregate throughput than HDFS-3 in writes by 1.95%, but higher aggregate throughput in normal reads and degraded reads by 12.9% and 8.97%, respectively. Nevertheless, considering the error bars in the figure, we do not see significant performance differences between OpenEC and HDFS-3.

Offline encoding: We compare the performance between HDFS-RAID and OpenEC in offline encoding. We now deploy OpenEC on HDFS-RAID for fair comparisons. We write 180 blocks, and use offline encoding to generate erasure-coded blocks using the $(9,6)$ RS code (i.e., a total of 30 coding groups). We then delete the blocks of one storage node and trigger full-node recovery. Here, we measure the offline encoding throughput (i.e., the amount of input data being encoded per unit time) and the full-node recovery throughput (i.e., the amount of lost data being recovered per unit time). Note that HDFS-RAID performs offline encoding and full-node recovery via MapReduce. To exclude the MapReduce startup overhead in our evaluation, we start an empty MapReduce job to measure its latency, and subtract this latency (which is around 20 s) in our evaluation of HDFS-RAID. Note that OpenEC does not use MapReduce in offline encoding and full-node recovery.

Figure 7(c) shows the results. Interestingly, OpenEC increases the offline encoding throughput of HDFS-RAID by 137%. We study the HDFS-RAID source code and find that the performance difference is mainly due to the extra step of HDFS-RAID in reading and re-writing all parity blocks into a single HDFS file after parity regeneration. For full-node recovery, OpenEC has slightly higher throughput than HDFS-RAID by 7.9%, yet the two systems have limited differences considering the error bars.

Online vs. offline encoding: We further compare online and offline encoding in OpenEC versus the file size, and study the performance difference between the striped layout (in online encoding) and the contiguous layout (in offline encoding). We deploy OpenEC atop HDFS-3, and show that it allows both online and offline encoding atop HDFS-3 (which currently supports online encoding only).

We consider the single-client performance, in which a client uses the $(12,8)$ RS code and writes a file of size ranging from 1 MiB to 64 MiB (assuming that the file size is divisible by eight). For online encoding, OpenEC stripes the file

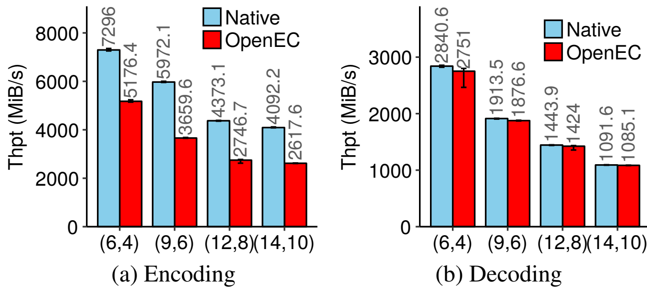


Figure 8: Comparisons with native coding operations.

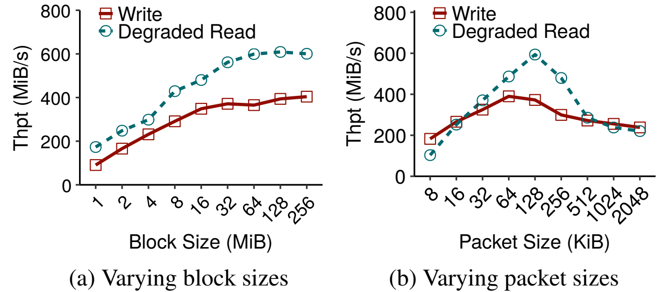


Figure 9: Impact of block and packet sizes.

in packets across eight blocks and seals the blocks after the file write is completed (note that each block is less than the default block size 64 MiB); for offline encoding, OpenEC stores the file in a block and later encodes it with seven other blocks (§4.2). We compare their performance in a normal read (without failures) and a degraded read (with one data block deleted) to the file; in offline encoding, we delete the data block that stores the file in our degraded read evaluation.

Figure 7(d) shows the results. The throughput increases with the file size, since the data transfer performance becomes more dominant as the blocks become larger. We also see the performance differences in online and offline encoding. In online encoding, both normal reads and degraded reads show similar performance, in which the client issues reads to eight blocks in parallel. In offline encoding, its normal read throughput is much higher than that in online encoding (by 44-718%), as any slowdown in one of the parallel reads to online-encoded data can degrade the overall performance. However, the degraded read throughput in offline encoding is much less than that in online encoding especially for larger file sizes, as it needs to retrieve eight blocks (i.e., seven additional blocks over the original file) to recover the file. To validate our results, we conduct similar experiments using the original erasure coding implementations in HDFS-3 and HDFS-RAID (which realize online and offline encoding, respectively) and they show similar performance differences as in OpenEC (we omit the results here in the interest of space).

Comparisons with native coding operations: We compare the computational performance of the ECDAG-based coding operations with that of the native coding operations using ISA-L in HDFS-3. Figure 8(a) shows the encoding throughput for k 64-MiB blocks under (n,k) RS codes. ECDAG-based encoding has 29-38% lower throughput than native encoding, mainly because there is additional overhead for creating multiple compute tasks for computing the $n-k$ parity blocks. Figure 8(b) shows the decoding throughput for decoding one block, in which ECDAG-based decoding has only slightly less throughput (by 0.6-3.2%) than native decoding, as there is only one compute task for decoding a single block. Nevertheless, compared to the overall read/write operations (Figure 7), the computations of ECDAG-based coding are much faster and incur limited overhead.

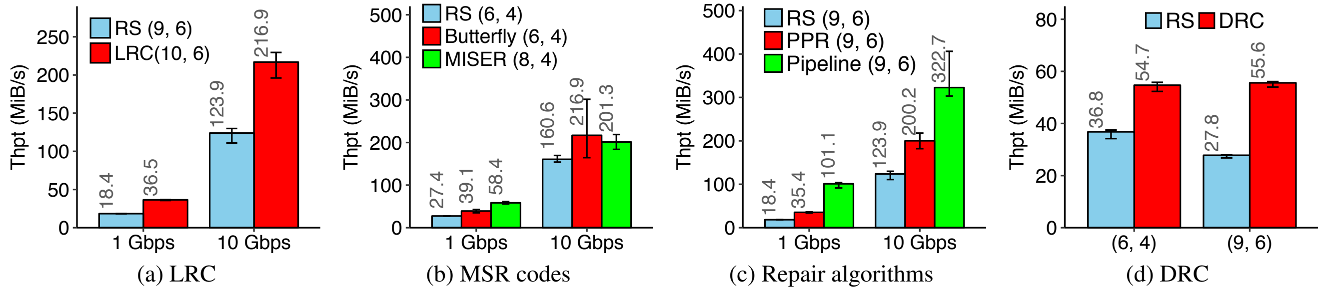
Impact of block and packet sizes: We study how the performance of OpenEC varies with block and packet sizes. We focus on the single-client throughput of online encoding and degraded reads under the $(9,6)$ RS codes as in §5.2. Figure 9(a) shows the throughput versus the block size, where the packet size is fixed as 128 KiB. The throughput of both operations increases with the block size as the disk and network bandwidths are better utilized, and stabilizes when the block size is at least 64 MiB. Figure 9(b) shows the throughput versus the packet size, where the block size is fixed as 64 MiB. The performance degrades if the packet size is too small since there are many function calls for retrieving individual packets, or if the packet size is too large since there is less parallelism. To achieve high performance, our default setup chooses the block size as 64 MiB and the packet size as 128 KiB.

5.3 Support of Erasure Coding Designs

We realize several state-of-the-art repair-friendly erasure coding solutions based on the ECDAG abstraction. Recall from §2.1 that existing repair-friendly codes are designed to minimize the repair bandwidth or I/O in single-failure repairs. Thus, we focus on evaluating their performance of repairing one lost block in a coding group under OpenEC. We configure two bandwidth settings in our local cluster: 1 Gb/s and 10 Gb/s. For the 1 Gb/s case, network transfer becomes the bottleneck (compared to coding computations and disk I/O), and we expect that the empirical performance conforms to the theoretical gains.

We use the conventional repair approach of RS codes as our baseline, in which it retrieves k blocks from k non-failed DataNodes to decode the lost block in a fetch-and-compute manner (§2.2). We compare the conventional repair approach with the following solutions:

- **LRC (Figure 10(a)):** We compare RS codes with Azure's LRC [21]. For RS codes, we set $(n,k)=(9,6)$; for LRC, we set $(n,k)=(10,6)$, in which there are two local parity blocks, each of which is encoded from a local group of three data blocks, and two global parity blocks that are encoded from all six data blocks.
- **MSR codes (Figure 10(b)):** We compare RS codes with MSR codes [14], which leverage sub-packetization to minimize the repair bandwidth. We focus on two variants of



(a) LRC (b) MSR codes (c) Repair algorithms (d) DRC

Figure 10: Support of erasure coding designs.

	LRC (10,6) vs. RS (9,6)	Butterfly (6,4) vs. RS (6,4)	MISER (8,4) vs. RS (6,4)	PPR (9,6) vs. RS (9,6)	Pipeline (9,6) vs. RS (9,6)	DRC (6,4) vs. RS (6,4)	DRC (9,6) vs. RS (9,6)
Gain	2×	1.6×	2.28×	2×	6×	1.5×	2×

Table 2: Theoretical gains of state-of-the-art erasure codes or repair algorithms over the conventional repair of RS codes.

MSR codes: MISER codes [45] (which require $n \geq 2k$) and Butterfly codes [32] (which require $n = k+2$). We consider the $(6,4)$ RS code, the $(6,4)$ Butterfly code, and the $(8,4)$ MISER code.

- **Repair algorithms (Figure 10(c)):** We study how the repair algorithms, namely PPR [29] and repair pipelining [25], improve the repair performance of RS codes by parallelizing partial repair operations. We compare them with the conventional repair under the $(9,6)$ RS code.
- **Double Regenerating Codes (DRC) (Figure 10(d)):** We compare RS codes with DRC [19] in a hierarchical network setting. We divide our local cluster into three logical racks. We use the Linux `tc` command to limit the bandwidth between any two storage nodes at different logical racks as 1 Gb/s [44], while the bandwidth between any two storage nodes within the same logical rack remains 10 Gb/s. We compare RS codes and DRC under $(n,k) = (6,4)$ and $(n,k) = (9,6)$. In both cases, we distribute the erasure-coded blocks of each coding group evenly across different nodes in three racks (with $n/3$ erasure-coded blocks each).

Figure 10 shows the results; for our comparisons, Table 2 also shows the theoretical throughput gains of the erasure coding solutions over the conventional repair approach for RS codes. For the 1 Gb/s network, we observe that the empirical throughput gains of the erasure coding solutions are consistent (with only slight degradations) with the theoretical throughput gains. For the 10 Gb/s network, the empirical gains decrease since the coding computation and disk I/O overheads become more significant. For example, MISER codes have less throughput than Butterfly codes in the 10 Gb/s network; the throughput gain of MISER codes drops to 1.25×, while that of Butterfly codes drops to 1.35× (Figure 10(b)). The reason is that both MSR codes retrieve data from $n-1$ non-failed storage nodes for repairs, and MISER codes connect to more storage nodes than Butterfly codes (seven versus five) and incur higher disk I/O overhead. Overall, OpenEC preserves the properties of the erasure coding solutions.

5.4 Improvements with Automated Optimizations

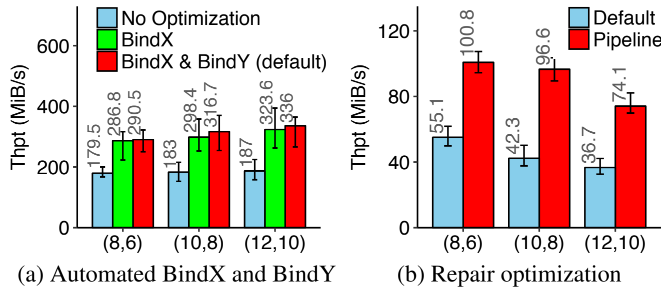
We now evaluate how OpenEC achieves performance gains via automated optimizations (§4.4) for a hierarchical topology. We again configure a three-rack logical topology in our local cluster as in our DRC experiments in §5.3.

We first compare the offline encoding performance for three configurations: (i) automated optimization is disabled, (ii) only automated `BindX` is enabled, and (iii) both automated `BindX` and `BindY` are enabled (our default setting). We consider the $(8,6)$, $(10,8)$, and $(12,10)$ RS codes. We measure the throughput of offline encoding by writing 30 coding groups of blocks into HDFS-3 via OpenEC, which evenly distributes the blocks across three racks. Figure 11(a) shows that enabling only `BindX` increases the throughput by 37-42%, while enabling both `BindX` and `BindY` increases the throughput by 38-44%.

We also evaluate how OpenEC automatically improves the repair performance via the construction of a pipelined ECDAG. We delete all blocks of one storage node and trigger full-node recovery on the same node. Figure 11(b) shows that the repair optimization increases the repair throughput of OpenEC by 82-128%.

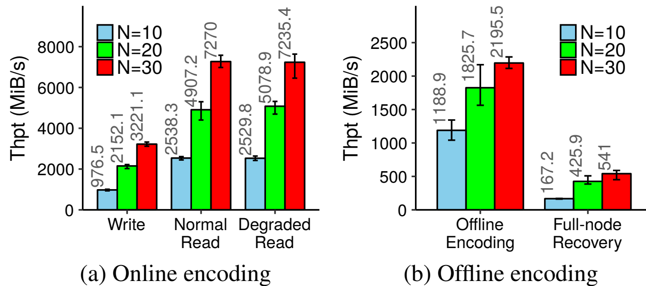
5.5 Performance in Amazon EC2

We finally evaluate OpenEC in Amazon EC2. We configure three settings with N instances, where $N = 10$, 20, and 30 (see §5.1 for the instance type). One instance hosts the OpenEC controller and the HDFS NameNode, and each of the remaining $N-1$ instances hosts an OECClient, an OpenEC agent, an HDFS client, and an HDFS DataNode. We consider the $(9,6)$ RS code, and all $N-1$ clients issue different basic operations as in §5.2. Figure 12 shows the results when OpenEC realizes online and offline encoding atop HDFS-3. We observe consistent throughput patterns as in our local cluster experiments in §5.2 (e.g., both normal reads and degraded reads have similar throughput). Also, the performance of OpenEC scales well with the number of instances.



(a) Automated BindX and BindY (b) Repair optimization

Figure 11: Automated optimization.



(a) Online encoding (b) Offline encoding

Figure 12: Performance in Amazon EC2.

6 Related work

New erasure coding solutions: RS codes [43] are widely deployed today (e.g., [5, 7, 15, 31, 54, 55]), mainly for two reasons. First, RS codes are *maximum distance separable (MDS)*, meaning that under the coding parameters (n,k), the fault tolerance against $n-k$ block failures is achieved with the minimum storage redundancy (i.e., n/k times the original data). Second, RS codes support general coding parameters n and k (provided that $k<n$). However, RS codes have high repair costs, and hence many new erasure coding solutions have been proposed to reduce the repair bandwidth or I/O.

One direction of research is to design new erasure codes. Minimum-storage regenerating (MSR) codes [14] minimize the repair bandwidth and preserve the MDS property. Follow-up studies design new MSR codes [18, 32, 39, 42, 45, 50, 52], some of which are evaluated in open-source DSSs (e.g., PM-RBT codes [39] are evaluated in HDFS, while Butterfly [32] and Clay [52] codes are evaluated in Ceph). Aside MSR codes, some MDS codes incur slightly more repair bandwidth than the minimum point but can be easily constructed with any (n,k) (e.g., [24, 41]), while some non-MDS erasure codes trade more storage redundancy than MDS codes for less repair I/O (e.g., [21–23, 33, 44, 49]). DRC [19] minimizes the cross-rack repair bandwidth in hierarchical topologies.

Another direction of research is to design efficient repair algorithms that apply to general erasure codes. Lazy repair [11, 47] reduces repair executions by deferring a repair until a threshold number of failures occurs. PPR [29] and repair pipelining [25] parallelize a single-failure repair across storage nodes. Proactive degraded reads [20] mitigate tail latencies via the load balancing of read requests.

Unlike the above studies, OpenEC targets a different perspective and focuses on unified and configurable erasure coding management. It supports different new erasure codes and repair algorithms in a unified framework.

Erasure coding programming: Several open-source libraries are available for erasure coding programming. Zfec [10] implements RS codes and is used by Tahoe-LAFS [55]. Jerasure [36] is a C library that supports various erasure codes. It is later extended with GF-Complete [34] to enable fast Galois Field arithmetic. ISA-L [6] is another C library that supports various erasure codes, and it optimizes Galois Field arithmetic for Intel hardware. Both Jerasure and ISA-L libraries are widely used in production (e.g., Ceph and Hadoop 3.0). PyEClib [9] is a Python library used by OpenStack Swift. It builds on liberasurecode [2], which unifies different erasure coding libraries including both Jerasure and ISA-L. OpenEC emphasizes the deployment of erasure codes in DSSs, and it can leverage the above libraries to implement erasure codes via the ECDAG abstraction.

Configurable storage: There is an increasing demand of providing flexibility for storage system management and configuring different storage policies based on application requirements. Existing approaches rely on either client-side customization [12, 13, 28, 37] or the coordination by a centralized controller under the software-defined storage (SDS) framework [16, 48, 51]. OpenEC borrows the same principle from SDS, but specifically focuses on configurable erasure coding management in distributed environments.

7 Conclusions and Future Work

This paper presents OpenEC, a new framework that provides unified and configurable erasure coding management for distributed storage. It leverages the ECDAG abstraction to define erasure codes and configure the workflows of coding operations. Our OpenEC prototype achieves effective performance atop HDFS in both local cluster and Amazon EC2 environments, while supporting a variety of state-of-the-art erasure codes and repair algorithms. Our work sheds light on how to facilitate erasure coding designers to deploy erasure coding solutions in a simple and flexible manner.

This paper currently focuses on HDFS, which organizes data in fixed-size blocks. Our technical report [27] also describes how we integrate OpenEC into QFS [31]. In future work, we study how OpenEC can be deployed in other DSSs, especially object-storage-based DSSs (e.g., Ceph and Swift) that organize data in variable-size objects.

Acknowledgments: We thank our shepherd, Brent Welch, and the anonymous reviewers for their comments. This work was supported by HKRGC (GRF 14216316, CRF C7036-15G) and NSFC (61872414, 61502191). Runhui Li is now with Sangfor Technologies and is the corresponding author.

References

[1] Apache Hadoop 3.0.0. https://hadoop.apache.org/docs/r3.0.0/.

[2] Erasure code API library written in c with pluggable erasure code backends. https://github.com/openstack/liberasurecode.

[3] Facebook's realtime distributed FS based on Apache Hadoop 0.20-append. https://github.com/facebookarchive/hadoop-20.

[4] HDFS erasure coding. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.

[5] HDFS-RAID. https://wiki.apache.org/hadoop/HDFS-RAID.

[6] Intelligent storage acceleration library. https://github.com/01org/isa-l.

[7] OpenStack Swift. https://wiki.openstack.org/wiki/Swift.

[8] Redis. http://redis.io/.

[9] A simple Python interface for implementing erasure codes. https://github.com/openstack/pyeclib.

[10] zfec – an efficient, portable erasure coding tool. https://github.com/tahoe-lafs/zfec.

[11] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. Total recall: System support for automated availability management. In *Proc. of USENIX NSDI*, 2004.

[12] F. Chen, M. P. Mesnier, and S. Hahn. Client-aware cloud storage. In *Proc. of IEEE MSST*, 2014.

[13] J. Y. Chung, C. Joe-Wong, S. Ha, J. W.-K. Hong, and M. Chiang. CYRUS: Towards client-defined cloud storage. In *Proc. of ACM EuroSys*, 2015.

[14] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, 2010.

[15] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, 2010.

[16] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom. Crystal: Software-defined storage for multi-tenant object stores. In *Proc. of USENIX FAST*, 2017.

[17] K. M. Greenan, E. L. Miller, and T. J. E. Schwarz. Optimizing Galois field arithmetic for diverse processor architectures and applications. In *Proc. of IEEE MASCOTS*, 2008.

[18] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang. NCCloud: Applying network coding for the storage repair in a cloud-of-clouds. In *Proc. of USENIX FAST*, 2012.

[19] Y. Hu, X. Li, M. Zhang, P. P. C. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal repair layering for erasure-coded data centers: From theory to practice. *ACM Trans. on Storage*, 13(4):33, 2017.

[20] Y. Hu, Y. Wang, B. Liu, D. Niu, and C. Huang. Latency reduction and load balancing in coded storage systems. In *Proc. of ACM SoCC*, 2017.

[21] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in Windows Azure storage. In *Proc. of USENIX ATC*, 2012.

[22] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proc. of USENIX FAST*, 2012.

[23] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg. On fault tolerance, locality, and optimality in locally repairable codes. In *Proc. of USENIX ATC*, 2018.

[24] K. Kralevska, D. Gligoroski, R. E. Jensen, and H. Øverby. Hashtag erasure codes: From theory to practice. *IEEE Trans. on Big Data*, 2017.

[25] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, 2017.

[26] R. Li, J. Lin, and P. P. C. Lee. Enabling concurrent failure recovery for regenerating-coding-based storage systems: From theory to practice. *IEEE Trans. on Computers*, 64(7):1898–1911, 2015.

[27] X. Li, R. Li, P. P. C. Lee, and Y. Hu. OpenEC: Toward unified and configurable erasure coding management in distributed storage systems. Technical report, CUHK, 2019. http://www.cse.cuhk.edu.hk/~pclee/www/pubs/tech_openec.pdf.

[28] M. Mesnier, F. Chen, T. Luo, and J. B. Akers. Differentiated storage services. In *Proc. of ACM SOSP*, 2011.

[29] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage. In *Proc. of ACM EuroSys*, 2016.

[30] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm blob storage system. In *Proc. of USENIX OSDI*, 2014.

[31] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast file system. In *Proc. of VLDB Endowment*, 2013.

[32] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Guyot, E. E. Gad, and Z. Bandic. Opening the chrysalis: On the real repair performance of MSR codes. In *Proc. of USENIX FAST*, 2016.

[33] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, C. Huang, and J. Li. Simple regenerating codes: Network coding for cloud storage. In *Proc. of IEEE INFOCOM*, 2012.

[34] J. S. Plank, K. Greenan, E. Miller, and W. Houston. GF-Complete: A comprehensive open source library for galois field arithmetic. Technical Report UT-CS-13-703, University of Tennessee, 2013.

[35] J. S. Plank and C. Huang. Tutorial: Erasure coding for storage applications. Slides presented at USENIX FAST 2013, Feb 2013.

[36] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

[37] R. Pontes, D. Burihabwa, F. Maia, J. Paulo, V. Schiavoni, P. Felber, H. Mercier, and R. Oliveira. SafeFS: A modular architecture for secure user-space file systems (one FUSE to rule them all). In *Proc. of ACM SYSTOR*, 2017.

[38] N. Prakash, V. Abdrashitov, and M. Médard. The storage versus repair-bandwidth trade-off for clustered storage systems. *IEEE Trans.on Information Theory*, 64(8):5783–5805, Aug 2018.

[39] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *Proc. of USENIX FAST*, 2015.

[40] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proc. of USENIX HotStorage*, 2013.

[41] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proc. of ACM SIGCOMM*, 2014.

[42] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction. *IEEE Trans. on Information Theory*, 57(8):5227–5239, 2011.

[43] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[44] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proc. of VLDB Endowment*, 2013.

[45] N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchandran. Interference alignment in regenerating codes for distributed storage: Necessity and code constructions. *IEEE Trans. on Information Theory*, 58(4):2134–2158, 2012.

[46] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proc. of IEEE MSST*, 2010.

[47] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proc. of ACM SYSTOR*, 2014.

[48] I. A. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska. sRoute: Treating the storage stack like a network. In *Proc. of USENIX FAST*, 2016.

[49] I. Tamo and A. Barg. A family of optimal locally recoverable codes. *IEEE Trans. on Information Theory*, 60(8):4661–4676, 2014.

[50] I. Tamo, Z. Wang, and J. Bruck. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Trans. on Information Theory*, 59(3):1597–1616, 2013.

[51] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *Proc. of ACM SOSP*, 2013.

[52] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, et al. Clay codes: Moulding MDS codes to yield an MSR code. In *Proc. of USENIX FAST*, 2018.

[53] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, 2002.

[54] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of USENIX OSDI*, 2006.

[55] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proc. of ACM StorageSS*, 2008.

Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity

Saurabh Kadekodi, K. V. Rashmi, Gregory R. Ganger
Carnegie Mellon University

Abstract

Large-scale cluster storage systems typically consist of a heterogeneous mix of storage devices with significantly varying failure rates. Despite such differences among devices, redundancy settings are generally configured in a one-scheme-for-all fashion. In this paper, we make a case for exploiting reliability heterogeneity to tailor redundancy settings to different device groups. We present HeART, an online tuning tool that guides selection of, and transitions between redundancy settings for long-term data reliability, based on observed reliability properties of each disk group. By processing disk failure data over time, HeART identifies the boundaries and steady-state failure rate for each deployed disk group (e.g., by make/model). Using this information, HeART suggests the most space-efficient redundancy option allowed that will achieve the specified target data reliability. Analysis of longitudinal failure data for a large production storage cluster shows the robustness of HeART's failure-rate determination algorithms. The same analysis shows that a storage system guided by HeART could provide target data reliability levels with fewer disks than one-scheme-for-all approaches: 11–16% fewer compared to erasure codes like 10-of-14 or 6-of-9 and 33% fewer compared to 3-way replication.

1 Introduction

Large cluster storage systems almost always include a heterogeneous mix of storage devices, even when using devices that are all of the same type (e.g., Flash SSDs or mechanical HDDs). Commonly, this heterogeneity arises from incremental deployment combined with per-acquisition optimization of which make/model to acquire, such as targeting the lowest cost-per-byte option available at the time. As a result, a given cluster storage system can easily include several makes/models, each in substantial quantity.

Beyond performance and capacity differences, different makes/models can also have substantially different reliabilities. For example, Fig. 1 shows the average *annualized failure rates (AFRs)* during the useful life (stable operation period) for the 6 HDD make/model-based disk groups that make up more than 90% of the cluster storage system (with 100,000+ disks) used for the Backblaze backup service [5]. The highest failure rate is over 3.5× greater than the lowest, and no two are the same. Schroeder et al. [32] recently showed that different Flash SSD makes/models similarly exhibit substantial failure rate differences.

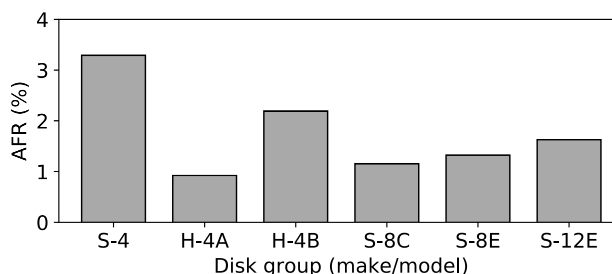


Figure 1: Annualized failure rate (AFR) for the six disk groups that make up >90% of the 100,000+ HDDs used for the Backblaze backup service [5]. Details of each disk group are given in Section 2.

Despite such differences, the degree of redundancy employed in cluster storage systems for the purpose of long term data reliability (e.g., the degree of replication or erasure code parameters) is generally configured as if all of the devices have the same reliability. Unfortunately, this approach leads to configurations that are overly resource-consuming, overly risky, or a mix of the two. For example, if the redundancy settings are configured to achieve a given data reliability target (e.g., a specific *mean time to data loss ($MTTDL$)*) based on the highest AFR of any device make/model (e.g., S-4 from Fig. 1), then too much space will be used for redundancy associated with data that is stored fully on lower AFR makes/models (e.g., H-4A). Continuing this example, our evaluations show that the overall wasted capacity can be up to 16% compared to uniform use of erasure code settings stated as being used in real large-scale storage clusters [13, 25, 26, 28] and up to 33% compared to using 3-replication for all data—the direct consequence is increased cost, as more disks are needed. If redundancy settings for

all data are based on lower AFRs, on the other hand, then data stored fully on higher-AFR devices is not sufficiently protected to achieve the data reliability target.

This paper presents HeART (Heterogeneity-Aware Redundancy Tuner), an online tool for guiding exploitation of reliability heterogeneity among disks to reduce the space overhead (and hence the cost) of data reliability. HeART uses failure data observed over time to empirically quantify each disk group's reliability characteristics and determine minimum-capacity redundancy settings that achieve specified target data reliability levels. For the Backblaze dataset of 100,000+ HDDs over 5 years, our analysis shows that using HeART's settings could achieve data reliability targets with 11–33% fewer HDDs, depending on the baseline one-scheme-for-all settings. Even when the baseline scheme is a 10-of-14 erasure code whose space-overhead is already low, HeART further reduces disk space used by up to 14%.

Online (real-time) use of observed device reliability requires careful design. HeART uses robust statistical approaches to identify not only a steady-state AFR estimate for each disk group, but also the transitions between deployment stages: infancy→useful life→wearout, as in bathtub curve visualizations. HeART assumes that administrators have a baseline redundancy configuration that would be used in HeART's absence; that same configuration should be used for a disk group, when it is initially deployed. HeART processes failure data for that disk group, during this initial period of 3–5 months, to determine both when infancy ends and a conservative AFR estimate for the useful life period. It also suggests the most space-efficient redundancy settings supported by the storage system that will achieve the specified data reliability target.

Naturally, the useful life period does not last forever. HeART continues to process failure data for each disk group, automatically identifying the onset of the wearout period. At this point, a transition to more conservative redundancy (e.g., the original baseline), and possibly decommisioning, is warranted. Importantly, HeART distinguishes between anomalous failure occurrences (e.g., one-time device-independent events, like a power surge, in which many devices fail together) and true changes in the underlying AFR.

This paper makes four primary contributions. First, it highlights an often overlooked aspect of device heterogeneity (reliability) that should be exploited in cluster storage systems, and quantifies potential cost–and/or–reliability benefits. Second, it confirms the above observation and quantification with analysis of multi-year reliability data from a sizable cluster storage deployment (Backblaze), showing up to 11–33% reduction in the overall number of disks needed to achieve target data reliability. Third, it describes an online tool (HeART) that automatically determines per-disk-group useful life AFRs and durations, and identifies the right redundancy scheme settings for each. Fourth, it shows that HeART's algorithms are effective using data from a large-scale production cluster (Backblaze) and are able to expose the expected capacity savings opportunities without compromising data reliability.

Make/Model	**Disk group shorthand**	**# of disks**	**Oldest disk age**
Seagate ST4000DM000	S-4	37015	5 yrs
HGST HMS5C4040ALE640	H-4A	8715	4.77 yrs
HGST HMS5C4040BLE640	H-4B	15048	4.2 yrs
Seagate ST8000DM002	S-8C	9885	1.99 yrs
Seagate ST8000NM0055	S-8E	14395	1.2 yrs
Seagate ST12000NM0007	S-12E	21581	8 mts

Table 1: The disk groups identified from the Backblaze dataset for reliability heterogeneity analysis. The disk group shorthand above is used to represent the respective makes/models throughout the paper.

2 Having HeART can make you rich

This section builds a case for HeART by showing the benefits of using different redundancy schemes for disk groups exhibiting different reliability characteristics in the same commercially used cluster storage system. To support the case, we quantify space overhead reductions that can be achieved by adopting the different redundancy schemes.

2.1 The Backblaze dataset

Our analysis is based on an open source dataset from a data backup organization, Backblaze [5]. This dataset consists of over 5 years of disk reliability statistics from a production cluster storage system with over 100,000 HDDs.

We use the standard metric, *annualized failure rate (AFR)*, to describe a disk's fail-stop rate[1] [9, 33]. As the name suggests, it is the expected percentage of disks that will fail-stop in a given year from a population of disks. AFR is calculated on day d, based on the past d days of reliability data, using the following formula:

$$AFR\ (\%) = \frac{f_d}{n_1 + n_2 + \ldots + n_d} \times 365 \times 100 \qquad (1)$$

where f_d is the number of disks failed in the past d days and n_i is the number of disks operational during day i.

Note that the AFR calculation is dependent on the number of days a disk was in operation. This can be tricky to estimate from the Backblaze dataset since the "death" of a disk in this dataset may also indicate its decommissioning, which may or may not imply its failure. We argue that, in the case of Backblaze, the date of decommissioning a disk only affects the absolute date at which it would have fail-stopped, but does not affect its *rate of failure*. Backblaze adopts a proactive disk replacement strategy that is driven by monitoring a combination of five S.M.A.R.T. (Self-Monitoring, Anal-

[1]Storage devices can exhibit *partial failures* and *fail-stops (complete failures)*. Partial failures might involve a particular read or write failing because of a sector error or checksum failure, while the disk as a whole is still functional. In the case of fail-stop, the disk stops functioning altogether.

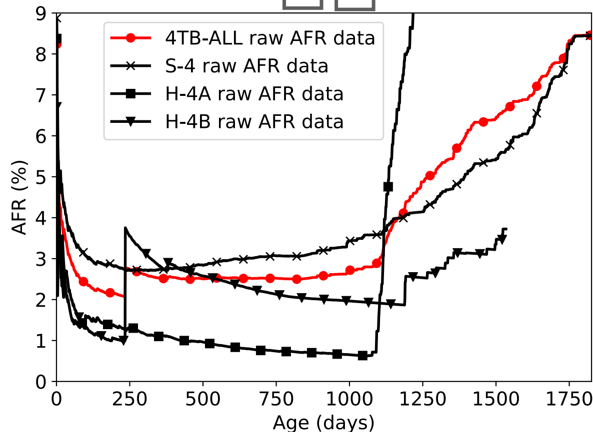


Figure 2: AFR comparison between all 4TB disks grouped together and disk groups broken down by make/model. The AFR differences in make/model-based grouping enables HeART to perform finer-grained specialization leading to higher benefits.

ysis and Reporting Technology) statistics.[2] The increased probability of failure indicated by grown defects in a disk is supported by several previous studies [7, 20, 24, 29]. In fact, Pinheiro et al. [24] show that the critical threshold for several S.M.A.R.T. attributes before their imminent failure is one—that is, the probability of failure of a disk in the next two months increases manifold when any of these S.M.A.R.T. attributes show a value greater than zero. Ma et al. [20] also show the high likelihood of disk failure by monitoring the reallocated sectors count (S.M.A.R.T. attribute 5), which is one of the signals used by Backblaze as a disk replacement indicator. Therefore, we believe that Backblaze's proactive disk replacement rate is a reasonable approximation for the actual disk failure rate.

2.2 Disk group formation and varying AFRs

To effectively exploit heterogeneity in AFRs of different disk groups, we need to categorize the disks using some parameter that (1) groups disks with similar AFRs together and (2) has substantially different AFRs across groups. In whichever manner we choose to group the disks, in order to gain statistical confidence in the AFR value, we need to ensure that each disk group has a sizeable population. Our definition of a sizeable population is approximately 10,000 or more disks. This is in line with disk populations considered in previous reliability studies [21]. We identify the following four ways to categorize disks:

- **By make/model:** Economies of scale result in large quantities of disks being purchased from the same vendor. Prior studies have shown that AFR may vary significantly by vintage [10, 20, 24].
- **By capacity:** Grown defects can be a function of disk capacity, thus causing disks of similar capacity to fail at a similar rate.
- **By operational conditions:** Disks that share similar vibration or temperature experiences may cause them to fail similarly. Thus, chassis placement and other operational conditions may influence failure rates.
- **By usage:** Increased space utilization or higher I/O rates may result in different disks showing different failure characteristics.

[2] Backblaze uses S.M.A.R.T. 5 (Reallocated Sectors Count), S.M.A.R.T. 187 (Reported Uncorrectable Errors), S.M.A.R.T. 188 (Command Timeout), S.M.A.R.T. 197 (Current Pending Sector Count) and S.M.A.R.T. 198 (Uncorrectable Sector Count) as indicators that a disk is about to fail. [6]

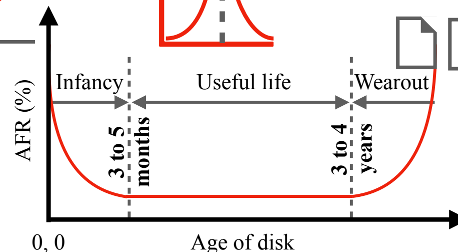


Figure 3: The canonical bathtub curve used to represent disk failure characteristics.

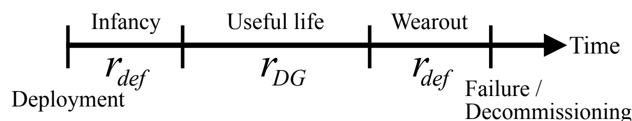


Figure 4: An abstract timeline of a disk group from deployment to failure or decommissioning, with the three distinct periods. The notations below the timeline (r_{def} and r_{DG}) denote the redundancy scheme employed during the respective stage.

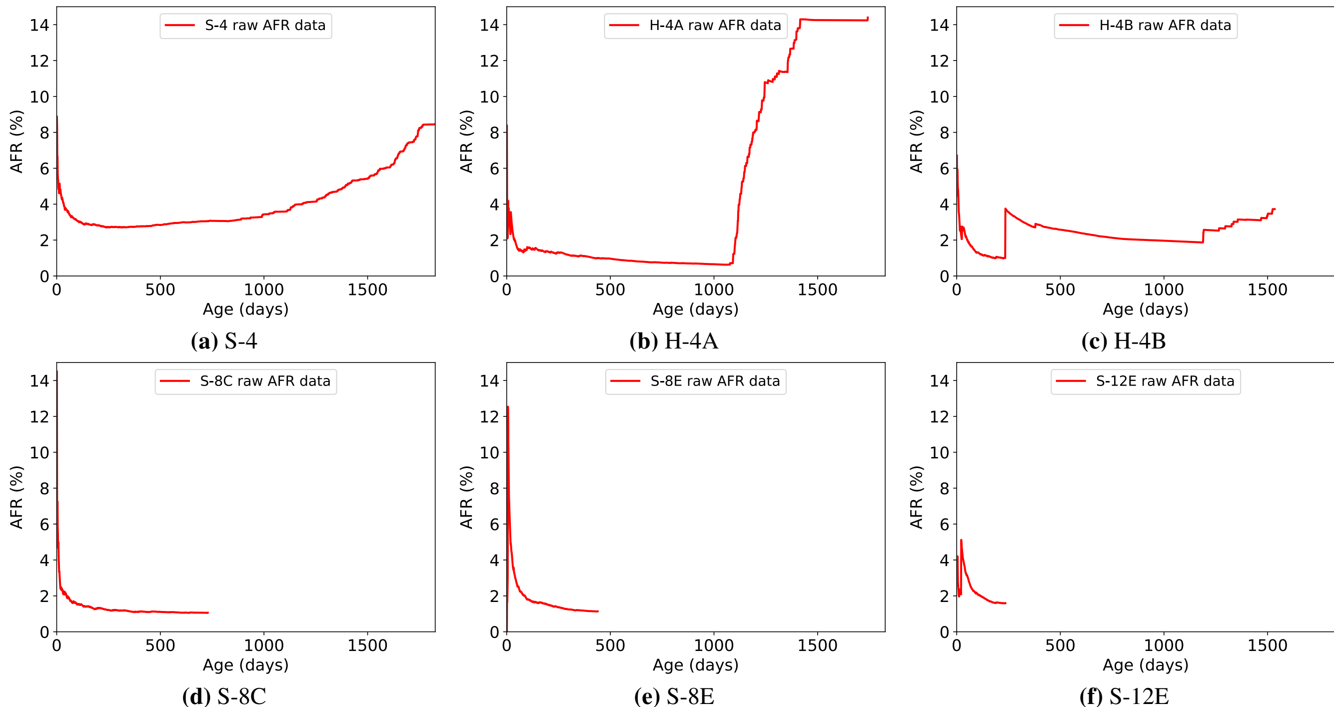
Unfortunately since we do not have access to the operational conditions or usage patterns, we can only analyze grouping on the basis of make/model or capacity.

Fig. 2 shows the AFR by considering all 4TB disks as one disk group (red curve with circular marks) and the AFRs of the three make/models of 4TB disks as individual disk groups (black curves). We see significant differences between AFRs when disks are grouped by make/model, suggesting that grouping by capacity is insufficient. HeART groups disks by make/model.

Table 1 shows the six make/model disk groups that make up over 90% of the Backblaze deployment, with their population size, the age of their oldest disk, and the shorthand names we will use throughout the paper.

AFR **variation over time.** As expected, AFR values of each disk group vary over the lifetime of disks. It is well known that the AFR values over a disk's lifetime follow a *bathtub curve* [11, 12, 45]. Fig. 3 shows the canonical representation of a bathtub curve. The lifetime is typically divided into three distinct periods:

- **Infant mortality:** A higher failure rate in the early days after deployment. This is also called the *burn-in period*.
- **Useful life period:** The stable region of operation, where rate of failure is lower.
- **Wearout stage:** A higher failure rate towards the disks' end of life due to wear and aging.



(a) S-4 **(b)** H-4A **(c)** H-4B

(d) S-8C **(e)** S-8E **(f)** S-12E

Figure 5: Cumulative raw AFR versus age (in days) for all six disk groups being analyzed.

Fig. 5 shows the AFR behavior versus age of the six disk makes/models. The three disks in the top row clearly exhibit all three stages of the bathtub curve.[3] This is because the oldest disks from the S-4, H-4A and H-4B disk groups in the dataset are old enough to have entered their wearout stages. Since the deployment of S-8C, S-8E and S-12E disks has been more recent, these disk groups have ended their infant mortality, but are yet to enter their wearout stages.

2.3 Space savings from heterogeneous AFRs

Our goal is to reduce storage overhead by tailoring the redundancy scheme employed to the failure rate of a disk group during its useful life period. We parameterize a redundancy scheme using two parameters n (called "length") and k (called "dimension"), and call it a (n,k) scheme.[4] For any replication based scheme, $k = 1$ and n represents the total number of replicas. For any erasure coding based scheme, k represents the number of data chunks and $(n-k)$ is the number of parity chunks, thus resulting in n chunks in total.[5] For an (n,k) redundancy scheme, the storage overhead is given by $\frac{n}{k}$.

HeART achieves reduction in storage overhead by explicitly factoring in the group-specific AFR values in deciding the appropriate redundancy scheme for each disk group. Based on the canonical bathtub curve (Fig. 3), and the AFR curves shown in Fig. 5, we conclude that the safest stage to apply lower redundancy (without risking not meeting the reliability target) during a disk group's lifetime is in its useful life period. Fig. 4 shows the abstract timeline of a disk group, where r denotes the redundancy scheme applied in each stage. Since all cluster storage systems today use some redundancy scheme whose resilience is acceptable to them, we assume that to be the *default* redundancy scheme. Since infancy and wearout periods have higher and less stable AFRs compared to useful life, for every disk group, HeART employs the default redundancy scheme for all infancy and wearout periods. This is shown as r_{def} in Fig. 4.

HeART suggests lower redundancy than the default scheme only during the useful life period, during which AFR values are relatively stable. Data redistribution and issues related to data placement and scheme transitions are discussed in Section 5.

We use the standard metric for reliability of data employed in storage systems, *mean time to data loss* ($MTTDL$). $MTTDL$ is calculated based on two rates – *mean time to failure* ($MTTF$) and *mean time to repair* ($MTTR$) [23, 38] $MTTF$ is directly related to the disk's AFR. $MTTR$ is the time it takes to reconstruct the lost data on the failed disk. Following prior work, we model the time to repair based on the time it takes to detect that a disk has failed (which is approximately 15 minutes) [13, 17]. We note that, by choosing the failure detection time as a proxy for the repair time, we are effectively choosing a lower bound on

[3] Fig. 5c corresponding to disk H-4B does not completely conform to the bathtub shape. We will discuss this case later in detail.

[4] This notation follows the standard notation employed in the coding theory literature.

[5] Although the description of the notation applies only to "systematic" codes, and most of the codes employed in storage systems are indeed systematic, HeART is applicable to storage systems employing non-systematic codes as well.

the repair time. Reliability differences between redundancy schemes are higher when repair times are higher, leading to even greater potential for space saving through HeART.

When a disk group enters its useful life period, HeART chooses a redundancy scheme (r_{DG}) that meets the following conditions:

1. is as reliable as r_{def}, i.e. $MTTDL^{r_{DG}} \geq MTTDL^{r_{def}}$
2. tolerates at least as many failures as r_{def}

According to condition 1 above, we need to set a target $MTTDL$ in order to compare the resilience of different redundancy schemes. Although prior studies have shown $MTTDL$ targets to be as low as 10,000 years [27], in order to ensure that we do not regress on reliability that disks in our dataset can currently offer, we set the target $MTTDL$ to be the $MTTDL$ of the default redundancy scheme applied on the disk group with the highest AFR. S-4 is the disk group with highest useful life AFR in the Backblaze dataset (refer Fig. 1). Therefore, for every default redundancy scheme, we will use S-4's $MTTDL$ for that scheme as the target $MTTDL$.

Multiple redundancy schemes can achieve the same or similar $MTTDL$ values. These schemes can differ in their dimension (k) or the number of parity chunks per stripe ($n-k$) or both. It is well known that, generally speaking, codes with a longer dimension can provide the same $MTTDL$ with lower space overhead compared to shorter codes. However, long codes consume significantly higher cluster bandwidth for reconstruction, since many more disks have to be accessed when performing reconstruction of failed data [17, 25, 26, 28]. The cluster bandwidth consumed during reconstruction is a major concern in erasure-coded storage systems. This has been highlighted in several works in the past [17, 25, 26, 28] and is consistent with our discussions with cluster storage system administrators. We, therefore, limit our cost reduction analysis to codes with at most 2× the dimension (i.e., parameter k) of the default redundancy scheme.

Table 2 shows space savings for one disk group (H-4A) as an example. We will first highlight the space reduction when erasure coding schemes are used as the default, focusing on the $(14,10)$ and $(9,6)$ schemes known to have been used in large data centers [13, 25, 26, 28]. For $(14,10)$ as the default scheme, the $MTTDL$ difference between H-4A and S-4 disks is over 580×. Thus, we can choose a weaker redundancy scheme (a scheme with lower storage overhead $\frac{n}{k}$), so long as conditions 1 and 2 above are fulfilled. In fact, the high AFR differences allow us to use the longest allowed optimized code (2× the dimension of the default redundancy scheme) for H-4A disks, i.e. $(24,20)$ leading to a useful life space reduction of 14%. Similarly, when using $(9,6)$ as the default scheme, the $MTTDL$ difference between H-4A and S-4 is over 160×. This again allows us to choose the longest code for H-4A when $r_{def} = (9,6)$, i.e. $(15,12)$, providing a space reduction of 16%.

Disk groups		r_{def} **= (14, 10)**		r_{def} **= (9, 6)**		r_{def} **= (3, 1)**	
DG	AFR	r_{DG}	**Cost↓**	r_{DG}	**Cost↓**	r_{DG}	**Cost↓**
S-4	3.29%	(14, 10)	**NA**	(9, 6)	**NA**	(3, 1)	**NA**
H-4A	0.92%	(24, 20)	**14%**	(15, 12)	**16%**	(4, 2)	**33%**

Table 2: A sample of the estimated savings achievable through HeART. The space reductions obtained on H-4A disks by using redundancy schemes with lower storage overhead while meeting the reliability target set by applying the default redundancy scheme (r_{def}) on S-4 disks.

For r_{def} = 3-replication (recall that, under the (n, k) notation introduced above, 3-replication is denoted as the $(3,1)$ erasure code), we can tune the redundancy on H-4A disks to $(4,2)$ to respect our 2× default stripe dimension limit and still achieve an $MTTDL$ that is approximately 11× that of S-4's $MTTDL$. Using a $(4,2)$ scheme leads to a 33% reduction in disk space.

Large internet services companies try very hard to minimize free space (as low as 5%, according to some administrators) in order to minimize capital and operating costs. We are told that space savings translate directly into reduced numbers of disks needed, and even modest space savings (e.g., 10%) would build a solid case for tailoring redundancy schemes to heterogeneous disk AFRs.

We note that much of the reduction in storage overhead arises from allowing codes up to 2× in dimension (i.e., parameter k). However, simply employing an erasure code with twice the dimension for all data is not generally a suitable solution. First, the AFR for certain disk groups might be high enough to make codes with 2× dimension not acceptable causing them to miss the target reliability. Second, and more broadly, the reconstruction overheads can be unacceptable. For popular codes employed in practice, the amount of cluster bandwidth required for reconstruction is proportional to $k \times AFR$, where k is the dimension of the code. The stable and lower AFR during a disk group's useful life period allows the I/O generated for reconstruction to be contained even if longer codes are employed, which is why HeART optimizes redundancy schemes *only* during a disk group's useful life. Using longer codes on data stored on disk groups in their infancy and wearout stages would exacerbate the cluster bandwidth consumption for reconstruction due to higher failure rates in these stages.

3 The ways of the HeART

This section describes the challenges, design and implementation of HeART. We also quantify the cost reductions achieved by HeART for the Backblaze dataset.

3.1 Challenges

There are several challenges in taking the idea presented in Section 2 to practice.

Challenge 1: Function online and be quick. In making our case for HeART, we made use of the complete fail-

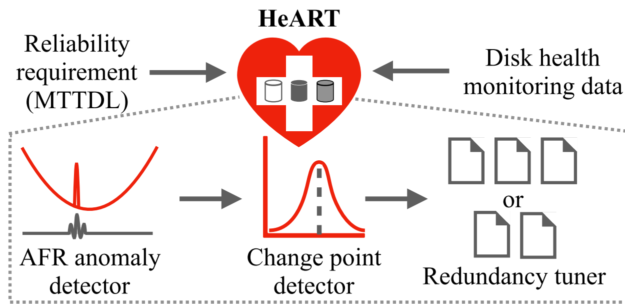


Figure 6: Schematic diagram of HeART. Components include an anomaly detector, an online change point detector, and a redundancy tuner.

ure information (e.g., the full bathtub curve) for the disk groups. This helped in clearly identifying the 3 stages of a disk group's lifetime and AFR values in each of the stages. In practice, however, AFR values for disk groups deployed in cluster storage systems can only be known in an online fashion (i.e., as a continuous stream of reliability data, as it is observed). Furthermore, the crux of the cost reduction from HeART comes from quickly tuning the redundancy scheme as soon as we are confident of a disk group having entered its useful life period. Thus, our first challenge in building HeART is that it needs to function in an online fashion taking a continuous stream of disk health data as input and quickly react to the changes in the failure rate.

Challenge 2: Be accurate. It is important to correctly identify the three different stages of the bathtub curve for each disk group (recall Fig. 3). If we are hasty in declaring the end of the infancy period or lax in identifying end of useful life, we might not meet the reliability target because of having tailored the redundancy to a relatively low failure rate during the useful life period. In contrast, if we are too lax about declaring end of infancy or too hasty in declaring onset of the wearout stage, the opportunity of cost reduction will diminish.

Challenge 3: Filter-out anomalies. Events such as power outages, natural disasters or human error can cause large numbers of disks to fail at once It is important to distinguish between an accidental rise in AFR due to such anomalous events versus the rise in AFR due to onset of the wearout stage. Our third challenge is to perform AFR anomaly detection to avoid prematurely declaring end of useful life, consequently reducing the window of opportunity for cost reduction. At the same time, HeART needs to exercise caution so as to not treat a genuine rise in AFR as an anomaly, which risks not meeting reliability targets.

3.2 HeART architecture

Fig. 6 shows the primary components of HeART. HeART assumes the existence of a disk health monitoring/logging mechanism already in place, which is common in large-scale cluster storage deployments. From the time of deployment till the end of infancy, the default redundancy scheme (r_{def}) is used to protect the data stored on a disk group. Periodically, disk health data for each disk group is passed through an *anomaly detector*. Following an anomaly check, the cumulative AFR of every disk group is passed through a *change point detector*, which checks if a transition to different phase of life has occurred. Once the change point detector announces start of the useful life period, HeART suggests a new redundancy mechanism for the useful life of the disk group (r_{DG}). It computes a *determined useful life AFR* (AFR_{DG}), which is the AFR at the end of infancy padded with a tunable buffer, and uses it to calculate $MTTDL^{r_{DG}}$ for different redundancy scheme (r_{DG}) options. The buffer is introduced to tolerate the fluctuation of AFR during the useful life period (see Section 4.3). HeART keeps checking for anomalies and change points throughout the useful life period. When the change point detector marks the end of useful life, HeART raises an alert to reset the redundancy scheme to r_{def} to handle the increased AFR during wearout, as was handled in the absence of HeART.[6]

The remainder of this section describes our approach to addressing the above mentioned challenges. We leverage established tools and algorithms from online services and time-series analysis literature. While other options may perform even better, our evaluations indicate that these established tools are effective. We show the efficacy of HeART using the Backblaze dataset in Section 4.

3.3 Online anomaly detection

Incidents like losing power to a rack of disks, a natural disaster, or an accident, can cause a large number of failures resulting in a sudden rise in AFR. Such bulk failures can easily exceed the limits of any reasonable redundancy scheme, so administrators seek to mitigate them by defining appropriate failure domains and spreading data+redundancy across the failure domains [25, 26]. Such failures are not reflective of the true rise in AFR because of wearout, and therefore HeART considers these incidents as anomalies. It is important to note that the benefits we extract from exploiting the reliability heterogeneity are proportional to the length of the useful life period, and therefore prematurely announcing wearout stage due to an anomaly would significantly diminish achievable gains.

We use the H-4B disks as a motivating example for anomaly detection (shown in Fig. 7). The raw AFR curve (red curve) shows that just after a few days into its useful life, there are large spikes in the AFR curve for drives that are about 235 days old (point A) and 380 days old (point B). Further along, we observe three more spikes that are in succession for disks that are about 1200 days old (points C, D and E). The failures corresponding to points A and B are all caused because of 322 drives failing on one particular date.

[6] We note that the current architecture of HeART determines one useful life AFR for all disks belonging to a disk group and does not handle changes in the intra-disk-group reliability distribution over time.

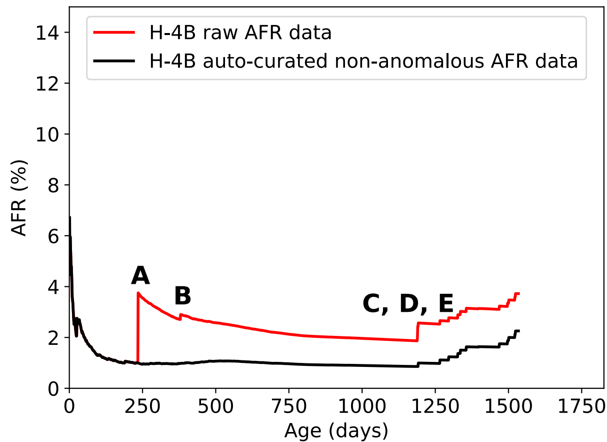


Figure 7: Raw and HeART-curated AFR curves for the H-4B disk group. Five spikes in AFR (points A–E), which correspond to four (anomalous) bulk failure events, are automatically filtered out by HeART.

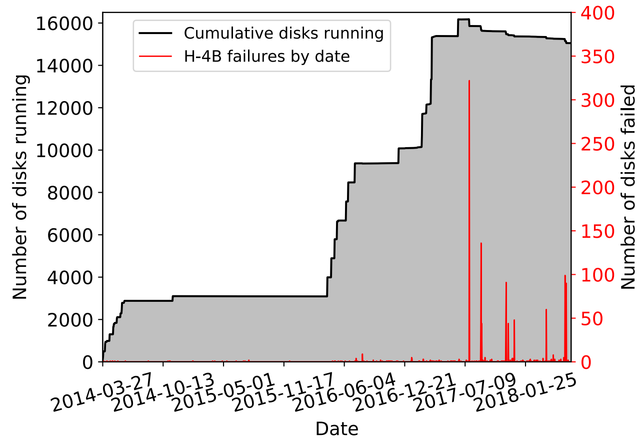


Figure 8: Total number of disks and number of disk failures by date for H-4B disks. The step-wise jumps in the black curve represent incremental deployments. The largest red spike represents the disks that failed on July 23, 2017, causing anomalies A and B in Fig. 7.

Here, failure of disks of two different ages correspond to a failure event on the same day because these disks were deployed on different dates. Fig. 8 shows the total number of disks running and the per-day number of disk failures of H-4B as a function of the date. The left y-axis shows the cumulative disks of H-4B running on each day. The steps in the black curve show the incremental deployments of H-4B disks. The right y-axis shows the number of H-4B disks failing on each day (red curve). The tallest red spike in Fig. 8 corresponds to points **A** and **B** from Fig. 7. Points C, D and E occurred because of disks failing on different days.

In the absence of anomaly detection, HeART would have incorrectly concluded that the disk group's wearout stage began as early as point A.

3.4 Online change point detection

We refer to a transition in the AFR curve of a disk group as a *change point*. There are two major change points for each disk group: end of infant mortality stage and the onset of the wearout stage. This subsection describes our methods of identifying the two change points.

Onset of useful life period. HeART uses prior studies about infant mortality in HDDs along with change point detection to decide a disk group's end of infancy. Prior studies performed on the Google and EMC disk fleets [20, 24] have shown that infant mortality lasts for approximately one quarter. Therefore, in order to be conservative, HeART exempts the first quarter from being assessed for end of infant mortality. Since disk reliability data is collected periodically, each time data is collected after the first 90 days, we run change point detection on the AFR curve generated by a sliding window of the past 30 days. HeART declares end of infancy if the last change point marked by the detector is over 30 days old, and the failure rate during the last 30 days is relatively constant. More precisely, HeART declares end of infancy when the difference between the observed maximum and minimum AFR values in at least 30 days past the last change point is less than a certain threshold T_{flat}. T_{flat} is the threshold for *flatness* and is a tunable parameter in HeART. Sensitivity to T_{flat} is evaluated in Section 4.3. Note that HeART takes a conservative approach in declaring the onset of the useful life period of a disk group in order to increase confidence about reducing redundancy for data stored on that disk group.

End of useful life period. Being lax in declaring the end of useful life period (i.e., onset of wearout) can risk in HeART not meeting the intended reliability target. Hence, HeART takes a conservative approach and marks the end of useful life for the first AFR observed that is greater than the determined useful life AFR. Since HeART checks for anomalous AFR fluctuations before checking for change points, if the anomaly detection phase does not filter out an increase in AFR, HeART assumes it to be a true increase in AFR. Thus, here too HeART takes a conservative approach and errs on the side of exiting the useful life period early and reverting to the default redundancy scheme.[7]

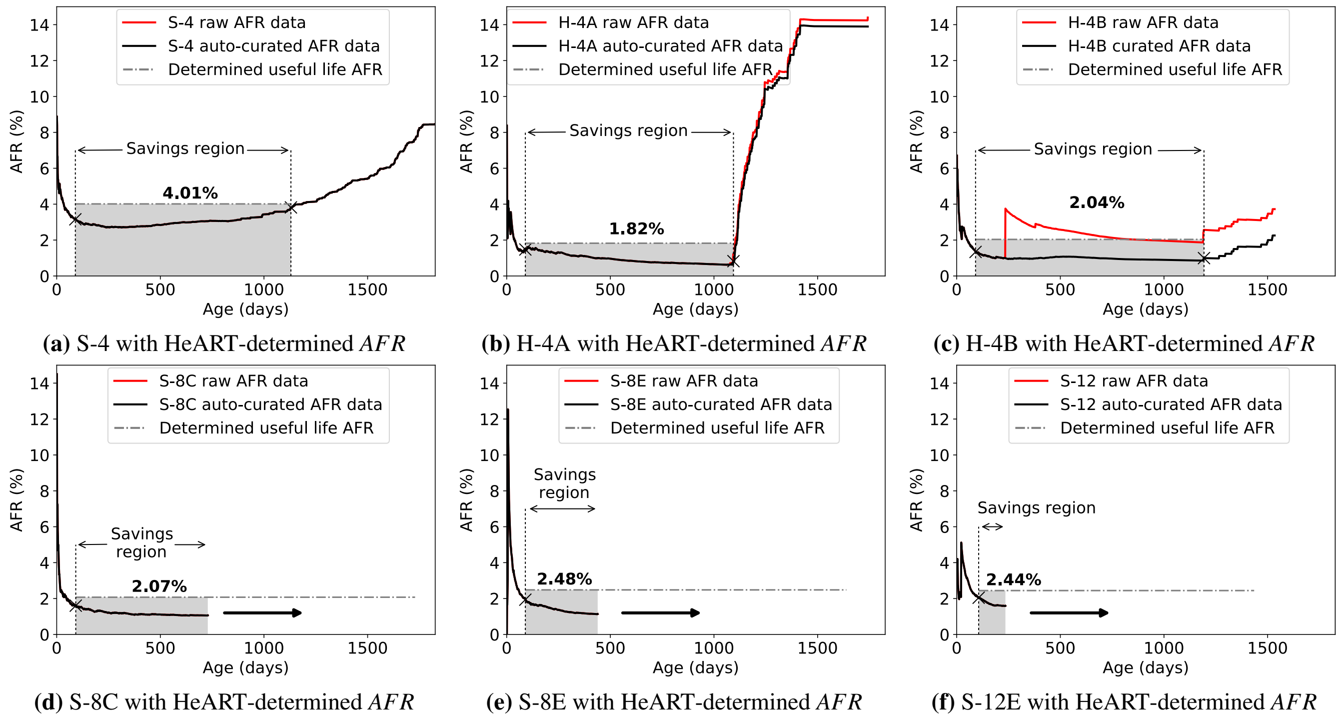
4 Measuring HeART

This section describes implementation details of various components that make up HeART and presents an evaluation of HeART on the Backblaze dataset.

4.1 Implementation of the components

Our current implementation of HeART leverages existing, standard algorithms for anomaly detection and change point

[7]Although the H-4A graph in Fig. 5b appears to show a sudden, huge rise in AFR, we believe that it is an artifact of Backblaze's recording of decommissioned disks as failed, based on the device removal pattern seen in the failure data. Data from more sources are needed to confirm this hypothesis. If some disks do exhibit such transitions, then strategies for predicting failures (and wearout onset), such as by using S.M.A.R.T. statistics [4, 21, 44, 48], will be needed to use any but the most conservative redundancy schemes.



(a) S-4 with HeART-determined AFR **(b)** H-4A with HeART-determined AFR **(c)** H-4B with HeART-determined AFR

(d) S-8C with HeART-determined AFR **(e)** S-8E with HeART-determined AFR **(f)** S-12E with HeART-determined AFR

Figure 9: HeART in action on all disk groups, showing successful identification of infant mortality, useful life and wearout periods as well as automatic removal of anomalies.

detection. Employing more sophisticated algorithms might lead to even better results.

Anomaly detector: For anomaly detection, our current implementation of HeART uses the RRCF algorithm [3] exposed by Amazon's data analytics service offering called Kinesis [2].[8] The anomaly detector acts on a reliability data stream made available by the disk health monitoring system. The output from the anomaly detector is also a data stream containing anomaly scores produced by the RRCF algorithm. Potential anomalies identified by RRCF have a higher anomaly score than data that the algorithm considers non-anomalous. RRCF generates the anomaly score based on how different the new data is compared to the recent past. For consistency with change point detection, we set the window size of the recent past to be one month. If the anomaly score is above a certain threshold, HeART considers that snapshot of reliability data as anomalous. RRCF advises to only consider the highest anomaly scores as true anomalies [3]. The anomaly score threshold is a tunable parameter in HeART. Lowering the score makes HeART more sensitive to fluctuations in AFRs.

Change point detector: Our current implementation of HeART uses a standard window-based change point detection algorithm, which compares the discrepancy between adjacent *sliding windows* within the AFR curve to determine if a change point has been encountered. In particular, we employ the *Ruptures* library for online change point detection [39, 40]. We set the sliding window size to one month, because AFRs at a lower granularity than a month are jittery.

4.2 Evaluation on the Backblaze dataset

Identifying useful life period. Fig. 9 shows the results from HeART running on all 6 disk groups of the Backblaze dataset. HeART accurately identifies the infancy, useful life and wearout stages of the S-4, H-4A and H-4B disk groups shown in Figs. 9a, 9b and 9c, respectively. For the S-8C, S-8E and S-12E disk groups (Figs. 9d, 9e and 9f), HeART identifies the end of infancy and correctly shows that they are still in their useful life. The width of the shaded region of each disk group highlights the "savings region", i.e. the useful life period determined by HeART for which HeART potentially suggests a lower redundancy scheme. The height of the shaded region in Fig. 9 denotes the AFR values protected by the useful life AFR value determined by HeART for that disk group.

It is important to note that even though Fig. 9 shows cumulative AFR behavior, HeART performs anomaly detection and online change point detection on AFRs calculated using monthly sliding windows. Thus, not only is the cumulative AFR always inside the shaded region, but the instantaneous failure rate for any 30-day period is also less than the determined AFR value. In fact, the first rise in the instantaneous failure rate is what determines the end of the useful life period. Fig. 10 shows the instantaneous failure rate of S-4 disks being lower than the determined useful life AFR value throughout the useful life period.

[8] We use Amazon's service so as to avoid re-implementing a state-of-the-art algorithm.

Disk groups		$r_{def} = MTTDL^{(14,10)}_{4.01\%AFR} = 1.46E+21$				$r_{def} = MTTDL^{(9,6)}_{4.01\%AFR} = 3.31E+16$				$r_{def} = MTTDL^{(3,1)}_{4.01\%AFR} = 6.36E+12$			
DG	AFR	$MTTDL^{r_{def}}$	r_{DG}	$MTTDL^{r_{DG}}$	Cost↓	$MTTDL^{r_{def}}$	r_{DG}	$MTTDL^{r_{DG}}$	Cost↓	$MTTDL^{r_{def}}$	r_{DG}	$MTTDL^{r_{DG}}$	Cost↓
S-4	4.01%	$1.46E+21$	(14, 10)	$1.46E+21$	**NA**	$3.31E+16$	(9, 6)	$3.31E+16$	**NA**	$6.36E+12$	(3, 1)	$6.36E+12$	**NA**
H-4A	1.82%	$7.57E+22$	(24, 20)	$3.56E+21$	**14%**	$7.80E+17$	(15, 12)	$7.20E+16$	**16%**	$6.80E+13$	(4, 2)	$1.70E+13$	**33%**
H-4B	2.04%	$4.28E+22$	(24, 20)	$2.01E+21$	**14%**	$4.94E+17$	(15, 12)	$4.56E+16$	**16%**	$4.83E+13$	(4, 2)	$1.21E+13$	**33%**
S-8C	2.07%	$3.98E+22$	(24, 20)	$1.87E+21$	**14%**	$4.66E+17$	(15, 12)	$4.30E+16$	**16%**	$4.62E+13$	(4, 2)	$1.16E+13$	**33%**
S-8E	2.48%	$1.61E+22$	(21, 17)	$1.58E+21$	**11%**	$2.26E+17$	(13, 10)	$3.99E+16$	**13%**	$2.69E+13$	(4, 2)	$6.72E+12$	**33%**
S-12E	2.44%	$1.75E+22$	(21, 17)	$1.72E+21$	**11%**	$2.41E+17$	(13, 10)	$4.26E+16$	**13%**	$2.82E+13$	(4, 2)	$7.06E+12$	**33%**

Table 3: Disk space saved by HeART by tuning the redundancy in the useful life of a disk group according to the observed disk group-specific AFRs. The units for $MTTDL$s is years. The cost savings are calculated for 3 default schemes: (14, 10) on AFR 4.01% disks, (9, 6) on AFR 4.01% disks and 3-replication (i.e. (3, 1)) on AFR 4.01% disks. Thus, the target reliability is the $MTTDL$ of the respective default redundancy schemes using a 4.01% AFR (the r_{def} table header). The max dimension of the scheme permitted during useful life for each disk group has at most twice the dimension of default redundancy scheme, i.e. 20 data chunks for $(14,10)$, 12 data chunks for $(9,6)$ and 2 data chunks for 3-replication.

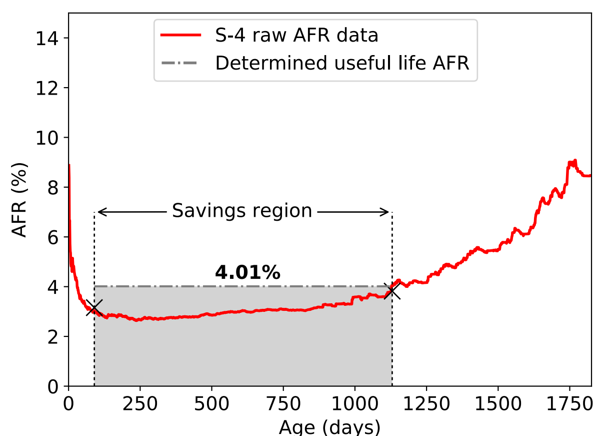


Figure 10: AFR of the S-4 disk group using a sliding window of 30 days. The determined useful life AFR value by HeART is conservative enough to subsume even the 30-day AFR values which vary more than the cumulative AFRs.

In contrast to S-4 (Fig. 9a), the H-4A (Fig. 9b) and H-4B (Fig. 9c) disk groups have a sudden occurrence of their respective wearout stages. The quick reactivity requirement explained in Section 3.1 comes into effect for these disk groups. How quickly HeART reacts to changes in the AFR is determined by how quickly failure data is provided to HeART. Since Backblaze maintains daily snapshots of disk health, the quickest reaction to an increased failure rate is on the day that the failures occur. In our evaluation, HeART successfully identifies the increased AFR on the very day it was provided with the increased AFR data.

Anomaly detection. As explained in Section 3.3, the anomaly detector successfully detects five anomalies in the lifetime of H-4B disks. Additionally, two anomalies are also detected for the H-4A disks. Correctly identifying anomalous events increased the identified useful life period of H-4B disks by over 5×. In the absence of anomaly detection, the end of useful life period would have been incorrectly identified at age 235 days (shown by point A in Fig. 7).

Cost savings per disk group. Table 3 summarizes the cost savings of employing disk group specific redundancy in their respective useful lifespans. Disk groups with similar AFRs are grouped together. As discussed in Section 2, we restrict the dimension (k) of the optimized code to at most 2× that of the default redundancy scheme (r_{def}). In each case of r_{def}, we set the target reliability to the $MTTDL$ achieved by using the highest-AFR disk group, which in the case of Backblaze are the S-4 disks.

It is important to note that the useful life AFRs determined by HeART are higher than the useful life AFRs shown in Fig. 1. Recall from Section 2, that HeART adds a (tunable) buffer above the useful life AFR determined at the end of infancy (which is an additional 25% by default). HeART chooses to be conservative in determining a useful life AFR value to ensure that reliability targets are comfortably met and to elongate the length of the useful life period to maximize benefits.

As in Section 2, we exemplify the space reduction for erasure coding schemes using $(14,10)$ and $(9,6)$ schemes, which are known to have been employed in large-scale data centers [13, 25, 26, 28].

First, we evaluate using $(14,10)$ as the default redundancy scheme. $(14,10)$ has the lowest storage overhead (1.4×) among the default redundancy schemes we evaluate, making it the hardest to find codes that meet the target $MTTDL$ and reduce overhead even further. Despite these constraints, HeART enables a 14% space reduction for H-4A, H-4B and S-8C disks by suggesting a $(24,20)$ code and a reduction of 11% for S-8E and S-12E disks by suggesting a $(21,17)$ code.

Next, we measure HeART's performance when using $(9,6)$ as the default redundancy scheme. We observe a space reduction of 16% on H-4A, H-4B and S-8C disks by using the maximum allowed $(15,12)$ redundancy scheme. For S-8E and S-12E disks, HeART suggests shorter $(13,10)$ code lengths compared to the above three disk groups in order to address their relatively higher determined AFR values, leading to a space reduction of 13%.

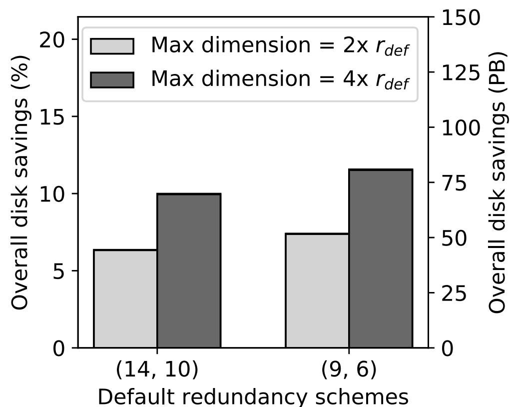


Figure 11: Overall space reduction achieved by HeART on the Backblaze dataset over the complete lifetime of every disk group, for erasure codes as the default scheme. For a maximum code dimension of up to $2 \times r_{def}$, we observe between $6-7.5\%$ space reduction and for a maximum code dimension of up to $4 \times r_{def}$, we observe between $10-12\%$ space reduction, translating to actual space savings of $40-80$ PBs.

Finally, we also include the cost reduction for the canonical redundancy scheme, 3-replication, for completeness. We see that HeART enables 33% space reduction for all disk groups. We note that if replication is employed primarily for availability, that data may not be a candidate for tuning redundancy through HeART.

For H-4A, H-4B and S-8C disks, HeART chose the $2\times$ max stripe-length for all three evaluated default redundancy schemes, extracting the maximum cost reduction (as explained in Table 2). Even with the maximum allowed stripe length, the $MTTDL$s for the above disks are approximately $2.5\times$ higher than the target $MTTDL$ value, suggesting further storage cost reductions if one is allowed even longer codes.

Overall cost reduction. To highlight the overall cost reduction achieved on the Backblaze disk fleet, we show the capacity-weighted cost savings in Fig. 11. This cost reduction is over the whole lifetime of the disks (including the unoptimized infancy and wearout periods) and for all six disk groups (including the unoptimized S-4 disks). We only show the benefits for the erasure coding schemes we evaluated, leaving out 3-way replication, since erasure codes are the more popular choice for data durability. The overall cost reduction achieved with the maximum stripe dimension being $2\times$ the default redundancy scheme is approximately 6% when using $(14,10)$ and approximately 7.5% when using $(9,6)$ as the default. If we relax the constraint of the maximum stripe dimension to $4\times$ the dimension of the default redundancy scheme, we can expect to achieve between $10-12\%$ overall space reduction. These modest percentage savings translate to significant savings in terms of actual storage space in large-scale clusters. For example, as shown on the right-side y-axis in Fig. 11, savings in storage space for the the Backblaze cluster range between $40-80$ PBs.

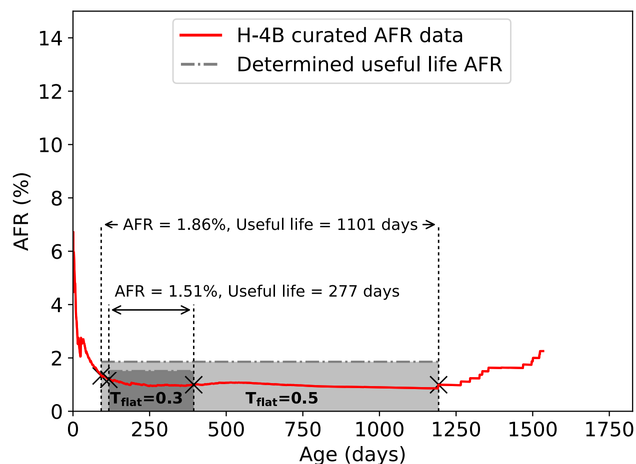


Figure 12: The effect of varying T_{flat} (AFR flatness threshold) on the H-4B disk group's AFR curve. Larger T_{flat} implies a higher useful life AFR along with a larger useful life period. The default value for T_{flat} in HeART is 0.5.

4.3 Sensitivity analysis

There are several configuration parameters that govern the behavior of HeART, of which most are dependent on the ready-made tools we have used for different components of our system (e.g., the threshold for anomaly scores when using RRCF for anomaly detection). There are, however, two fundamental parameters that are independent of which anomaly detector or change point detector is used.

Before going into the details about the two parameters, we note that the modulation of both the parameters only has an effect on the gains that our optimization can yield. *Neither of them affects correctness* of our framework or protection of data in any way. This allows operators of cluster storage systems to start with conservative values, observe the AFR behavior of their disks and accordingly choose apt values to minimize their costs without risking not reaching their reliability target. We next discuss the two parameters.

Flatness parameter (T_{flat}): T_{flat} is used to deduce the end of the infant mortality period. As mentioned in Section 3.4, the end of infancy is defined as the first 30+ day period beyond the change point detected after the first quarter such that the difference between maximum and minimum observed AFR is below the threshold T_{flat}. Thus, T_{flat} essentially determines the flatness in the AFR curve for a given period. Currently, we define T_{flat} to be 0.5. A larger T_{flat} value will reduce the length of infancy until it reaches 90 days, beyond which it will have no effect. A lower T_{flat} will enforce a stricter flatness criteria, typically causing end of infancy to be declared late. Ending infancy sooner potentially causes HeART to choose a larger value as the determined useful life AFR. This, in turn causes HeART to choose a stronger redundancy scheme (with higher space overhead) compared to one that would have been chosen with the determined AFR value derived as a result of a lower T_{flat} value. This reduces the achievable savings within the useful life period of the

disk group. As a tradeoff, we get a larger useful life period with a larger T_{flat}, since not only does infancy end sooner, but also the onset of wearout stage is postponed, since the increased useful life AFR now has higher tolerance to AFR variances throughout the useful life.

Fig. 12 shows the effect of varying T_{flat} on H-4B disks. We show the results for two different values $T_{flat} = 0.3$ and $T_{flat} = 0.5$. When T_{flat} was set to 0.3, we can see HeART declaring end of infancy at close to 100 days. Despite the buffer added to the determined useful life AFR at the end of infancy, the fluctuation in monthly AFRs caused a spike on day 394 to rise above the determined useful life AFR, causing HeART to announce end of useful life. In contrast, when $T_{flat} = 0.5$, infancy was declared to end on day 91, and the determined AFR value was high enough to tolerate the spike on day 394, increasing the useful life period by a significant amount.

Useful life AFR buffer: The AFR buffer is the conservative padding added to the useful life AFR determined at the end of infancy. Currently, the useful life AFR is determined as the AFR value at the end of infancy *plus an additional buffer*, the tunable AFR buffer parameter. The choice of the buffer value has similar tradeoffs to the flatness parameters discussed above. A high buffer value implies a more conservative approach to setting the determined useful life AFR. This will prolong the useful life period, but restrict the tuning of the redundancy scheme due to the high useful life AFR value determined (and thus reducing benefits). In contrast, setting a low buffer value will shorten the useful life period but allow more cost reductions during the useful life. Operators can set the buffer based on AFR fluctuations observed in their storage systems, which can stem anywhere from workload patterns to operational conditions.

5 Changes of the HeART (discussion)

HeART suggests redundancy schemes for use with each disk group during its useful life period, enabling safe redundancy tuning based on observed failure data. HeART recommends using the default redundancy scheme employed in the cluster during infancy and wearout periods. Exploiting HeART's recommendations in a cluster storage system requires some minor data placement policy changes and some online data redistribution. This section discusses both. Furthermore, since HeART changes redundancy schemes in response to the observed AFR curve, we also discuss estimating the required sample size (number of disks) for statistical confidence.

Data placement. HeART suggests per-disk-group redundancy schemes for hitting a particular data reliability target, based on observed AFRs. To use HeART safely, all data stored using a tailored redundancy scheme must be fully stored within the corresponding disk group—that is, all n chunks (data and parity chunks) of a stripe must be stored on disks within the same group. This restriction may be incompatible with some data placement schemes, such as Ceph's CRUSH [42, 43], and will add some complexity (conforming to disk group boundaries) to schemes that choose based on considerations like available capacity and load balancing.

Data redistribution. Many cluster storage systems include data redistribution mechanisms to deal with planned decommissioning and capacity/load balancing. Use of HeART will also require their use for transitioning from the default redundancy scheme (r_{def}) to a disk-group-specific scheme (r_{DG}), after infancy, and back again upon onset of wearout. Although this introduces extra redistribution load, we expect it to have a small impact on overall cluster load—at worst, it is two redistributions of the data over the 3–5 year deployment time of the disks.

Bulk changes should not be needed. On its face, HeART's redundancy scheme transitions appear to require massive redistributions, all at once. Not only would this be a load spike concern, not assuaged by the "not much load over the lifetime" argument, but it also potentially creates a capacity concern: a bulk transition from r_{DG} to the less space-efficient r_{def}, at the end of the useful life period, could require more space than is available. Fortunately, we do not expect this issue to arise in practice, as disks of a disk group are deployed over time rather than all at once (e.g., see Fig. 8) Since end of useful life is determined based on deployment age, rolling deployment will mean rolling wearout. Capacity exhaustion due to any given transition (of one subset of disks from one disk group to another) should not be as large a driver of slack capacity requirements, in practice, as other sources of variability (e.g., user demand). Furthermore, transitions from r_{def} to r_{DG} at the onset of useful life period can be gradually executed as this only reduces the achieved capacity savings and does not affect correctness or reliability guarantees.

Accurately characterizing bathtub curves. HeART is an online framework actively engaging with each disk group's bathtub curve. Naturally, it is important to understand how many disks one needs to observe before one can be confident of behavior of the bathtub curve of a particular disk group. There are several statistical bounds on the number of samples needed to reach a specified statistical confidence level. One technique is to use the *Chernoff-Hoeffding* Theorem [16, 1] to obtain a bound on the sample size (number of disks) required. For example, to achieve 99% confidence that the AFR of S-4 disks (which have an AFR of 3.29%, refer Table 2) is within the configured AFR-buffer of its determined useful life AFR (recall from Section 4, that the default AFR-buffer is an additional 25% over the useful life AFR value determined by HeART), the number of disks required is approximately 4,000. More advanced statistical techniques may provide tighter bounds and thereby indicate fewer required devices in a disk group.

6 HeART-less alternatives (related work)

The closest related work can be classified into disk reliability studies that identify reliability heterogeneity, techniques to predict disk failures using reliability data, and systems that automate redundancy scheme selection.

Numerous studies have been conducted to characterize disk failures [7, 10, 15, 18, 20, 23, 24, 29, 30, 31, 34]. Among the studies conducted on large production systems, Shah and Elerath [10, 34], Pinheiro et al. [24] and Ma et al. [20] independently verify that failure rates are highly correlated with disk manufacturers. These studies were conducted on the NetApp, Google and EMC disk fleets, respectively. Schroeder and Gibson also conducted a similar reliability study on disks from a high performance computing environment [30], not only highlighting reliability heterogeneity between disks deployed across systems, but also pointing out that disk datasheet reliability is very different from reliability observed in the field. Recently, Schroeder et al. [32] highlighted the heterogeneity in the reliability of different SSD technologies from four different manufacturers. Also, Schroeder et al. [29] reported heterogeneity of partial disk failures (sector errors) across makes/models for NetApp's disk fleet.

There have been numerous works that predict disk failures [14, 22, 36, 41, 47]. Among the more recent ones, Mahdisoltani et al. [21] use machine learning techniques to predict occurrence of partial disk errors using S.M.A.R.T. data. Anantharaman et al. [4] use random forests and recurrent neural networks to predict remaining useful life for HDDs. Both studies were performed on the Backblaze dataset.

Thereska et al. [37] built a self-prediction capability in cluster storage systems to assist in making informed redundancy and data placement decisions by answering *what-if* questions. It differs from HeART in that it does not perform and adapt to online analysis of reliability characteristics, relying on pre-knowledge of reliability metrics. Keeton et al. [19] built an optimization framework that automatically provided data dependability solutions to protect against site-level disasters by using information like workload patterns, and cost of recovery. This work also assumes prior knowledge of failure rates. Tempo [35] is a system that proactively creates replicas to ensure high durability in wide-area network distributed systems. It does this economically by allowing the user to specify a maximum maintenance bandwidth, and its design revolves around the efficient use of a distributed hash table. Carbonite [8] is a replica maintenance solution for distributed storage systems spread over the Internet, which makes efficient use of bandwidth in maintaining redundancy in the face of transient failures.

7 Conclusion

HeART enables more cost-effective data reliability for cluster storage systems. By robustly estimating per-disk-group *AFR*s and selecting the best redundancy settings for each, it avoids the space-inefficiency of one-size-fits-all redundancy schemes. Analysis of failure data for a large-scale production storage cluster shows that using HeART could achieve target data reliabilities with 11–33% fewer disks than popular configurations, offering huge potential cost savings.

Acknowledgements

We thank our shepherd Gala Yadgar and the anonymous reviewers for their valuable feedback and suggestions. We also thank the members and companies of the PDL Consortium (Alibaba, Amazon, Datrium, Dell EMC, Facebook, Google, Hewlett-Packard Enterprise, Hitachi, IBM, Intel, Micron, Microsoft Research, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma, Veritas, and Western Digital) and VMware for their interest, insights, feedback, and support.

References

[1] Chernoff-Hoeffding Theorem. https://en.wikipedia.org/wiki/Chernoff_bound.

[2] AMAZON. Kinesis. https://aws.amazon.com/kinesis/.

[3] AMAZON. Robust Random Cut Forest. https://docs.aws.amazon.com/kinesisanalytics/latest/sqlref/sqlrf-random-cut-forest.html.

[4] ANANTHARAMAN, P., QIAO, M., AND JADAV, D. Large Scale Predictive Analytics for Hard Disk Remaining Useful Life Estimation. In *2018 IEEE International Congress on Big Data (BigData Congress)* (2018).

[5] BACKBLAZE. Disk Reliability Dataset. https://www.backblaze.com/b2/hard-drive-test-data.html.

[6] BACKBLAZE. HDD SMART Stats. https://www.backblaze.com/blog/what-smart-stats-indicate-hard-drive-failures.

[7] BAIRAVASUNDARAM, L. N., GOODSON, G. R., PASUPATHY, S., AND SCHINDLER, J. An analysis of latent sector errors in disk drives.

[8] CHUN, B.-G., DABEK, F., HAEBERLEN, A., SIT, E., WEATHERSPOON, H., KAASHOEK, M. F., KUBIATOWICZ, J., AND MORRIS, R. Efficient Replica Maintenance for Distributed Storage Systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2006).

[9] COLE, G. Estimating drive reliability in desktop computers and consumer electronics systems. *Seagate Technology Paper TP* (2000).

[10] ELERATH, J. Hard-disk drives: The good, the bad, and the ugly. *Communication of ACM* (2009).

[11] ELERATH, J. G. AFR: problems of definition, calculation and measurement in a commercial environment. In *IEEE Reliability and Maintenance Symposium (RAMS)* (2000).

[12] ELERATH, J. G. Specifying reliability in the disk drive industry: No more MTBF's. In *IEEE Reliability and Maintenance Symposium (RAMS)* (2000).

[13] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in Globally Distributed Storage Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[14] HAMERLY, G., ELKAN, C., ET AL. Bayesian approaches to failure prediction for disk drives. In *International Conference on Machine Learning (ICML)* (2001).

[15] HEIEN, E., KONDO, D., GAINARU, A., LAPINE, D., KRAMER, B., AND CAPPELLO, F. Modeling and tolerating heterogeneous failures in large parallel systems. In *ACM / IEEE High Performance Computing Networking, Storage and Analysis (SC)* (2011).

[16] HOEFFDING, W. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association* (1963).

[17] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., YEKHANIN, S., ET AL. Erasure Coding in Windows Azure Storage. In *USENIX Annual Technical Conference (ATC)* (2012).

[18] JIANG, W., HU, C., ZHOU, Y., AND KANEVSKY, A. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *ACM Transactions on Storage (TOS)* (2008).

[19] KEETON, K., SANTOS, C. A., BEYER, D., CHASE, J. S., WILKES, J., ET AL. Designing for disasters. In *USENIX File and Storage Technologies (FAST)* (2004).

[20] MA, A., TRAYLOR, R., DOUGLIS, F., CHAMNESS, M., LU, G., SAWYER, D., CHANDRA, S., AND HSU, W. RAIDShield: characterizing, monitoring, and proactively protecting against disk failures. *ACM Transactions on Storage (TOS)* (2015).

[21] MAHDISOLTANI, F., STEFANOVICI, I., AND SCHROEDER, B. Proactive error prediction to improve storage system reliability. In *USENIX Annual Technical Conference (ATC)* (2017).

[22] MURRAY, J. F., HUGHES, G. F., AND KREUTZ-DELGADO, K. Hard drive failure prediction using non-parametric statistical methods. In *Springer Artificial Neural Networks and Neural Information Processing (ICANN/CONIP* (2003).

[23] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (RAID). In *ACM International Conference on Management of Data (SIGMOD)* (1988).

[24] PINHEIRO, E., WEBER, W.-D., AND BARROSO, L. A. Failure Trends in a Large Disk Drive Population. In *USENIX File and Storage Technologies (FAST)* (2007).

[25] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2013).

[26] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. *ACM Special Interest Group on Data Communication (SIGCOMM)* (2014).

[27] SAITO, Y., FRØLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. FAB: building distributed enterprise disk arrays from commodity components. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2004).

[28] SATHIAMOORTHY, M., ASTERIS, M., PAPAILIOPOULOS, D., DIMAKIS, A. G., VADALI, R., CHEN, S., AND BORTHAKUR, D. Xoring elephants: Novel erasure codes for big data. In *International Conference on Very Large Data Bases* (2013).

[29] SCHROEDER, B., DAMOURAS, S., AND GILL, P. Understanding latent sector errors and how to protect against them. *ACM Transactions on Storage (TOS)* (2010).

[30] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you? In *USENIX File and Storage Technologies (FAST)* (2007).

[31] SCHROEDER, B., AND GIBSON, G. A. Understanding failures in petascale computers. In *Journal of Physics: Conference Series* (2007), IOP Publishing.

[32] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. Flash Reliability in Production: The Expected and the Unexpected. In *USENIX File and Storage Technologies (FAST)* (2016).

[33] SEAGATE. Hard disk drive reliability and MTBF / AFR. `http://knowledge.seagate.com/articles/en_US/FAQ/174791en`.

[34] SHAH, S., AND ELERATH, J. G. Disk drive vintage and its effect on reliability. In *IEEE Reliability and Maintenance Symposium (RAMS)* (2004).

[35] SIT, E., HAEBERLEN, A., DABEK, F., CHUN, B.-G., WEATHERSPOON, H., MORRIS, R. T., KAASHOEK, M. F., AND KUBIATOWICZ, J. Proactive Replication for Data Durability. In *USENIX International Workshop on Peer-to-Peer Systems (IPTPS)* (2006).

[36] STROM, B. D., LEE, S., TYNDALL, G. W., AND KHURSHUDOV, A. Hard disk drive reliability modeling and failure prediction. *IEEE Transactions on Magnetics* (2007).

[37] THERESKA, E., ABD-EL-MALEK, M., WYLIE, J. J., NARAYANAN, D., AND GANGER, G. R. Informed data distribution selection in a self-predicting storage system. In *IEEE International Conference on Autonomic Computing (ICAC)* (2006).

[38] TRIVEDI, K. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Wiley, 2001.

[39] TRUONG, C., OUDRE, L., AND VAYATIS, N. A review of change point detection methods. In *arXiv:1801.00718v1 [cs.CE]* (2018).

[40] TRUONG, C., OUDRE, L., AND VAYATIS, N. ruptures: change point detection in python. In *arXiv:1801.00826v1 [cs.CE]* (2018).

[41] WANG, Y., MA, E. W., CHOW, T. W., AND TSUI, K.-L. A two-step parametric method for failure prediction in hard disk drives. *IEEE Transactions on industrial informatics* (2014).

[42] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006).

[43] WEIL, S. A., BRANDT, S. A., MILLER, E. L., AND MALTZAHN, C. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *ACM / IEEE High Performance Computing Networking, Storage and Analysis (SC)* (2006).

[44] XU, Y., SUI, K., YAO, R., ZHANG, H., LIN, Q., DANG, Y., LI, P., JIANG, K., ZHANG, W., LOU, J.-G., ET AL. Improving service availability of cloud systems by predicting disk error. In *USENIX Annual Technical Conference (ATC)* (2018).

[45] YANG, J., AND SUN, F.-B. A comprehensive review of hard-disk drive reliability. In *IEEE Reliability and Maintenance Symposium (RAMS)* (1999).

[46] ZHANG, Z., WANG, A., ZHENG, K., MAHESWARA, G. U., AND VINAYAKUMAR, B. Introduction to hdfs erasure coding in apache hadoop. *blog.cloudera.com* (2015).

[47] ZHAO, Y., LIU, X., GAN, S., AND ZHENG, W. Predicting disk failures with HMM-and HSMM-based approaches. In *Springer Industrial Conference on Data Mining (ICDM)* (2010).

[48] ZHU, B., WANG, G., LIU, X., HU, D., LIN, S., AND MA, J. Proactive drive failure prediction for large scale storage systems. In *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)* (2013).

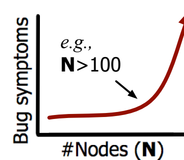
ScaleCheck: A Single-Machine Approach for Discovering Scalability Bugs in Large Distributed Systems

Cesar A. Stuardo, Tanakorn Leesatapornwongsa*, Riza O. Suminto, Huan Ke,
Jeffrey F. Lukman, Wei-Chiu Chuang†, Shan Lu, and Haryadi S. Gunawi

University of Chicago　　*Samsung Research America　　†Cloudera

Abstract

We present SCALECHECK*, an approach for discovering scalability bugs (a new class of bug in large storage systems) and for democratizing large-scale testing.* SCALECHECK *employs a program analysis technique, for finding potential causes of scalability bugs, and a series of colocation techniques, for testing implementation code at real scales but doing so on just a commodity PC.* SCALECHECK *has been integrated to several large-scale storage systems, Cassandra, HDFS, Riak, and Voldemort, and successfully exposed known and unknown scalability bugs, up to 512-node scale on a 16-core PC.*



Scalability bugs: *Latent bugs that are scale dependent, whose symptoms surface in large-scale deployments (e.g., N>100 nodes), but not necessarily in small/medium-scale (e.g., N<100) deployments.*

Examples:
"obvious symptom in 1000 nodes" [Cassandra bug #6127],
"with >500 nodes, ... trouble" [# 6409];
"16800 maps [recovery] was slow" [Hadoop #3711],
"1900 nodes, [namenode's] queue overflowed" [#4061];
"with >200 nodes, it doesn't work" [HBase #12139].

Figure 1: **Scalability bugs.** *Definition and quotes from scalability bug reports. Detailed examples are in §2a and §5.1.*

1 Introduction

Being a critical backend of many today's applications and services, storage systems must be highly reliable. Decades of research address a variety of storage dependability issues, including availability [44, 55], consistency [41, 77], durability [51, 72], integrity [36, 56], security [53, 71], and reliability [73, 74].

The dependability challenge grows as storage systems continue to *scale* in large distributed manners, especially in the last couple of years where the field witnesses a phenomenal deployment scale; Netflix runs tens of 500-node Cassandra clusters [34], Apple deploys a total of 100,000 Cassandra nodes [2], Yahoo! revealed the largest Hadoop/HDFS cluster with 4500 nodes [35], and Cloudera's customers deploy Spark on 1000 nodes [24, 27].

Is scale a friend or a foe [68]? On the positive side, scale surpasses the limit of a single machine in meeting increasing demands of compute and storage. On the negative side, this new era of "cloud-scale" storage systems has given birth to a new class of bug, *scalability bugs*, as defined in Figure 1.

From our in-depth study of scalability bugs (§2), we identified two challenges. First, scalability bugs are not easy to discover; their symptoms only surface in large deployment scales (*e.g.*, N>100 nodes). Protocol algorithms might seem scalable in design sketch, but until real deployment takes place, some bugs remain unforeseen (*i.e.*, there are specific implementation choices whose impacts at scale are unpredictable). Last but not least, their root causes are often hidden in the rarely tested background and operations protocols.

Second, the common practice of debugging scalability bugs is arduous, slow and expensive. For example, when customers report scalability issues, the developers might *not* have direct access to the same cluster scale and must wait for a "higher-level" budget approval for using large test clusters. As it stands today, many developers are heavily reliant on test clusters operated by large companies to do scale testing and only accessible to expert developers [26].

These realities raise the following question: how to discover latent scalability bugs and democratize large-scale testing? To this end, we introduce SCALECHECK, a concept that emphasizes the need to scale-check distributed system implementations at real scales, but do so cheaply on just one machine, hence empowering more developers to perform large-scale testing and debugging.

We design SCALECHECK with two components (SFIND and STEST) to address the two challenges. First, to reveal hidden scalability bugs, we build SFIND, a program analysis support for finding "scale-dependent loops." This strategy is based on our findings that the common root cause of scalability bugs is loops that iterate on data structures that grow as the system scales out (*e.g.*, an $O(N^3)$ loop that iterates through lists of node descriptors). Such loops can span across multiple functions and classes and iterate a va-

riety of data structures, hence the need for an automated approach. With SFIND output, developers can setup the necessary workloads that will exercise the loops and reveal any potential impacts to performance or availability.

Next, to democratize large-scale testing, we build STEST, a single-machine scale-testing framework. We target one machine because arguably the most popular testing practice is via unittests, which only requires a PC. Developers already invest a significant effort on unittests; their LOC can reach 20% of the system's code itself. However, current distributed systems and their unittests are not built with single-machine scale-testing in mind. For example, naively packing nodes as processes/VMs onto one machine quickly hits a colocation limit of 50 nodes/machine and we found no way to achieve a high colocation factor with black-box methods (no target system modification). Thus, we introduce novel colocation techniques such as global-event driven architecture (GEDA) in single-process cluster and processing illusion (PIL) with non-intrusive modification.

To show the generality and effectiveness of SCALECHECK, we have integrated SCALECHECK to a variety of large-scale storage systems, Cassandra [58], HDFS [18], Riak [30], and Voldemort [29], across a total of 15 earlier and newer releases. We scale-checked a total of 18 protocols (bootstrap, rebalance, add/decommission nodes, etc.), reproduced 10 known bugs and discovered 4 unknown critical scalability bugs (in Cassandra and HDFS). By only modifying the target systems in 179 to 918 LOC (and with a generic STEST library), we can colocate up to 512 nodes on a 16-core 32-GB commodity PC with high result accuracy (*i.e.*, observe a similar behavior as in the real-scale deployment).

SCALECHECK is unique compared to related work. For example, scalability simulation [39, 57] only checks models, but SCALECHECK checks implementation code. Extrapolation from "mini clusters" [57, 75, 80] does not work if the bug symptoms do not surface in small deployments, but SCALECHECK checks at real scales. Finally, emulation "tricks" run implementation code at real scale but in a smaller emulated environment [10, 48, 78] (the same category SCALECHECK can be put in), however existing techniques have limitations such as not addressing CPU contention and not finding potential causes automatically (more in §7). We also acknowledge many other works in improving storage scalability [42, 70], while our work emphasizes on scalability faults.

In summary, scalability bugs are new-generation bugs to combat in modern cloud-scale storage. Finding them without dependence of large clusters is a new research area to explore. In fact, this problem was discussed in a recent large meeting of Hadoop committee [26]. Currently, many new features in the alpha releases of Hadoop/HDFS still "sit on the shelf," *i.e.*, it is hard to test alpha (or even beta) releases at real scales as large production systems are not always accessible for testing. Some new features are still pushed and deployed but without much confidence. With this unideal reality, the committee agrees on the need for this new research, that it will increase their confidence on new releases [26]. Some companies began to invest in building scale-testing frameworks. For example, LinkedIn just released their scale-testing framework this year [9, 10] but it only emulates storage space specifically for HDFS.

For interested readers, we provide a supplemental file [1]. In the following sections, we present an extended motivation (§2), SCALECHECK design, application and implementation, and evaluation (§3-5) discussion, related work, and conclusion (§6-8).

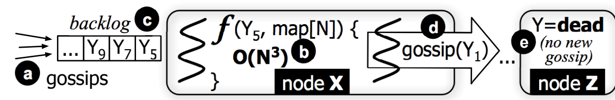


Figure 2: **An example bug (Section 2a).** *(a) Every second every node gossips to its peers its ring view and version number (e.g., Y gossiped up to version Y_9), (b) the receiving node (e.g., X) executes "$f()$" to synchronize the view, (c) when N is large, this $O(N^3)$ scale-dependent process creates a backlog of new gossips, (d) thus X keeps gossiping only the latest (old) versions (e.g., Y_1), (e) as Y's recent gossips are not propagated on time, other nodes (e.g., Z) mark Y as dead.*

2 Scalability Bugs

Scalability bugs are not a well-understood problem. To the best of our knowledge, we provide the first in-depth look at scalability bugs in scale-out systems.

(a) What is an example of scalability bugs? In Cassandra issue #c6127 in Figure 2 [7], the bug surfaced when bootstrapping a large cluster. Here, every node receives gossips from peer nodes (with their ring views), then find any difference to synchronize their views of the ring. The *root cause* is that during bootstrapping with many view changes, the gossip processing is *scale-dependent*, $O(N^3)$, as it iterates through the node's and peer's ring data structures and uses a list-copy mechanism. When N is large, this CPU-intensive process creates a backlog of new gossips, hence many nodes are inadvertently declared dead (and then alive after the gossips arrive). This repeating process leads to a *cluster instability* with thousands of "flappings" as N grows; a "*flap*" is when a node marks a peer as down and alive again. More detailed examples are presented in §5.1.

(b) Do they exist in many scalable systems? We have collected a total of 55 bugs in many modern distributed systems (13 in Cassandra, 5 in Couchbase, 6 in Hadoop, 13 in HBase, 16 in HDFS, 1 in Riak, and 1 in Voldemort). This is an arduous process due to the lack of searchable keywords for "scalability bugs"; we might have missed some other bugs. We post the full list in Section 2 of [1]. All the bugs were reported from large deployments (100-1900

nodes). We emphasize again that all these bugs can *only* be reproduced at scale.

(c) What are the root causes? We study the buggy code, patches, and developer discussions and find that the majority (52) of the bugs are caused by *scale-dependent loops*, which iterate scale-dependent data structures (*e.g.*, list of nodes); the rest is about logic bugs that can be caught with single-function testing. We break them down to three categories: (1) CPU-intensive loops (15 bugs); Figure 2 shows an example. (2) Disk IO loops (26 bugs); the pattern is similar to Figure 2 but the nested-loops contain disk IOs. (3) Locking-related loops (11 bugs); they can be in the form of locks inside the loops or vice versa. These patterns suggest that this problem lends itself to program analysis (§3.1).

(d) Where are they located? The bugs are within the user-facing read/write calls (12 bugs) and operational protocols (40 bugs) such as block report, bootstrap, consistency repair, decommission, de-replication, distributed fsck, heartbeat, job recovery, log cleaning, rebalance, and region assignment. This suggests that scalability correctness is not merely about the user-facing paths. Large systems are full of operational paths that must be scale-tested as well.

(e) When do they happen? User-facing read/write protocols run "all the time" in deployment, hence are continuously tested. Operational protocols, however, are not frequently exercised. In a stable-looking cluster, scalability bugs can linger silently until the buggy operational protocols are triggered (akin to buggy error handling). For the bugs in user-facing calls, most were triggered by unique workloads such as large deletions or writes after decommission.

(f) How do scalability bugs impact users? Scalability bugs can cause both performance and availability problems. Although many of the bugs are in the operational protocols, they can cascade to user-visible impacts. For example, when nodes are incorrectly declared dead, some data become unreachable; or scale-dependent operations in the master node (*e.g.*, in HDFS) can cause global lock contention, hence longer time to process user read/write requests.

(g) Why were the bugs not found before? First, the workloads and the necessary scales to cover the buggy protocols are not captured in the unittests as creating a scalable test platform is not straightforward [26]. Second, protocols might be scalable in design, but not in practice. Related to c6127 (Figure 2), the failure detector/gossiper [50] was adopted for its "scalable" design [58]. However, the design does not account for the gossip processing time during bootstrap/cluster-changes, which can be long, and the subsequent backlogs. To debug, the developers tried to "do the [simple] math" but failed [7]. Specific implementation choices such as overloading gossips with many other purposes (*e.g.*, announcing boot/rebalance changes) deviate from the original design sketch, hence the need for scale-testing the implementation code at real scales.

```
applyStateLocally (epStateMap)                    O(N^3)
  for (e : epStateMap)
    if (!localStateMap.get(e.key))
      handleChange(ep, e.val);
handleChange (ep, epState)
  for (subscriber : subscribers)
    subscriber.onJoin(ep, epState);
onJoin (ep, epState)
  for (e : epState)
    onChange (ep, e.key, e.val);
onChange (ep, state, val)
  if (state == STATUS)
    if (val.val[0] == NORMAL)
      handleNormal(ep, val.val);
```

```
handleStateNormal (ep, pieces)
  calcPendingRanges();
calcPendingRanges ()
  for (tab : nonSysTabs)
    calcPendingRanges(tab);
calcPendingRanges (tab)                           O(N^2)
  for (r : affectedRanges)
    tm.cloneOnlyTokenMap();
cloneOnlyTokenMap ()
  HashMap.create(ep.map);
create(map)
  for (m : map)
    newmap.add(m);                                O(N)
```

Figure 3: $O(N^3)$ **scale-depended loops (§3.1).** *The partial code segment above depicts the $O(N^3)$ loops in Figure 2.* SFIND *automatically tags* `epStateMap`, `affected- Ranges`, *and* `map` *as scale-dependent collections.*

(h) Are scalability bugs easy to debug and fix? The bugs took 1 month to fix on average with tens of back-and-forth discussions. One big factor of delayed fixes is the lack of budget for large test clusters as such luxury tends to only be accessible in large companies, but not to open-source developers [26]. Another factor is that debugging and fixing are not a single-iteration task; developers must repeatedly instrument the system and re-run at scale to pinpoint the root cause and test the patch.

3 SCALECHECK

We now present the design of SCALECHECK, which is composed of two parts to achieve two goals: SFIND (§3.1), a program analysis that exposes scale-dependent loops to developers, and STEST (§3.2), a set of colocation techniques that enable hundreds of nodes to be colocated on one machine for testing. While STEST produces accurate bug symptoms in most cases, it does not deliver accurate results when all nodes are CPU intensive. For this, we introduce PIL (§3.3), an emulation technique that provides processing illusion.

3.1 SFIND

The first challenge to address is: how to find scale-dependent loops? Unfortunately, it is not trivial as such loops can span multiple functions and iterate many scale-dependent collections (iterable data-structure instances such as `list`). In Figure 3, the $O(N^3)$ loops span 1000+ LOC, 3 classes, and 10 functions and iterate 3 scale-dependent collections. This difficulty motivates SFIND, a generic program analysis that helps developers pinpoint scale-dependent loops. Below are the three main steps of SFIND. For space, the pseudo-code can be found in our supplement, Section 3.1 of [1].

(1) Auto-tagging of scale-dependent collections: SFIND first automatically tags scale-dependent collections. This is done by growing the cluster and data sizes (*e.g.*, add nodes and add files/blocks) in steps. After each step, we record the size of each instantiated collection. When all the steps are done, we check each collection's growth tendency and

mark as scale dependent those whose size increases as the cluster/data size grows.

This, however, is insufficient due to two reasons. First, there are collections that only grow when background/operational tasks are triggered (§2d); thus, we must also run all non-foreground tasks. Second, there are "ephemeral" collections (*e.g.*, messages) whose content are scale-dependent but might have been garbage collected by the runtime. Given that the measurements are taken in steps, garbage collection can happen in between them so these collections will not be detected consistently, thus this phase must be iterated multiple times to remove such noise.

For Java systems, we track heap objects and map them to their instance names by writing around 1042 LOC of analysis on top of Java language supports such as JVMTI [67] and Reflection [22]. This phase also performs a dataflow analysis to taint all other variables derived from scale-dependent collections. In our experience, by scaling out to just 30 nodes (30 steps), which can be done easily on one machine, scale-dependent collections can be clearly observed (though not the symptoms). This phase found 32 scale-dependent collections in Cassandra (three in Figure 3) and 12 in HDFS.

(2) Finding scale-dependent loops: With the tagging, SFIND then automatically searches for scale-dependent loops, specifically by tainting loops (`for`, `while`) as well as recursive functions that iterate through the scale-dependent collections, performing a control-flow analysis to construct the nested Big O complexity of each loop, and identifying the loop contents (CPU/instructions only, IOs, or locks). With these steps, in Figure 3 for example, SFIND can mark `applyStateLocally` as an $O(N^3)$ function.

We also cover a special "implicit loop" – a synchronized (locking) function in a node that is being called by all the peer nodes. A common example is in the master-worker architecture where all the N worker nodes RPC into a master's lock-protected function. When N grows, there is a potential of lock contention (congestion) to the function (examples are in §5.1). SFIND also handles such scenarios by tagging RPC classes and searching for functions called by the peer nodes.

(3) Reporting and triaging: SFIND finds 131 scale-dependent loops in Cassandra and 92 in HDFS, hence the need for triaging. For example, if a function g has lower complexity than f, and g is within the call path of f, then testing f can be prioritized. For every nested loop to test, SFIND reports the relevant control- and data-flows from the outer-most to inner-most loop, along with the entry points (either client/admin RPCs or background daemon threads). The entry points are finally ranked by counting the number of spanned scale-dependent lines of code, the theoretical complexity (in terms of scale-dependent data structures), the number of IO operations (including reads/writes) and the number of blocking operations (including locking and operations that block waiting for a future result) in that path. The theoretical complexity is not by itself a complete indicator of potential bottlenecks. For example, an entry point reported with high complexity ,*e.g.* $O(N^3)$, but with no IO/Blocking operations on its code path might not be as bottleneck prone as one reported with less complexity, *e.g.* $O(N)$, but many IO/Blocking operations on its code path. This ranking helps developers prioritize and create the necessary test workloads. For example, in Figure 3, the $O(N^3)$ path is only exercised if the cluster bootstraps from scratch when peers do not know about each other (hinted from the "`if(!localStateMap.get())`", "`onChange()`", "`state==STATUS`" and "`val==NORMAL`"). SFIND reports that this entry point spans over 6700 scale-dependent lines of code and performs over $20N$ IO and $4N$ blocking operations, which implies that it is likely to become a bottleneck as the cluster size grows and should be prioritized.

Creating test workloads from SFIND report is a manual process. Automated test generation is possible for single-machine programs/libraries [38], however, we are not aware of any work that automates such process in the context of real-world, complex, large-scale distributed systems. We put our work in the context of DevOps culture [62] where developers are testers and vice versa, which (hopefully) simplifies test workload creation.

3.2 STEST

The next challenge is: how to test scale-dependent loops at real scales (hundreds of nodes) on one machine? Many scale-dependent loops were unfortunately not subjected to testing because existing unittest frameworks do not scale. Below we describe the hurdles to achieve a high colocation factor. Starting in Section 3.2.1, we began with black-box methods (no/small target system modification).

Unfortunately, we found that existing systems are *not* built with single-machine scale-testing in mind (the theme of this section); we faced many colocation bottlenecks (memory/CPU contentions and context switching delays) that limit large colocation. In Section §3.2.2, we will describe our solutions to achieve single-machine scale-testable systems with minimal changes. All the methods we use are summarized in Table 1 using Cassandra as an example. Abbreviations of our methods (*e.g.*, NP, SPC, GEDA) are added for ease of reference in the evaluation.

3.2.1 Black-Box Approaches

• **Naive Packing (NP):** The easiest setup is (naively) packing all nodes as processes on a single machine. However, we did not reach a large colocation factor, which is caused by the following reasons.

(a) Memory bottlenecks: Many distributed systems today are implemented in managed languages (*e.g.*, Java, Erlang) whose runtimes consume non-negligible memory overhead. Java and Erlang VMs, for example, use around 70 and 64

	#Nodes per PC	LOC added	Colocation bottlenecks
Black/gray-box approaches (§3.2.1)			
(a) Naive (NP)	50	–	Memory, proc. switch
(b) SPC	70	–	User-kernel switch
(c) SPC+Stub	120	+91	Context switch
White-box approaches (§3.2.2)			
(d) GEDA	130	+581	CPU
(e) GEDA+PIL	512	+246	CPU

Table 1: **Colocation strategies and bottlenecks (§3.2).**

MB of memory per process respectively. We also tried running nodes as Linux KVM VMs and using KSM (kernel samepage merging) tool. Interestingly, the tool does not find many duplicate pages even though the VMs/processes are supposed to be similar (as reported elsewhere [65]). Overall, including Cassandra's memory usage, per-node memory consumption reaches 100 MB. Thus, a 32-GB machine can only colocate around 300 nodes.

(b) Process context switches: Before we hit the memory bottleneck (*e.g.*, reach 300 nodes), we observed that the target systems' "inaccuracy" is already high when we colocate just 50 nodes. For measuring inaccuracy, we measure several application-level metrics; for example, in Cassandra, if gossips should be sent every 1 second, but are sent every 1.3 second, then the inaccuracy is 30%. We use 10% as the maximum acceptable inaccuracy/event lateness. We noticed high inaccuracies even before we hit the CPU bottlenecks (*i.e.*, CPU has not reached 90% utilization). We suspected that the process context switches could be the reasons.

(c) Managed-language VM limitations: We also found that managed-language VMs are backed by advanced services. For example, Erlang VMM contains a DNS service that sends heartbeat messages among connected VMs. When hundreds of Erlang VMs (one for each Riak node) run on one Erlang VMM, the heartbeat messages cause a "network" overflow that undesirably disconnects Erlang VMs (also reported in [40]). Naive packing is infeasible.

• **Single-Process Cluster (SPC) + Network Stub:** To address the bottlenecks above, we deployed all nodes as threads in a single process. Surprisingly, our target systems are not easy to run in this "single-process cluster." For example, Cassandra developers bemoan the fact that their gossip/fault-detector protocols are not adequately scale-tested [15, 28] because Cassandra (and many other systems) uses "singleton" design pattern for simplicity (but bad for modularity) [32]. That is, most global states are static variables that cannot be modularized to per-node isolated variables.

Our strawman attempt was a redesign to a more modular one, which costs us almost 3000 LOC (and no longer a black-box method); Cassandra developers also attempted a similar method to no avail [15, 28]. We found another way: leveraging class loader isolation support from the language runtime [23], which is rarely used but fits SPC purpose. In Java systems, we can manipulate the class loader hierarchy such that a node's main thread (and all child threads) use an isolated set of Java class resources, not shared with those belonged to other nodes, hence no target system modification. Very recently, we found that Cassandra developers also begin to develop a similar method to address this problem [8].

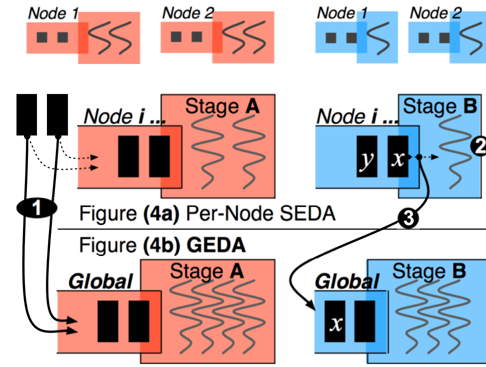


Figure 4: **Global Event Driven Arch. (Section 3.2.2).** *The figure format follows [79, Figure 6].*

By SPC-ing Cassandra, we now hit a colocation limit of 70 nodes (Table 1b), but still have not reached the memory or CPU bottlenecks. We suspected thread and/or user-kernel context switching as a root cause. We removed the latter by creating a generic network stub that (de)marshalls inter-node messages and skips the OS. This stub is also helpful in reducing network memory footprints under higher colocation. For example, in Voldemort, the nodes communicate via Java NIO [25] which is fast but contains buffers and connection metadata that take up memory space and prevent >200-node colocation (more in §5.4). For Cassandra, the network stub allows up to 120-node colocation (Table 1c).

3.2.2 A White-Box Approach

Adding network stub is our last black-box approach as we found no other way to reduce thread context switching in a black-box way. In fact, we observed a massive thread context switching issue. In P2P systems such as Cassandra, *each* node spawns *a* thread to listen from *a* peer. Thus, just for messaging, there are N^2 threads to manage for the whole cluster. This can be solved by using `select()`-like system call [21], which would reduce the problem to N threads. However, we still observed around $N \times 26$ active threads – each node still runs multiple service stages (gossiper, failure detector, etc.), each can be multi-threaded. A high colocation factor will spawn thousands of threads.

• **Global Event Driven Arch. (GEDA):** To address the problem, we must redesign the target system, but with minimal changes. We leverage the staged event-driven architecture (SEDA) [79] (Figure 4a), common in server code, in which each service/stage (in each node) exclusively has an event queue and a thread pool. In STEST mode, we convert SEDA to a *global-event driven architecture* (GEDA; Figure

4b). That is, for every stage, there is only *one* queue and *one* thread pool for the *whole* cluster. As an example, let's consider a periodic gossip service. With 500-node colocation, there are 500 threads in SPC, each sending a gossip every second. With GEDA, we only deploy a few threads (matched with the number of available cores) shared among all the nodes for sending gossips. As another example, for gossip processing stage, there is only one global gossip-receiving queue shared among all the nodes.

GEDA works with a minimal code change to the target system. Logically, as events are about to be *enqueued* into the original per-node event queues (① in Figure 4), we redirect them to GEDA-level event queues, to be later processed by GEDA worker threads. This only requires ∼10 LOC change per stage (as we use aspect-oriented programming [3]). While simple, care must be taken for single-threaded/serialized stage. For example, Cassandra's gossip processing is intentionally single-threaded to prevent concurrency issues. This is illustrated in case ② in Figure 4 where the per-node stage is serialized (*i.e.*, y must be processed after x). Here, *if* the events are forwarded down during *enqueue*, GEDA's multiple threads will break the program semantic (*e.g.*, x and y can be processed concurrently). Thus, for single-threaded/serialized stage, we must interpose at *dequeue* time (③ in Figure 4), which costs ∼50 LOC change per stage (details in §3.2 of [1]). Thus, by default we interpose at enqueue (small changes) and at dequeue for single-threaded stage (more changes).

Adding GEDA to Cassandra only costs us 581 LOC (Table 1d) and is simple; the same 10-50 LOC method above is simply repeated across all the stages. Overall, GEDA does not change the logic of the target systems, but successfully removes some delays that should have never existed in the first place, as if the nodes run exclusively on independent machines. For HDFS tests, GEDA enables 512-node colocation (§5.4) but for some Cassandra tests, it only enables around 130-node colocation (Table 1d), which we elaborate in the next section.

3.3 Processing Illusion (PIL)

Finally, the last challenge we address is: how to produce accurate results (*i.e.*, the same bug symptoms observed in real-scale deployment) when colocating hundreds of CPU-intensive nodes? We found that STEST is sufficient for accurately revealing bug symptoms in scale-dependent lock-related loops or IO serializations, as these root causes do not contend for CPUs. For CPU-intensive loops, STEST is also sufficient for master-worker architecture where only one node is CPU intensive (*e.g.*, HDFS master).

However, for CPU-intensive loops in P2P systems such as Cassandra, where *all* nodes are busy, the bug symptoms reported by STEST are not accurate. For example, for Cassandra issue #c6127 (§2a), in 256-node real deployment, we observed around 2000 flappings (the bug symptom) but 21,000 flappings in STEST. The inaccuracy gets worse as we scale; with N CPU-intensive nodes on a C-core machine, roughly N/C nodes contend on a given core.

To address this, we need to emulate CPU-intensive processing by supplementing STEST with *processing illusion* (PIL), an approach that replaces an actual processing with `sleep()`. For example, for c6127, we can replace the expensive gossip/stage-changes processing (see Figures 2 and 3), with `sleep(t)` where `t` is an accurate timing of how long the processing takes.

The intuition behind PIL is similar to the intuition behind other emulation techniques. For example, Exalt provides an illusion of storage space; their insight was "how data is processed is not affected by the content of the data being written, but only by its size" [78]. Similarly, PIL provides an illusion of compute processing; our insight is that *"the key to computation is not the intermediate results, but rather the execution time and eventual output."* In other words, with PIL, we will still observe the overall timing behaviors and the corresponding impacts accurately.

PIL might sound outrageous, but it is feasible as we address the following concerns: how a function (or code block) can be safely replaced with `sleep()` *without* changing the whole processing semantic (§3.3.1) and how we can produce the output and predict the timing "`t`" if the actual compute is skipped (§3.3.2)?

3.3.1 PIL-Safe Functions

Our first challenge is to ensure that functions (or code blocks) can be safely replaced with `sleep()`, but still retain the cluster-wide behavior and unearth the bug symptoms. We name such functions as "PIL-safe functions." We identify two main characteristics of such functions: (1) Memoizable output: a PIL-safe function must have a memoizable (deterministic) output based on the input of the function. (2) Non-pertinent IOs: if a function performs local/remote disk IOs that are not pertinent to the correctness of the corresponding protocol, the function is PIL-safe. For example, in c6127, there is a ring-table checkpoint (not shown) needed for fault tolerance but is irrelevant (never read) during bootstrapping.

We extend SFIND to SFIND$_{PIL}$, which includes a static analysis that finds code blocks in scale-dependent loops that can be safely PIL-ed. SFIND$_{PIL}$ analyzes the content of each loop in functions related to the relevant cluster state and checks for two cases: (1) The loop performs operations that affect the cluster state, so we need to insert pre-memoization and replay code to record/reconstruct the cluster state [1, §3.3]. We consider all variables involved in the execution of a target protocol as relevant states. While our static analysis tool eases the identification of these variables, programmer intervention can help for additional verification. In (2), the loop performs non-pertinent operations only (such as IO). In this case, we can automatically replace the loop with a *sleep* call without affecting the behavior of the protocol.

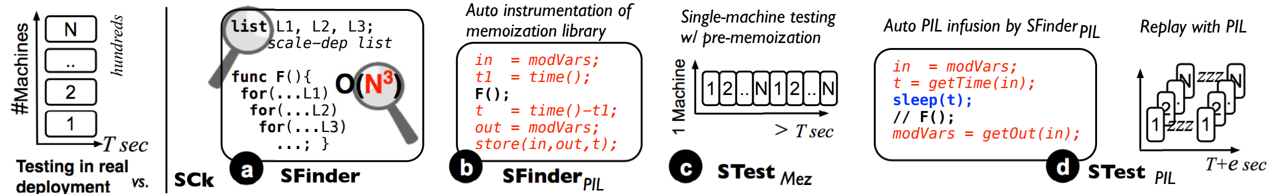


Figure 5: **SCALECHECK complete automated flow (Section 3.4).** *"SCk" represents* SCALECHECK*. The left-most figure illustrates testing in real deployments, where testing time is fast (T) but requires N machines. Stages (a) to (d) reflect the automated* SCALECHECK *process as described in Section 3.4.* STEST$_{mez}$ *in stage (c) runs on one machine but will take some time (>T).* STEST$_{PIL}$ *in stage (d) still runs on one machine but only consumes a similar time as in deployment testing (T+e) and can be replayed numerous times.*

3.3.2 Pre-Memoization (with Determinism)

As PIL-safe functions no longer perform the actual computation, the next question to address is: how do we manufacture the output such that the global behavior is not altered (*e.g.*, rebalancing protocol should terminate successfully)?. For functions with no pertinent outputs, we just need to do time profiling but not output recording. For functions with pertinent outputs, our solution is *pre-memoization*, which records input-output pairs and the processing time, specifically a tuple of three items (`ByteString in`, `out`, `long nanoSec`) indexed by `hash(in)`), which represent the to-be-modified variables before and after the function is executed and the processing time, respectively (Figure 5b).

Another challenge encountered is non-determinism: the state of each node (the input) depends on the order of arriving messages (which are typically random). Let's consider Riak's bootstrap+rebalance protocol where eventually all nodes own a similar number of partitions. A node initially has an unbalanced partition table, receives another partition table from a peer node, then inputs it to a rebalance function, and finally sends the output to a *random* node via gossiping. *Every* node repeats the same process until the cluster is balanced. In a Riak cluster with N=256 and P=64, there are in total 2489 rebalance iterations with a set of specific inputs in *one* run. *Another* run of the protocol will result in a *different* set of inputs due to gossip randomness. Our calculation shows that there are $(N^{NP})^2$ possible inputs.

To address this, during pre-memoization, we also record non-determinism such as message orderings such that order determinism is enforced during replay. For example, across different runs, a Riak node now receives gossips from the same sequence of nodes. With order determinism, pre-memoization and SCALECHECK work as follow: **(1)** We first run the whole cluster on a real deployment and interpose sleep-safe functions. **(2)** When sleep-safe functions are executed, we record the inputs and corresponding outputs to a *memoization database* (SSD-backed files). **(3)** During this pre-memoization phase, we *record message non-determinism* (*e.g.*, gossip send-receive pairs and their timings). **(4)** After pre-memoization completes, we can repeatedly run SCALECHECK wherein order determinism is enforced (*e.g.*, no randomness), sleep-safe functions replaced with PIL, and their outputs retrieved from the memoization database. Note that steps 1-3 are the only steps that require real deployment.

Other than this, similar to the theme in the previous section that existing systems are not amenable to single-machine testing, we found similar issues such as the use of wall-clock time which essentially incapacitates memoization and replay. Here, we convert wall-clock time to "cluster start time + elapse time" in 296 LOC (Table 1e).

3.4 Putting It All Together

Figure 5a-d summarizes the complete four stages of SCALECHECK: ⓐ SFIND searches for scale-dependent loops which helps developers create test workloads. ⓑ For test workloads that show CPU busyness in all nodes, SFIND$_{PIL}$ finds PIL-safe functions and inserts our pre-memoization library calls. Next, STEST now works in two parts. ⓒ STEST$_{mez}$ (without PIL) will run the test on a real cluster, but just one time, to pre-memoize PIL-safe functions and store the tuples to a SSD-backed database file. ⓓ STEST$_{PIL}$ (with PIL) will then run by having SFIND$_{PIL}$ remove the pre-memoization library calls, replace the expensive PIL-safe function with `sleep(t)`, and insert our code that constructs the memoized output data. SCALECHECK also records message ordering during STEST$_{mez}$ and replays the same order in STEST$_{PIL}$ (not shown).

As another benefit, SCALECHECK can also ease real-scale debugging efforts. First, the only step that consumes more time is the no-PIL pre-memoization phase (Figure 5c), up to 6x longer time than real-deployment testing (§5.5). However, this is only a one-time overhead. Most importantly, developers can repeatedly re-run STEST$_{PIL}$ (Figure 5d) as many times as needed (tens of iterations) until the bug behavior is completely understood. In STEST$_{PIL}$, the protocol under test runs in a similar duration as if all the nodes run on independent machines.

Second, some fixes can be tested by only re-running the last step; for example, fixes such as changing the failure detector Φ algorithm (for c6127), caching slow methods (c3831), changing lock management (c5456), and enabling parallel processing (v1212). However, if the fixes involve a complete redesign (*e.g.*, optimized gossip processing in c3881, decentralized to centralized rebalancing in r3926), STEST$_{mez}$ must be repeated.

	Cass	HDFS	Riak	Vold
STEST-able systems	918	179	217	800
SFIND code	4026 (generic)			
STEST library	6047 (generic)			

Table 2: **Integrations LOC (Section 4).** *More explanations are in Section 4 of [1]. We will release our code publicly.*

4 Application and Implementation

Table 2 quantifies the application of SCALECHECK techniques to a variety of distributed systems, Cassandra [58], HDFS [18], Riak [30], and Voldemort [29]. The major system-specific change is achieving "STEST-able systems" (*i.e.*, supporting SPC and GEDA), which range between 179 to 918 LOC (less than 1 % of the target code size). This is analogous to how file systems code are modified to make them "friendlier" to `fsck` [52, 63]. The rest is the generic SFIND and STEST library code (pre-memoization, auto PIL insertion, message order determinism support, AspectJ utilities). SFIND was built with Eclipse AST Parser [11] to support Java programs. We leave porting to Erlang's parser [12, 13] as future work.

Generality: We show the generality of SCALECHECK with two major efforts. First, we scale-checked a total of 18 protocols: 8 Cassandra (*e.g.*, bootstrap, scale-out, decommission), 8 HDFS (*e.g.*, decommission, block reports, snapshot), 1 Riak (rebalance), and 1 Voldemort (rebalancing) protocols (full list in §4 of [1]). A protocol can be built on top of other protocols (*e.g.*, bootstrap on gossip and failure detection protocols). Second, for exposing known bugs, we applied SCALECHECK to a total of 10 earlier releases: 4 Cassandra, 4 HDFS, 1 Riak, and 1 Voldemort old releases. For finding unknown bugs, we also ran SCALECHECK on recent releases of the four systems.

5 Evaluation

We now evaluate SCALECHECK: Is SCALECHECK effective in exposing scalability bugs (§5.1-5.2), accurate (§5.3), scalable and efficient (§5.4-5.5)? We compare SCALECHECK with real deployments of 32 to 512 nodes, deployed on at most 128 machines (testbed group limit), each has 16-core AMD Opteron(tm) with 32-GB DRAM.Our target protocols only make at most 2 busy cores per node, which justifies why we pack 8 nodes per one 16-core machine for the real deployment.

5.1 Exposing Scalability Bugs

Table 3 lists the 10 real-world bugs we use for benchmarking SCALECHECK. We chose these 10 bugs (among the 55 bugs we studied) because the reports contain detailed descriptions of the bugs, which is important for us to create the "input" (*i.e.*, the test cases). Figure 6 shows the accuracy

Bug#	N	Protocol	Metric	T_m	T_{pil}
c6127 [7]	≥256	Bootstrap	*#flaps*	2h	15m
c3831 [6]	≥256	Decomm.	*#flaps*	17m	9m
c3881 [5]	≥64	Add nodes	*#flaps*	7m	5m
c5456 [4]	≥256	Add nodes	*#flaps*	16m	4m
r3926 [31]	≥128	Rebalance	T_{Comp}	6h	2h
v1212 [33]	≥128	Rebalance	T_{Comp}	22h	–
h9198 [19]	≥256	Blk. report	Q_{Size}	8m	–
h4061 [17]	≥256	Decomm.	T_{Lock}	6h	–
h1073 [16]	≥512	Pick nodes	T_{Comp}	1m	–
h395 [20]	≥512	Blk. report	T_{Comp}	5m	–

Table 3: **Bug benchmark (§5.1).** *The table lists the scalability bugs we use for benchmarking* SCALECHECK. *"**c**" stands for Cassandra, "**h**" for HDFS, "**r**" for Riak, and "**v**" for Voldemort. The "**N**" column represents the #nodes for the bug symptoms to surface. The "**Metric**" column lists the quantifiable metrics of the bug symptoms; T_{Comp}, T_{Lock}, and Q_{Size} denote computation time, lock time, and queue size, respectively. The "T_m" and "T_{pil}" columns quantify the duration of the pre-memoization* (STEST_{mez}) *and PIL replay* (STEST_{PIL}) *stages when $N \geq 256$, as discussed in §5.5. "–" implies PIL is unnecessary.*

of SCALECHECK in exposing the 10 bugs using the "bug-symptom" metrics in Table 3 (the first bug c6127 will be shown later in Section 5.3 and the last bug h395 is omitted in Figure 6 for space).

Results summary: First, SCALECHECK is effective and accurate in exposing scalability bugs, some of which only surface in 256+ nodes. As shown, for Cassandra and Riak bugs where all nodes are CPU intensive, PIL is needed for accuracy (`SCk+PIL` vs. `Real` lines in Figures 6a-d), but for the rest, STEST suffices (`SCk` vs. `Real` in 6e-f).

Second, SCALECHECK can help developers prevent recurring bugs; the series of Cassandra bugs (as described later below) involves the same protocols (gossip, rebalance, and failure detector) and create the same symptom (high *#flaps*). As code evolves, it can be continuously scale-checked with SCALECHECK.

Third, different systems of the same type (*e.g.*, key-value stores, master-worker file systems) implement similar protocols. The effectiveness of SCALECHECK methods in scale-checking the different protocols above can be useful to many other distributed systems.

Bug descriptions: We now briefly describe the bugs. Longer descriptions can be found in Section 5.1 of [1].

(a) Figure 6a: In Cassandra c3831 [6] when a node X is removed, all other nodes must own X's key-partitions. This scale-dependent, CPU-intensive "pending keyrange calculation" cause cluster-wide flapping (the y-axis), observable in 256+ nodes. The fix caches the outputs of slow methods.

(b) Figure 6b: c3881 [5] is similar to the previous bug (c3831), but the fix was obsolete as the concept of multi-

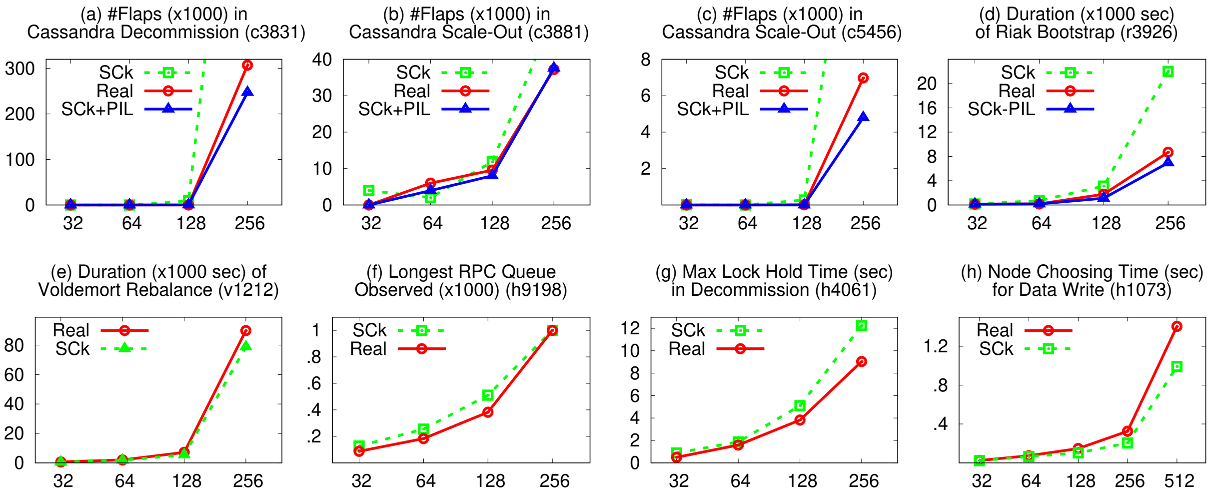


Figure 6: **SCALECHECK effectiveness in exposing scalability bugs (Section 5.1).** *"SCk" represents* SCALECHECK*. The bugs are listed in Table 3. The* ***x-axis*** *represents the number of nodes (N). The figure title describes the* ***y-axis****, i.e., the bug symptom metrics as recorded in "Real" deployment vs.* SCALECHECK*. For Cassandra and Riak bugs (a-d), where all nodes are CPU-intensive, the bug symptoms are inaccurate without PIL ("*`SCk`*" lines). However, with PIL ("*`SCk+PIL`*" lines), the bug symptoms are relatively accurate as in the real deployment scenarios. For Voldemort and HDFS bugs (e-h), where there is no concurrent CPU busyness, PIL is not needed.*

ple key-partitions per node was added. The calculation is now scale-dependent on $N \times P$. This causes CPU spikes and massive flapping during scaling out; the bug surfaced in 64+ nodes (when 32+ new nodes are added to existing 32+ nodes). The bug was fixed with a complete redesign of the pending keyrange calculation.

(c) Figure 6c: Interestingly, c5456 [4] is a bug in the *same* protocol as above. The previous fix was obsolete again as pending range calculation is now multi-threaded; range calculations can happen concurrently. However, this new design introduces a new coarse-grained lock that can block gossip processing for a long time, thus introduces flapping (in 256+ nodes). The fix changed the lock management.

(d) Figure 6d: In r3926 [31], Riak's rebalancing algorithm employed 3 complex stages (claim-target, claim-hole, full-rebalance) to converge to a perfectly balanced ring. Each node runs this CPU-intensive algorithm on *every* bootstrap-gossip received. The larger the cluster, the longer time the perfect balance is achieved (a high y value in 128+ nodes).

(e) Figure 6e: In v1212 [33], Voldemort's rebalancing was not optimized for large clusters; it led to more stealer-donor partition transitions as the cluster size grows (128+ nodes). The fix changed the stealer-donor transition algorithm.

(f) Figure 6f: In h9198 [19], incremental block reports (IBRs) from HDFS datanodes to the namenode acquire the global master lock (*i.e.*, a special worker-to-master "loop" as explained in §3.1). As N grows, more IBR calls acquire the lock. The IBR requests quickly backlog the namenode's IPC queue; with 256 nodes, the IPC queue hits the max of 1000 pending requests; y=1 ($\times$1000). When this happens, user requests are undesirably dropped by the namenode. The fix batches the IBR request processing. In HDFS, to emulate large blocks, we reuse the "`TinyDataNode`" class (1KB blocks) that the developers already use in the unit tests.

(g) Figure 6g: In h4061 [17], when D datanodes are decommissioned, the blocks must be replicated to the other $N-D$ nodes. Every 5 minutes, the `DecommissionMonitor` thread in the namenode iterates all the block descriptors to check if the D nodes can be safely decommissioned (when all data replications complete). This thread, unfortunately, must hold the global file system lock. When N is 256+, this process can hold the lock (*i.e.*, stall user requests) for more than 10 seconds (y>10). The fix used a dedicated thread to manage decommissioning and refined the algorithm.

(h) Figure 6h: In h1073 [16], for a new file creation, the namenode calls a `chooseTarget` function to sort a list of target datanodes from their distances from the writer and choose the best nodes. When N and the replication factor are large, it can take more than one second to choose. The fix modified the sorting algorithm.

(i) Finally, in h395 [20] (figure not shown for space), datanodes send block reports too frequently and when N>512 nodes, the namenode spends more time in this background process as opposed to serving users.

5.2 Discovering Unknown Bugs

We also integrated SCALECHECK to recent stable versions of Cassandra, HDFS, Riak, and Voldemort, and found 1 unknown bug in Cassandra and 3 bugs in HDFS.

For Cassandra, SFIND pointed us to another nested scale-dependent loop. We created the corresponding test case and SCALECHECK showed that cluster-wide flapping resurfaces again but only in 512-node deployment. As an example, decommissioning just only one node already caused almost 100,000 flaps. The developers confirmed that the bug is related to a design problem. To prevent flappings, the devel-

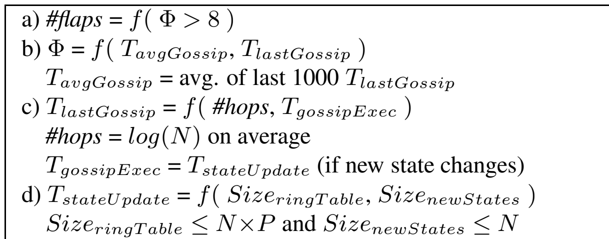

a) *#flaps* = $f(\ \Phi > 8\ )$
b) $\Phi = f(\ T_{avgGossip}, T_{lastGossip}\ )$
$T_{avgGossip}$ = avg. of last 1000 $T_{lastGossip}$
c) $T_{lastGossip} = f($ *#hops*, $T_{gossipExec}\ )$
#hops = $log(N)$ on average
$T_{gossipExec} = T_{stateUpdate}$ (if new state changes)
d) $T_{stateUpdate} = f(\ Size_{ringTable}, Size_{newStates}\ )$
$Size_{ringTable} \leq N \times P$ and $Size_{newStates} \leq N$

Figure 7: **Cassandra internal metrics (§5.3).** *Above are the metrics we measured within the Cassandra bootstrap protocol for measuring* SCALECHECK *accuracy (Figure 8). "f" represents "a function of" (i.e., an arbitrary function).*

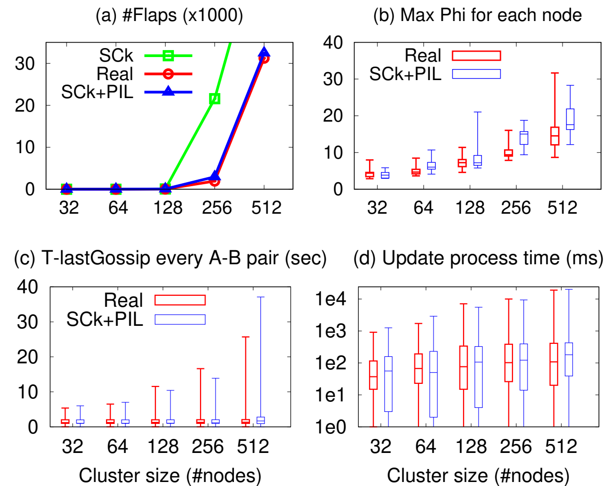


Figure 8: **Accuracy in exposing c6127 (§5.3).** *The figures represent the metrics presented in Figure 7, measured in real deployment ("Real") and in* SCALECHECK *("SCk") with different cluster sizes (32, 64, 128, 256, and 512 in the* ***x-axis***). *The* ***y-axes*** *(the metrics) are described in the figure titles.*

opers suggested us to add/remove node one at a time with 2-minute separation, which means scaling-out/down 100 nodes will take over 3 hours (*i.e.*, this bug impedes instant elasticity). The developers recently started a new initiative for designing "Gossip 2.0" to scale to 1000+ nodes [14].

For Riak and Voldemort, we found that their latest-stable bootstrap/rebalance protocols do not exhibit any scalability bug, up to 512 nodes.

For HDFS, we found 3 instances of scale-dependent loops that hold the entire namenode read/write lock (also confirmed by the developers). Specifically, SFIND reports the following number of lines executed:

```
FSNamesystem.getSnapshotDiff         N*(85*B+17)
DatanodeManager.refreshDatanodes     N*(136*B+137)
FSNamesystem.metaSave                N*(50*B+21)
```

Here, "B" represents the number of blocks per datanode (*e.g.*, 10,000). The first function, `getSnapshotDiff`, contains a bug that the HDFS developers were hunting for 4 weeks, as the unresponsive-namenode impact recently affected a customer. In this path, there is a recursive function iterating on a list of files and blocks and a conditional path that makes ACL lookups which causes the namenode to be unresponsive for more than 40 seconds in at least a 512-node deployment. Similar symptoms were also reproduced for the second and third bugs (`refreshDatanodes` and `metaSave`). The developers say these bugs are dangerous because if the namenode is paused for 45 seconds, it will cause a heavy failover. They also say these bugs are hard to find in a million-plus lines of code. More details/graphs are in §5.2 of [1].

5.3 Accuracy

The goal of our next evaluation is to show that PIL-infused SCALECHECK mimics similar behaviors as in real-deployment testing and is accurate not only in the final bug-symptom metric but also in the detailed internal metrics. For this, we collected roughly 18 million values. For space, we only focus on c6127 [7] (see §2a).

Figure 7a-d shows the internal metrics that we measured within Cassandra failure detection protocol for *every pair* of nodes; the algorithm runs on every node A for every peer B. Figures 8a-d compare in detail the accuracy of STEST without PIL ("`SCk`") and STEST$_{PIL}$ with PIL ("`SCk+PIL`"), respective to the real-deployment testing ("`Real`").

(a) Figure 8a shows the total number of flaps (alive-to-dead transitions) observed in the whole cluster during bootstrapping. STEST by itself will not be accurate if all nodes are CPU intensive (§3.3). However, with PIL, SCALECHECK closely mimics real deployment scenarios. Next, Figure 7a defines that *#flaps* depends on Φ [50]. Every node A maintains a Φ for a peer B (a total of $N \times (N-1)$ variables to monitor).

(b) Figure 8b shows the maximum Φ values observed for every peer node; for graph clarity, from here on we only show with-PIL results. For example, for the 512-node setup, the whisker plots show the distribution of the maximum Φ values observed for each of the 512 nodes. As shown, the larger the cluster, more Φ values exceeds the threshold value of 8, hence the flapping. Figure 7b points that Φ depends on the average inter-arrival time of when new gossips about B arrives at A ($T_{avgGossip}$) and the time since A heard the last gossip about B ($T_{lastGossip}$). The point is that $T_{lastGossip}$ should not be much higher than $T_{avgGossip}$.

(c) Figure 8c shows the whisker plots of gossip inter-arrival times ($T_{lastGossip}$) that we collected for every A-B pair (millions of gossips as a gossip message contains N gossips of the peer nodes). The figure shows that in larger clusters, new gossips do not arrive as fast as in smaller clusters, especially at high percentiles. Figure 7c shows that $T_{lastGossip}$ depends on how far B's new gossips propagate through other nodes to A (*#hops*) and the gossip processing time in each hop ($T_{gossipExec}$). The latter ($T_{gossipExec}$) is essentially the state-update processing time ($T_{stateUpdate}$), triggered whenever there are state changes.

(d) Figure 8d (*in log scale*) shows the whisker plots of the state-update processing time ($T_{stateUpdate}$). In the 512-node setup, we measured around 25,000 state-update invocations. The figure shows that at high percentiles, $T_{stateUpdate}$ is *scale dependent (the culprit)*. As shown in Figure 7d, $T_{stateUpdate}$ complicatedly depends on a scale-dependent 2-dimensional input ($Size_{ringTable}$ and $Size_{newStates}$). A node's $Size_{ringTable}$ depends on how many nodes it knows, including the partition arrangement ($\leq N \times P$) and $Size_{newStates}$ ($\leq N$), which increases as cluster size grows.

5.4 Colocation Factor

This section shows the maximum colocation factor ScaleCheck can achieve as each technique is added one at a time on top of the other. To recap, the techniques are: single-process cluster (SPC), network stub (Stub), global event driven architecture (GEDA), and processing illusion (PIL). The results are based on a 16-core machine.[1]

Maximum colocation factor ("MaxCF"): A maximum colocation factor is reached when the system behavior in ScaleCheck mode starts to "deviate" from the real deployment behavior. Deviation happens when one or more of the following bottlenecks are reached: (1) high average CPU utilization (>90%), (2) memory exhaustion (nodes receive out-of-memory exceptions and crash), and (3) high event "lateness."

Queuing delays from thread context switching can make events late to be processed, although the CPU utilization is not high. We instrument our target systems to measure *event lateness* of relevant events (as described in §3.2.2). We use 10% as the maximum acceptable event lateness. Note that the residual limiting bottlenecks come from the main logic of the target protocols, not removable with general methods.

Results and observations: Figure 9 shows different sequences of integration to our four target systems and the resulting maximum colocation factors. We make several important observations from this figure.

First, when multiple techniques are combined, they collectively achieve a high colocation factor (up to 512 nodes for the three systems respectively). For example, in Figure 9a, without using PIL in Cassandra, MaxCF only reaches 136. But with PIL, MaxCF significantly jumps to 512. When we increased the colocation factor (+100 nodes) beyond the maximum, we hit the residual bottlenecks mentioned before; at this point, we did not measure MaxCF with small increments (*e.g.*, +1 node) due to time limitation.

Second, distributed systems are implemented in different ways. Thus, integrations to different systems face different sequences of bottlenecks. To show this, we tried different sequences of integration sequences. For example, in Cassandra (Figure 9a), our integration sequence is +SPC, +Stub, +GEDA, and +PIL (as we hit context switching overhead before CPU). For Riak (Figure 9b), we began with PIL as we hit CPU limitation first before hitting Erlang VMM network overflow which requires SPC (§3.2.1), and Riak does not require GEDA because Erlang, as an event-driven language, manages thread executions as events (more in Section 5.4 of [1]). For Voldemort (Figure 9c), we began with SPC and then network stub to reduce Java VM and Java NIO memory overhead respectively, and PIL so far is not needed as the tested workload does not involve parallel CPU-intensive operations. For HDFS (Figure 9d), we only need SPC and GEDA but not PIL as only the master node that is CPU intensive (but not the datanodes).

[1] So far, we consistently use the same testbed, but a higher-end machine can be used in the future.

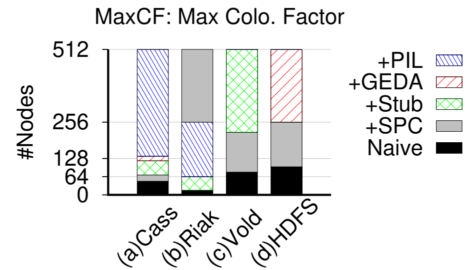


Figure 9: **Maximum colocation factor (Section 5.4).** *The colocation factor reached as each technique is added.*

Finally, it is the *combination* of all techniques that make ScaleCheck effective. For example, while in Figure 9a we apply the sequence of SPC+Stub+GEDA+PIL resulting in PIL as the dominant factor, in another experiment we applied a different sequence PIL+SPC+Stub and failed to hit 512 nodes, not until GEDA is added and becomes the dominant factor.

5.5 Pre-Memoization and Replay Time

The "T_m" and "T_{pil}" columns in Table 3 on page quantifies the duration of the pre-memoization (STest_{mez}) and PIL-based replay (STest_{PIL}) stages when $N \geq 256$. For example, for CPU-intensive bugs such as c6127, the pre-memoization time takes 2 hours while the PIL-based replay is only 15 minutes (similar to the real-deployment test); for r3926, it is 6 vs. 2 hours. Pre-memoization does not necessarily take $N\times$ longer time because one node only consumes 2 cores (while the machine has 16 cores) and also not every node is busy all the time.

5.6 Test Coverage

SFind labeled 32 collections in Cassandra and 12 in HDFS as scale dependent. From these, SFind identified 131 and 92 scale-dependent loops in Cassandra and HDFS (out of more than 1500 and 1900 total loops) respectively. So far, we have tested 57 (44%) and 64 (69%) of the loops in Cassandra and HDFS. The time-consuming factor is the manual creation of new test cases that will exercise the loops (see end of §3.1).

We emphasize that SFIND is *not* a bug-finding tool, hence the reason why we do not report false positives. A more complete picture of SFIND's output can be found in Section 5.6 of our supplemental document [1].

6 Discussion

At the moment, our work focuses on scale-dependent CPU/processing time (§2c), and the "scale" here implies the scale of *cluster size*. However, there are other scaling problems that lead to IO and memory contentions [46, 69, 76], usually caused by the scale of *load* [37, 47] or *data size* [64]. For emulating data size, we are only aware of one work, Exalt [78], which is orthogonal to SCALECHECK (more in §7). In our bug study, we learn that some load or data-size related bugs can be addressed with accurate modeling [47] (*e.g.*, d dead nodes will add $d/(N-d)$ load to every live node) and some others can already be reproduced with a single machine (*e.g.*, loading as much file metadata to check the limit of HDFS memory bottleneck [76]). Nevertheless, we will continue our study of these other scaling dimensions, especially as scaling bugs in datacenter distributed systems is not a well-understood problem.

So far, SCALECHECK is limited as a single-machine framework, which integrates well to the de-facto unit-test style. To increase colocation factor, a higher-end machine can be used. Another approach is to extend SCALECHECK to run on multiple machines. However, this means that we need to enable back the networking library, which originally already caused a colocation bottleneck. We also acknowledge as a limitation that adding new code will also add new maintenance costs. In future work, we intend to approach zero-effort emulation.

Finally, SFIND by itself is not sufficient to reveal scalability bugs. Building a program analysis that covers all paths and understands the cascading impacts is challenging. Not all scale-dependent loops imply buggy code.

7 Related Work

In Section 1, we briefly discussed related work in four categories: real-scale testing/benchmarking (direct, but not economical) [26, 59], large-scale simulation (easy to run, but rarely used for server infrastructure code) [39, 54, 57], extrapolation (easy to run, but missing bugs in small training scale) [57, 61, 75, 80], and emulation. SCALECHECK falls in this category and below discuss three closely related works [10, 48, 78].

Exalt [78] targets IO-intensive (Big Data) scalability problems where storage capacity is the colocation bottleneck. Exalt's library (Tardis) compresses users' data to zero bytes on disk. With this, Exalt can co-locate 100 space-emulated HDFS datanodes per machine. As the authors stated, their approach "may not discover scalability problems that arise at the nodes that are being emulated" [78]. Thus, it cannot cover P2P systems where the scale-dependent code is in all the nodes. However, as Exalt targets storage space emulation and SCALECHECK addresses processing time emulation, we believe they complement each other. LinkedIn's Dynamometer is similar to Exalt [10].

DieCast [48], invented for network emulation, can colocate processes/VMs on a single machine as if they run individually, by "dilating" time. The trick is adding a "time dilation factor" (TDF) support [49] into the VMM. For example, TDF=5 implies that for every second of wall-clock time, each emulated VM believes that time has advanced by only 200 ms (1/TDS second). DieCast was only evaluated with a colocation factor (TDF) of 10 as the testing time significantly increases proportionally to the TDF; colocating 500 nodes will increase testing time by 500 times. DieCast was introduced for answering "what if the network is much faster?", but not specifically for single-machine scale-testing. Another significant difference is that both Exalt and DieCast papers do not present an in-depth bug study.

In terms of related work in the static/program analysis space, Clarity [66] and Speed [45] use static analysis to look for potential performance bottlenecks by focusing on redundant traversals and precise complexity bounding. Both approaches are evaluated in libraries. However, for distributed systems, real-scale testing can help reveal unintended complex component interactions, and not all scale-dependent loops cause problems.

Finally, a recent work also highlights the urgency of combating scalability bugs [60]. The work, however, does not employ methodical and incremental changes, only suggests a manual approach, and reproduces only 4 bugs in 1 system.

8 Conclusion

Technical leaders of a large cloud provider emphasized that "the most critical problems today is how to improve testing coverage so that bugs can be uncovered during testing and not in production" [43]. It is now evident that scalability bugs are new-generation bugs to combat, that existing large-scale testing is arduous, expensive, and slow, and that today's distributed systems are not single-machine scale-testable. Our work addresses these contemporary issues and will hopefully spur more solutions in this new area.

9 Acknowledgments

We thank Cheng Huang, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. This material was supported by funding from NSF (grant Nos. CNS-1350499, CNS-1526304, CNS-1405959, and CNS-1563956) as well as generous donations from Dell EMC,Google, Huawei, and NetApp, and CERES center.

References

[1] Anonymized document for ScaleCheck supplementary materials (also submitted to HotCRP), for interested reviewers. https://tinyurl.com/sck-supp-mat.

[2] Apache Cassandra. https://en.wikipedia.org/wiki/Apache_Cassandra.

[3] AspectJ. www.eclipse.org/aspectj.

[4] Cassandra bug: Large number of bootstrapping nodes cause gossip to stop working. https://issues.apache.org/jira/browse/CASSANDRA-5456.

[5] Cassandra bug: reduce computational complexity of processing topology changes. https://issues.apache.org/jira/browse/CASSANDRA-3881.

[6] Cassandra bug: scaling to large clusters in GossipStage impossible due to calculatePendingRanges. https://issues.apache.org/jira/browse/CASSANDRA-3831.

[7] Cassandra bug: vnodes don't scale to hundreds of nodes. https://issues.apache.org/jira/browse/CASSANDRA-6127.

[8] Cassandra feature: Make it possible to run multi-node coordinator/replica tests in a single JVM. https://issues.apache.org/jira/browse/CASSANDRA-14821.

[9] Dynamometer Github Repository. https://github.com/linkedin/dynamometer.

[10] Dynamometer: Scale Testing HDFS on Minimal Hardware with Maximum Fidelity. https://engineering.linkedin.com/blog/2018/02/dynamometer--scale-testing-hdfs-on-minimal-hardware-with-maximum.

[11] Eclipse Java development tools. http://www.eclipse.org/jdt/.

[12] Elvis: Erlang Style Reviewer. https://github.com/inaka/elvis.

[13] Erlang man page: Dialyzer. http://erlang.org/doc/man/dialyzer.html.

[14] Gossip 2.0. https://issues.apache.org/jira/browse/CASSANDRA-12345.

[15] Gossip is inadequately tested. https://issues.apache.org/jira/browse/CASSANDRA-9100.

[16] Hadoop bug: DFS Scalability: high CPU usage in choosing replication targets and file open. https://issues.apache.org/jira/browse/HADOOP-1073.

[17] Hadoop bug: Large number of decommission freezes the Namenode. https://issues.apache.org/jira/browse/HADOOP-4061.

[18] HDFS. https://hortonworks.com/apache/hdfs/.

[19] HDFS bug: Coalesce IBR processing in the NN. https://issues.apache.org/jira/browse/HDFS-9198.

[20] HDFS bug: DFS Scalability: Incremental block reports. https://issues.apache.org/jira/browse/HDFS-395.

[21] Java NIO Selector. http://tutorials.jenkov.com/java-nio/selectors.html.

[22] Java Reflection API. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html.

[23] JBoss AS 7 classloading. http://www.mastertheboss.com/jboss-server/jboss-as-7/jboss-as-7-classloading.

[24] Meet Cloudera's Apache Spark Committers. http://blog.cloudera.com/blog/2015/09/meet-clouderas-apache-spark-committers/.

[25] NIO in Voldemort: Non-heap memory usage. https://groups.google.com/forum/#!topic/project-voldemort/J7ADKefjR50.

[26] Personal Communication from Andrew Wang and Wei-Chiu Chuang of Cloudera and Uma Maheswara Rao Gangumalla of Intel; they are also part of Apache Hadoop Project Management Committee (PMC) members.

[27] Personal Communication from Imran Rashid (Software Developer at Cloudera).

[28] Personal Communication from Jonathan Ellis, Joel Knighton, Josh McKenzie, and other Cassandra developers.

[29] Project Voldemort. http://www.project-voldemort.com/voldemort/.

[30] Riak. http://basho.com/products/riak-kv.

[31] Riak bug: Large ring_creation_size. http://lists.basho.com/pipermail/riak-users_lists.basho.com/2011-April/003895.html.

[32] Singletons are pathological liars. https://testing.googleblog.com/2008/08/by-miko-hevery-so-you-join-new-project.html.

[33] Voldemort bug: Number of partitions. https://groups.google.com/forum/#!msg/project-voldemort/3vrZfZgQp2Y/Uqt8NgJHg4AJ.

[34] Running Netflix on Cassandra in the Cloud. https://www.youtube.com/watch?v=97VBdgIgcCU, 2013.

[35] Why the world's largest Hadoop installation may soon become the norm. http://www.techrepublic.com/article/why-the-worlds-largest-hadoop-installation-may-soon-become-the-norm/, 2014.

[36] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)*, 2008.

[37] Peter Bodik, Armando Fox, Michael Franklin, Michael Jordan, and David Patterson. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[38] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[39] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[40] Natalia Chechina, Huiqing Li, Amir Ghaffari, Simon Thompson, and Phil Trindera. Improving the network scalability of Erlang. *Journal of Parallel and Distributed Computing*, 90-91:22–34, April 2016.

[41] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[42] James Cipar, Gregory R. Ganger, Kimberly Keeton, Charles B. Morrey III, Craig A. N. Soules, and Alistair C. Veitch. LazyBase: trading freshness for performance in a scalable database. In *Proceedings of the 2012 EuroSys Conference (EuroSys)*, 2012.

[43] 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 Symposium on File and Storage Technologies (FAST)*, 2016.

[44] Daniel Ford, Franis Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlna. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[45] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009.

[46] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[47] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure Recovery: When the Cure Is Worse Than the Disease. In *The 14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, 2013.

[48] Diwaker Gupta, Kashi Venkatesh Vishwanath, and Amin Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[49] Diwaker Gupta, Kenmeth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[50] Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. The Phi Accrual Failure Detector. In *The 23rd Symposium on Reliable Distributed Systems (SRDS)*, 2004.

[51] Alyssa Henry. Cloud Storage FUD: Failure and Uncertainty and Durability. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST)*, 2009.

[52] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *IEEE 2nd Workshop on Hot Topics in System Dependability (HotDep)*, 2006.

[53] Bernard Dickens III, Haryadi S. Gunawi, Ariel J. Feldman, and Henry Hoffmann. StrongBox: Confidentiality, Integrity, and Performance using Stream Ciphers for Full Drive Encryption. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[54] Håvard D. Johansen, Robbert Van Renesse, Ymir Vigfusson, and Dag Johansen. Fireflies: A secure and scalable membership and gossip service. *ACM Transactions on Computer Systems*, 33:5:1–5:32, June 2015.

[55] Kimberly Keeton, Cipriano A. Santos, Dirk Beyer, Jeffrey S. Chase, and John Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST)*, 2004.

[56] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[57] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, Bowen Zhou, Zhezhe Chen, and Feng Qin. Debugging High-Performance Computing Applications at Massive Scales. *Communications of the ACM (CACM)*, 58(9), September 2015.

[58] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2009.

[59] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi. The Case for Drill-Ready Cloud Computing. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[60] Tanakorn Leesatapornwongsa, Cesar A. Stuardo, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, and Haryadi S. Gunawi. Scalability Bugs: When 100-Node Testing is Not Enough. In *The 16th Workshop on Hot Topics in Operating Systems (HotOS XVII)*, 2017.

[61] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Dongsheng Li, and Xicheng Lu. PCatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In *Proceedings of the 2018 EuroSys Conference (EuroSys)*, 2018.

[62] Thomas A. Limoncelli and Doug Hughe. LISA '11 Theme – DevOps: New Challenges, Proven Values. *USENIX ;login: Magazine*, 36(4), August 2011.

[63] 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 Symposium on File and Storage Technologies (FAST)*, 2013.

[64] 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 *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[65] Kazunori Ogata and Tamiya Onodera. Increasing the transparent page sharing in java. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[66] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[67] Oracle. JVMTM Tool Interface version 1.2. https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html.

[68] John Ousterhout. Is Scale Your Enemy, Or Is Scale Your Friend?: Technical Perspective. *Communications of the ACM (CACM)*, 54(7), July 2011.

[69] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[70] Swapnil Patil and Garth Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST)*, 2011.

[71] Jason K. Resch and James S. Plank. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST)*, 2011.

[72] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: a Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 2012 EuroSys Conference (EuroSys)*, 2012.

[73] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST)*, 2010.

[74] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[75] Rong Shi, Yifan Gan, and Yang Wang. Evaluating Scalability Bottlenecks by Workload Extrapolation. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2018.

[76] Konstantin V. Shvachko. HDFS Scalability: The Limits to Growth. *USENIX ;login:*, 35(2), April 2010.

[77] 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 *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[78] Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[79] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[80] Bowen Zhou, Milind Kulkarni, and Saurabh Bagchi. Vrisha: Using Scaling Properties of Parallel Programs for Bug Detection and Localization. In *Proceedings of the 20th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2011.