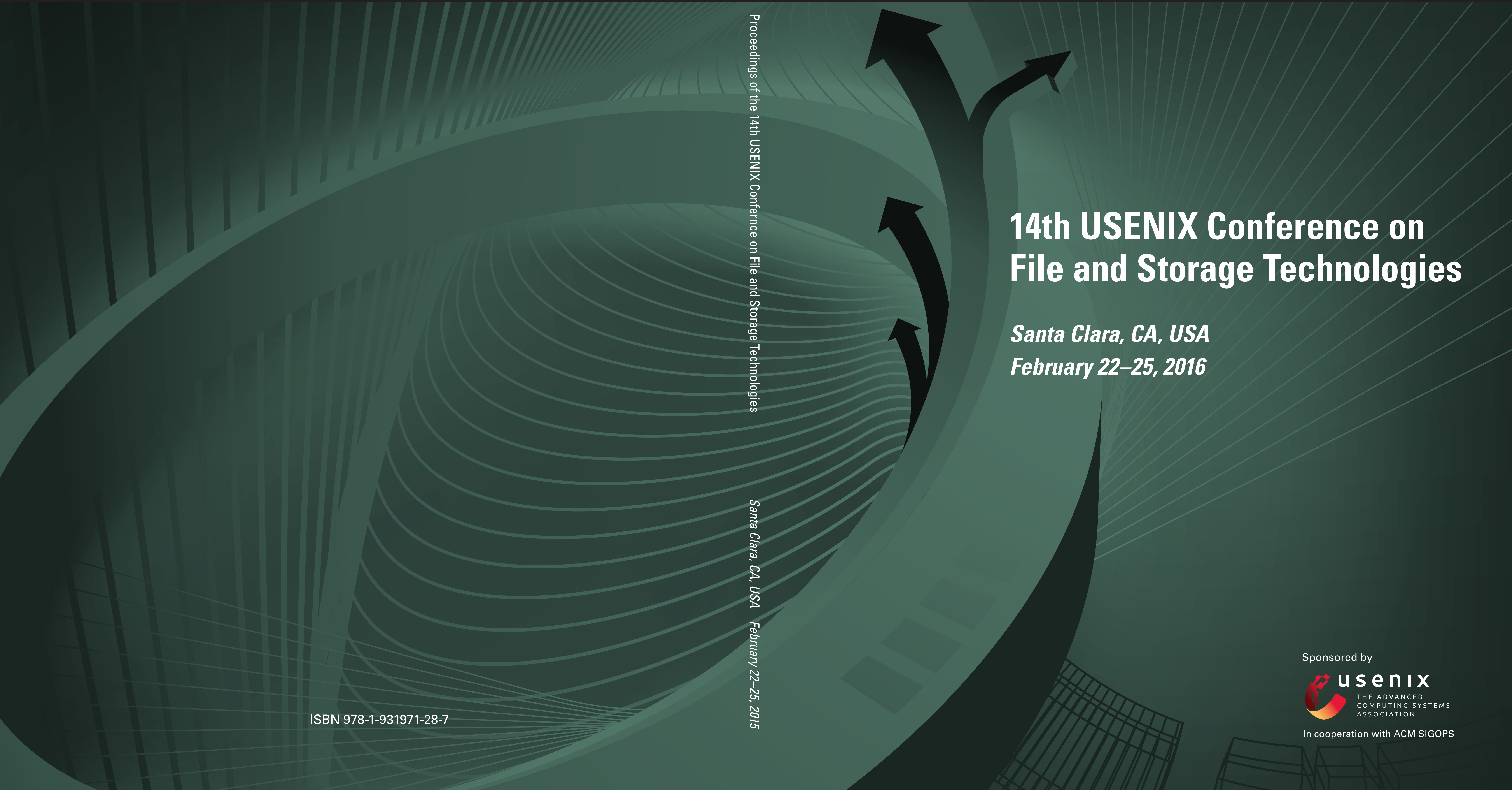


ISBN 978-1-931971-28-7

usenix ASSOCIATION

Proceedings of the 14th USENIX Conference on File and Storage Technologies

Santa Clara, CA, USA February 22–25, 2015

conference proceedings

14th USENIX Conference on File and Storage Technologies

Santa Clara, CA, USA

February 22–25, 2016

Sponsored by

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

In cooperation with ACM SIGOPS

Thanks to Our FAST ’16 Sponsors

Platinum Sponsor

NetApp

Gold Sponsors

Google NSF VERITAS vmware

Silver Sponsors

EMC² HUAWEI Microsoft Research redhat Tintri

Bronze Sponsors

facebook Hewlett Packard Enterprise IBM Research Radian MEMORY SYSTEMS SNIA

Media Sponsors and Industry Partners

ACM *Queue*
ADMIN magazine
CRC Press
Distributed Management Task Force (DMTF)

FreeBSD Foundation
Linux Pro Magazine
LXer
No Starch Press

O’Reilly Media
Raspberry Pi Geek
UserFriendly.org

ISBN 978-1-931971-28-7

Thanks to Our USENIX Supporters

USENIX Patrons

Facebook Google Microsoft Research
NetApp VMware

USENIX Benefactors

ADMIN Magazine *Linux Pro Magazine* Symantec

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16):

February 22–25, 2016
Santa Clara, CA, USA

Conference Organizers

Program Co-Chairs

Angela Demke Brown, *University of Toronto*

Florentina Popovici, *Google*

Program Committee

Atul Adya, *Google*

Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*

Angelos Bilas, *University of Crete and FORTH-ICS*

Jason Flinn, *University of Michigan*

Garth Gibson, *Carnegie Mellon University and Panasas, Inc.*

Haryadi Gunawi, *University of Chicago*

Cheng Huang, *Microsoft Research and Azure*

Eddie Kohler, *Harvard University*

Geoff Kuenning, *Harvey Mudd College*

Kai Li, *Princeton University*

James Mickens, *Harvard University*

Ethan L. Miller, *University of California, Santa Cruz, and Pure Storage*

Sam H. Noh, *UNIST (Ulsan National Institute of Science & Technology)*

David Pease, *IBM Research*

Daniel Peek, *Facebook*

Dan R. K. Ports, *University of Washington*

Ken Salem, *University of Waterloo*

Bianca Schroeder, *University of Toronto*

Keith A. Smith, *NetApp*

Michael Swift, *University of Wisconsin—Madison*

Nisha Talagala, *SanDisk*

Niraj Tolia, *EMC*

Joseph Tucek, *Hewlett-Packard Laboratories*

Mustafa Uysal, *Google*

Carl Waldspurger, *CloudPhysics*

Hakim Weatherspoon, *Cornell University*

Sage Weil, *Red Hat*

Brent Welch, *Google*

Theodore M. Wong, *Human Longevity, Inc.*

Gala Yadgar, *Technion—Israel Institute of Technology*

Yiying Zhang, *Purdue University*

Work-in-Progress/Posters Co-Chairs

Haryad Gunawi, *University of Chicago*

Daniel Peek, *Facebook*

Tutorial Coordinator

John Strunk, *NetApp*

Steering Committee

Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*

William J. Bolosky, *Microsoft Research*

Jason Flinn, *University of Michigan*

Greg Ganger, *Carnegie Mellon University*

Garth Gibson, *Carnegie Mellon University and Panasas, Inc.*

Casey Henderson, *USENIX Association*

Kimberly Keeton, *HP Labs*

Erik Riedel, *EMC*

Jiri Schindler, *SimpliVity*

Bianca Schroeder, *University of Toronto*

Margo Seltzer, *Harvard University and Oracle*

Keith A. Smith, *NetApp*

Eno Thereska, *Confluent and Imperial College London*

Ric Wheeler, *Red Hat*

Erez Zadok, *Stony Brook University*

Yuanyuan Zhou, *University of California, San Diego*

External Reviewers

Abutalib Aghayey
Samer Al-Kiswany
Ram Alagappan
Mona Attariyan
Ben Blum
Tyler Harter
Jun He
Choulseung Hyun
Saurabh Kadekodi
James Kelley
Jaeho Kim
Jin Kyu Kim
Eunji Lee
Tanakorn Leesatapornwongsa
Lanyue Lu
Arif Merchant
David Moulton
Sanketh Nalli
Beomseok Nam
Yuvraj Patel
Kai Ren
Thanumalayan Sankaranarayana Pillai
Priya Sehgal
Zev Weiss
Lin Xiao
Jinliang Wei
Suli Yang
Qing Zheng

14th USENIX Conference
on File and Storage Technologies (FAST '16)
February 22–25, 2016
Santa Clara, CA, USA

Tuesday, February 23, 2016

The Blueprint: File and Storage System Designs

Emotional Rescue: Reliability

They Said It Couldn't Be Done: Writing to Flash

Wednesday, February 24, 2016

Songs in the Key of Life: Key-Value Stores

Master of Puppets: Adapting Cloud and Datacenter Storage

Magical Mystery Tour: Miscellaneous

Thursday, February 25, 2016

Eliminator: Deduplication

The Unforgettable Fire: Flash and NVM

Message from the FAST ’16 Program Co-Chairs

Welcome to the 14th USENIX Conference on File and Storage Technologies. This year’s conference continues the FAST 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 flash and NVM, reliability, key-value stores, cloud and datacenter storage, and deduplication. Our authors hail from 15 countries on 3 continents and represent academia, industry, and the open-source communities. Many of the submitted papers are the fruits of a collaboration among all these communities.

FAST ’16 received 115 submissions. Of these, we selected 27 for an acceptance rate of 23%. 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, 74 papers received at least two more reviews. The Program Committee discussed 55 papers in an all-day meeting on December 4, 2015, at the University of Toronto, Toronto, Canada. We used Eddie Kohler’s superb HotCRP software to manage all stages of the review process, from submission to author notification.

As in the previous four years, we have included a category of short papers in the program. Short papers provide a vehicle for presenting research ideas that do not require a full-length paper to describe and evaluate. In judging short papers, we applied the same standards as for full-length submissions. We received 27 short paper submissions, of which we accepted 5. We were happy to see a growing number of submissions (and accepted papers) from adjacent areas such as database systems and verification.

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 ’16. We would also like to thank the attendees of FAST ’16 and 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 USENIX staff, 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. Our thanks go also to the members of the FAST Steering Committee who provided invaluable advice and feedback.

Finally, we wish to thank our Program Committee for their many hours of hard work in reviewing and discussing the submissions, some of whom traveled halfway across the world for the one-day in-person PC meeting. Together with a few external reviewers, they wrote over 496 thoughtful and meticulous reviews. HotCRP recorded over 310,000 words in reviews and comments. The reviewers’ reviews, and their thorough and conscientious deliberations at the PC meeting, contributed significantly to the quality of our decisions. Finally, we also thank several people who helped make the PC meeting run smoothly: student volunteers George Amvrosiadis, Daniel Fryer, and Ioan Stefanovici; local arrangements and administrative support from Joseph Raghubar and Regina Hui; and IT support from Tom Glinos.

We look forward to an interesting and enjoyable conference!

Angela Demke Brown, *University of Toronto*
Florentina Popovici, *Google*
FAST ’16 Program Co-Chairs

Optimizing Every Operation in a Write-Optimized File System

Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala,
Kanchan Chandnani, Pooja Deo, Zardosht Kasheff*, Leif Walsh**, Michael A. Bender,
Martin Farach-Colton†, Rob Johnson, Bradley C. Kuszmaul‡, and Donald E. Porter

*Stony Brook University, *Facebook, **Two Sigma, †Rutgers University,
and ‡Massachusetts Institute of Technology*

Abstract

File systems that employ write-optimized dictionaries (WODs) can perform random-writes, metadata updates, and recursive directory traversals orders of magnitude faster than conventional file systems. However, previous WOD-based file systems have not obtained all of these performance gains without sacrificing performance on other operations, such as file deletion, file or directory renaming, or sequential writes.

Using three techniques, *late-binding journaling*, *zoning*, and *range deletion*, we show that there is no fundamental trade-off in write-optimization. These dramatic improvements can be retained while matching conventional file systems on *all* other operations.

BetrFS 0.2 delivers order-of-magnitude better performance than conventional file systems on directory scans and small random writes and matches the performance of conventional file systems on rename, delete, and sequential I/O. For example, BetrFS 0.2 performs directory scans 2.2× faster, and small random writes over two orders of magnitude faster, than the fastest conventional file system. But unlike BetrFS 0.1, it renames and deletes files commensurate with conventional file systems and performs large sequential I/O at nearly disk bandwidth. The performance benefits of these techniques extend to applications as well. BetrFS 0.2 continues to outperform conventional file systems on many applications, such as as `rsync`, `git-diff`, and `tar`, but improves `git-clone` performance by 35% over BetrFS 0.1, yielding performance comparable to other file systems.

1 Introduction

Write-Optimized Dictionaries (WODs)[1], such as Log-Structured Merge Trees (LSM-trees) [24] and B$^\varepsilon$-trees [6], are promising building blocks for managing on-disk data in a file system. Compared to conventional file systems, previous WOD-based file systems have improved the performance of random writes [7, 12, 30], metadata updates [7, 12, 25, 30], and recursive directory traversals [7, 12] by orders of magnitude.

However, previous WOD-based file systems have not obtained all three of these performance gains without sacrificing performance on some other operations. For example, TokuFS [7] and BetrFS [12] have slow file deletions, renames, and sequential file writes. Directory traversals in KVFS [30] and TableFS [25] are essentially no faster than conventional file systems. TableFS stores large files in the underlying ext4 file system, and hence offers no performance gain for random file writes.

This paper shows that a WOD-based file system can retain performance improvements to metadata updates, small random writes, and recursive directory traversals—sometimes by orders of magnitude—while matching conventional file systems on other operations.

We identify three techniques to address fundamental performance issues for WOD-based file systems and implement them in BetrFS [11, 12]. We call the resulting system BetrFS 0.2 and the baseline BetrFS 0.1. Although we implement these ideas in BetrFS, we expect they will improve any WOD-based file system and possibly have more general application.

First, we use a *late-binding journal* to perform large sequential writes at disk bandwidth while maintaining the strong recovery semantics of full-data journaling. BetrFS 0.1 provides full-data journaling, but halves system throughput for large writes because all data is written at least twice. Our late-binding journal adapts an approach used by no-overwrite file systems, such as zfs [4] and btrfs [26], which writes data into free space only once. A particular challenge in adapting this technique to a B$^\varepsilon$-tree is balancing crash consistency of data against sufficient I/O scheduling flexibility to avoid reintroducing large, duplicate writes in B$^\varepsilon$-tree message flushing.

Second, BetrFS 0.2 introduces a tunable directory tree partitioning technique, called *zoning*, that balances the tension between fast recursive directory traversals and fast file and directory renames. Fast traversals require

[1] The terms Write-Optimized Index (WOI), Write-Optimized Dictionary (WOD), and Write-Optimized Data Structure (WODS) can be used interchangeably.

co-locating related items on disk, but to maintain this locality, renames must physically move data. Fast renames can be implemented by updating a few metadata pointers, but this can scatter a directory's contents across the disk. Zoning yields most of the benefits of both designs. BetrFS 0.2 traverses directories at near disk bandwidth and renames at speeds comparable to inode-based systems.

Finally, BetrFS 0.2 contributes a new *range delete* WOD operation that accelerates unlinks, sequential writes, renames, and zoning. BetrFS 0.2 uses range deletes to tell the WOD when large swaths of data are no longer needed. Range deletes enable further optimizations, such as avoiding the read-and-merge of stale data, that would otherwise be difficult or impossible.

With these enhancements, BetrFS 0.2 can roughly match other local file systems on Linux. In some cases, it is much faster than other file systems or provides stronger guarantees at a comparable cost. In a few cases, it is slower, but within a reasonable margin.

The contributions of this paper are:

- A **late-binding journal** for large writes to a message-oriented WOD. BetrFS 0.2 writes large files at 96MB/s, compared to 28MB/s in BetrFS 0.1.
- A **zone-tree schema** and analytical framework for reasoning about trade-offs between locality in directory traversals and indirection for fast file and directory renames. We identify a point that preserves most of the scan performance of the original BetrFS and supports renames competitive with conventional file systems for most file and directory sizes. The highest rename overhead is bound at 3.8× slower than the ext4.
- A **range delete** primitive, which enables WOD-internal optimizations for file deletion, and also avoids costly reads and merges of dead tree nodes. With range delete, BetrFS 0.2 can unlink a 1 GB file in 11ms, compared to over a minute on BetrFS 0.1 and 110ms on ext4.
- A thorough evaluation of these optimizations and their impact on real-world applications.

Thus, BetrFS 0.2 demonstrates that a WOD can improve file-system performance on random writes, metadata updates, and directory traversals by orders of magnitude without sacrificing performance on other file-system operations.

2 Background

This section gives the background necessary to understand and analyze the performance of WOD-based file systems, with a focus on B^ε-trees and BetrFS. See Bender et al. [3] for a more comprehensive tutorial.

2.1 Write-Optimized Dictionaries

WODs include Log-Structured Merge Trees (LSM-trees) [24] and their variants [29, 30, 37], B^ε-trees [6], xDicts [5], and cache-oblivious lookahead arrays (COLAs) [2, 28]. WODs provide a key-value interface supporting insert, query, delete, and range-query operations.

The WOD interface is similar to that of a B-tree, but the performance profile is different:

- WODs can perform inserts of random keys orders of magnitude faster than B-trees. On a rotating disk, a B-tree can perform only a couple of hundred inserts per second in the worst case, whereas a WOD can perform many tens of thousands.
- In WODs, a delete is implemented by inserting a tombstone message, which is extremely fast.
- Some WODs, such as B^ε-trees, can perform point queries as fast as a B-tree. B^ε-trees (but not LSM-trees) offer a provably optimal combination of query and insert performance.
- WODs perform range queries at nearly disk bandwidth. Because a WOD can use nodes over a megabyte in size, a scan requires less than one disk seek per MB of data and hence is bandwidth bound.

The key idea behind write optimization is deferring and batching small, random writes. A B^ε-tree logs insertions or deletions as *messages* at the root of the tree, and only flushes messages down a level in the tree when enough messages have accrued to offset the cost of accessing the child. As a result, a single message may be written to disk multiple times. Since each message is always written as part of a larger batch, the amortized cost for each insert is typically much less than one I/O. In comparison, writing a random element to a large B-tree requires a minimum of one I/O.

Most production-quality WODs are engineered for use in databases, not in file systems, and are therefore designed with different performance requirements. For example, the open-source WOD implementation underlying BetrFS is a port of TokuDB[2] into the Linux kernel [32]. TokuDB logs all inserted keys and values to support transactions, limiting the write bandwidth to at most half of disk bandwidth. As a result, BetrFS 0.1 provides full-data journaling, albeit at a cost to large sequential writes.

Caching and recovery. We now summarize relevant logging and cache-management features of TokuDB.

TokuDB updates B^ε-tree nodes using redirect on write [8]. In other words, each time a dirty node is written to disk, the node is placed at a new location. Recovery is based on periodic, stable checkpoints of the tree. Between checkpoints, a write-ahead, logical log tracks

[2] TokuDB implements Fractal Tree indexes [2], a B^ε-tree variant.

all tree updates and can be replayed against the last stable checkpoint for recovery. This log is buffered in memory, and is made durable at least once every second.

This scheme of checkpoint and write-ahead log allows the B$^\varepsilon$-tree to cache dirty nodes in memory and write them back in any order, as long as a consistent version of the tree is written to disk at checkpoint time. After each checkpoint, old checkpoints, logs, and unreachable nodes are garbage collected.

Caching dirty nodes improves insertion performance because TokuDB can often avoid writing internal tree nodes to disk. When a new message is inserted into the tree, it can immediately be moved down the tree as far as possible without dirtying any new nodes. If the message is part of a long stream of sequential inserts, then the *entire* root-to-leaf path is likely to be dirty, and the message can go straight to its leaf. This caching, combined with write-ahead logging, explains why large sequential writes in BetrFS 0.1 realize at most half[3] of the disk's bandwidth: most messages are written once to the log and only once to a leaf. Section 3 describes a late-binding journal, which lets BetrFS 0.2 write large data values only once, without sacrificing the crash consistency of data.

Message propagation. As the buffer in an internal B$^\varepsilon$-tree node fills up, the B$^\varepsilon$-tree estimates which child or children would receive enough messages to amortize the cost of flushing these messages down one level. Messages are kept logically consistent within a node buffer, stored in commit order. Even if messages are physically applied to leaves at different times, any read applies all matching buffered messages between the root and leaf in commit order. Section 5 introduces a "rangecast" message type, which can propagate to multiple children.

2.2 BetrFS

BetrFS stores all file system data—both metadata and file contents—in B$^\varepsilon$-trees [12]. BetrFS uses two B$^\varepsilon$-trees: a metadata index and a data index. The metadata index maps full paths to the corresponding `struct stat` information. The data index maps (path, block-number) pairs to the contents of the specified file block.

Indirection. A traditional file system uses indirection, e.g., inode numbers, to implement renames efficiently with a single pointer swap. This indirection can hurt directory traversals because, in the degenerate case, there could be one seek per file.

The BetrFS 0.1 full-path-based schema instead optimizes directory traversals at the expense of renaming large files and directories. A recursive directory traversal maps directly to a range query in the underlying B$^\varepsilon$-tree, which can run at nearly disk bandwidth. On the other hand, renames in BetrFS 0.1 must move all data from the old keys to new keys, which can become expensive for large files and directories. Section 4 presents schema changes that enable BetrFS 0.2 to perform recursive directory traversals at nearly disk bandwidth and renames at speeds comparable to inode-based file systems.

Indexing data and metadata by full path also harms deletion performance, as each block of a large file must be individually removed. The sheer volume of these delete messages in BetrFS 0.1 leads to orders-of-magnitude worse unlink times for large files. Section 5 describes our new "rangecast delete" primitive for implementing efficient file deletion in BetrFS 0.2.

Consistency. In BetrFS, file writes and metadata changes are first recorded in the kernel's generic VFS data structures. The VFS may cache dirty data and metadata for up to 5 seconds before writing it back to the underlying file system, which BetrFS converts to B$^\varepsilon$-tree operations. Thus BetrFS can lose at most 6 seconds of data during a crash—5 seconds from the VFS layer and 1 second from the B$^\varepsilon$-tree log buffer. `fsync` in BetrFS first writes all dirty data and metadata associated with the inode, then writes the entire log buffer to disk.

3 Avoiding Duplicate Writes

This section discusses *late-binding journaling*, a technique for delivering the sequential-write performance of metadata-only journaling while guaranteeing full-data-journaling semantics.

BetrFS 0.1 is unable to match the sequential-write performance of conventional file systems because it writes all data at least twice: once to a write-ahead log and at least once to the B$^\varepsilon$-tree. As our experiments in Section 7 show, BetrFS 0.1 on a commodity disk performs large sequential writes at 28MB/s, whereas other local file systems perform large sequential writes at 78–106MB/s—utilizing nearly all of the hard drive's 125 MB/s of bandwidth. The extra write for logging does not significantly affect the performance of small random writes, since they are likely to be written to disk several times as they move down the B$^\varepsilon$-tree in batches. However, large sequential writes are likely to go directly to tree leaves, as explained in Section 2.1. Since they would otherwise be written only once in the B$^\varepsilon$-tree, logging halves BetrFS 0.1 sequential write bandwidth. Similar overheads are well-known for update-in-place file systems, such as ext4, which defaults to metadata-only journaling as a result.

Popular no-overwrite file systems address journal write amplification with indirection. For small values,

[3] TokuDB had a performance bug that further reduced BetrFS 0.1's sequential write performance to at most 1/3rd of disk bandwidth. See Section 6 for details.

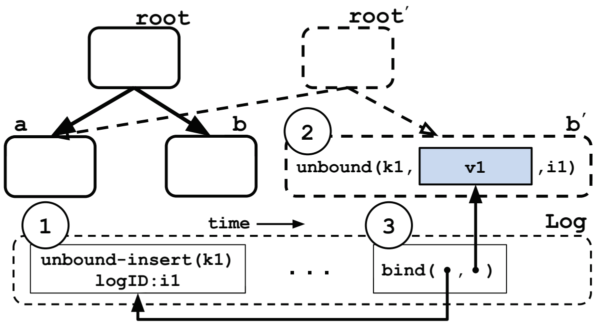


Figure 1: Late-binding journaling in a B$^\varepsilon$-tree.

zfs embeds data directly in a log entry. For large values, it writes data to disk redirect-on-write, and stores a pointer in the log [21]. This gives zfs fast durability for small writes by flushing the log, avoids the overhead of writing large values twice, and retains the recovery semantics of data journaling. On the other hand, btrfs [26] uses indirection for all writes, regardless of size. It writes data to newly-allocated blocks, and records those writes with pointers in its journal.

In the rest of this section, we explain how we integrate indirection for large writes into the BetrFS recovery mechanism, and we discuss the challenges posed by the message-oriented design of the B$^\varepsilon$-tree.

BetrFS on-disk structures. The BetrFS B$^\varepsilon$-tree implementation writes B$^\varepsilon$-tree nodes to disk using redirect-on-write and maintains a logical write-ahead redo log. Each insert or delete message is first recorded in the log and then inserted into the tree's in-memory nodes. Each entry in the log specifies the operation (insert or delete) and the relevant keys and values.

Crash consistency is implemented by periodically checkpointing the B$^\varepsilon$-tree and by logging operations between checkpoints. An operation is durable once its log entry is on disk. At each checkpoint, all dirty nodes are written to ensure that a complete and consistent B$^\varepsilon$-tree snapshot is on disk, and the log is discarded. For instance, after checkpoint i completes, there is a single B$^\varepsilon$-tree, T_i, and an empty log. Any blocks that are not reachable in T_i can be garbage collected and reallocated.

Between checkpoints i and $i+1$, all operations are logged in Log_{i+1}. If the system crashes at any time between the completion of checkpoint i and checkpoint $i+1$, it will resume from tree T_i and replay Log_{i+1}.

Late-binding journal. BetrFS 0.2 handles large messages, or large runs of consecutive messages, as follows and illustrated in Figure 1:

- A special *unbound log entry* is appended to the in-memory log buffer ①. An unbound log entry specifies an operation and a key, but not a value. These messages record the insert's logical order.
- A special *unbound message* is inserted into the B$^\varepsilon$-tree ②. An unbound message contains the key, value, and log entry ID of its corresponding unbound log entry. Unbound messages move down the tree like any other message.
- To make the log durable, all nodes containing unbound messages are first written to disk. As part of writing the node to disk, each unbound message is converted to a normal insert message (non-leaf node) or a normal key-value pair (leaf node). After an unbound message in a node is written to disk, a *binding log entry* is appended to the in-memory log buffer ③. Each binding log entry contains the log entry ID from the unbound message and the physical disk address of the node. Once all inserts in the in-memory log buffer are bound, the in-memory log buffer is written to disk.
- Node write-backs are handled similarly: when a node containing an unbound message is written to disk as part of a cache eviction, checkpoint, or for any other reason, binding entries are appended to the in-memory log buffer for all the unbound messages in the node, and the messages in the node are marked as bound.

The system can make logged operations durable at any time by writing out all the tree nodes that contain unbound messages and then flushing the log to disk. It is an invariant that all unbound inserts in the on-disk log will have matching binding log entries. Thus, recovery can always proceed to the end of the log.

The on-disk format does not change for an unbound insert: unbound messages exist only in memory.

The late-binding journal accelerates large messages. A negligible amount of data is written to the log, but a tree node is forced to be written to disk. If the amount of data to be written to a given tree node is equivalent to the size of the node, this reduces the bandwidth cost by half.

In the case where one or more inserts only account for a small fraction of the node, logging the values is preferable to unbound inserts. The issue is that an unbound insert can prematurely force the node to disk (at a log flush, rather than the next checkpoint), losing opportunities to batch more small modifications. Writing a node that is mostly unchanged wastes bandwidth. Thus, BetrFS 0.2 uses unbound inserts only when writing at least 1MB of consecutive pages to disk.

Crash Recovery. Late-binding requires two passes over the log during recovery: one to identify nodes containing unbound inserts, and a second to replay the log.

The core issue is that each checkpoint only records the on-disk nodes in use for that checkpoint. In BetrFS 0.2, nodes referenced by a binding log entry are not marked as allocated in the checkpoint's allocation table. Thus, the first pass is needed to update the allocation table to include all nodes referenced by binding log messages. The second pass replays the logical entries in the log. After the next checkpoint, the log is discarded, and the reference counts on all nodes referenced by the log are decremented. Any nodes whose reference count hits zero (i.e. because they are no longer referenced by other nodes in the tree) are garbage collected at that time.

Implementation. BetrFS 0.2 guarantees consistent recovery up until the last log flush or checkpoint. By default, a log flush is triggered on a `sync` operation, every second, or when the 32 MB log buffer fills up. Flushing a log buffer with unbound log entries also requires searching the in-memory tree nodes for nodes containing unbound messages, in order to first write these nodes to disk. Thus, BetrFS 0.2 also reserves enough space at the end of the log buffer for the binding log messages. In practice, the log-flushing interval is long enough that most unbound inserts are written to disk before the log flush, minimizing the delay for a log write.

Additional optimizations. Section 5 explains some optimizations where logically obviated operations can be discarded as part of flushing messages down one level of the tree. One example is when a key is inserted and then deleted; if the insert and delete are in the same message buffer, the insert can be dropped, rather than flushed to the next level. In the case of unbound inserts, we allow a delete to remove an unbound insert before the value is written to disk under the following conditions: (1) all transactions involving the unbound key-value pair have committed, (2) the delete transaction has committed, and (3) the log has not yet been flushed. If these conditions are met, the file system can be consistently recovered without this unbound value. In this situation, BetrFS 0.2 binds obviated inserts to a special NULL node, and drops the insert message from the B$^{\varepsilon}$-tree.

4 Balancing Search and Rename

In this section, we argue that there is a design trade-off between the performance of renames and recursive directory scans. We present an algorithmic framework for picking a point along this trade-off curve.

Conventional file systems support fast renames at the expense of slow recursive directory traversals. Each file and directory is assigned its own inode, and names in a directory are commonly mapped to inodes with pointers. Renaming a file or directory can be very efficient, requiring only creation and deletion of a pointer to an inode, and a constant number of I/Os. However, searching files or subdirectories within a directory requires traversing all these pointers. When the inodes under a directory are not stored together on disk, for instance because of renames, then each pointer traversal can require a disk seek, severely limiting the speed of the traversal.

BetrFS 0.1 and TokuFS are at the other extreme. They index every directory, file, and file block by its full path in the file system. The sort order on paths guarantees that all the entries beneath a directory are stored contiguously in logical order within nodes of the B$^{\varepsilon}$-tree, enabling fast scans over entire subtrees of the directory hierarchy. Renaming a file or directory, however, requires physically moving every file, directory, and block to a new location.

This trade-off is common in file system design. Intermediate points between these extremes are possible, such as embedding inodes in directories but not moving data blocks of renamed files. Fast directory traversals require on-disk locality, whereas renames must issue only a small number of I/Os to be fast.

BetrFS 0.2's schema makes this trade-off parameterizable and tunable by partitioning the directory hierarchy into connected regions, which we call *zones*. Figure 2a shows how files and directories within subtrees are collected into zones in BetrFS 0.2. Each zone has a unique zone-ID, which is analogous to an inode number in a traditional file system. Each zone contains either a single file or has a single root directory, which we call the root of the zone. Files and directories are identified by their zone-ID and their relative path within the zone.

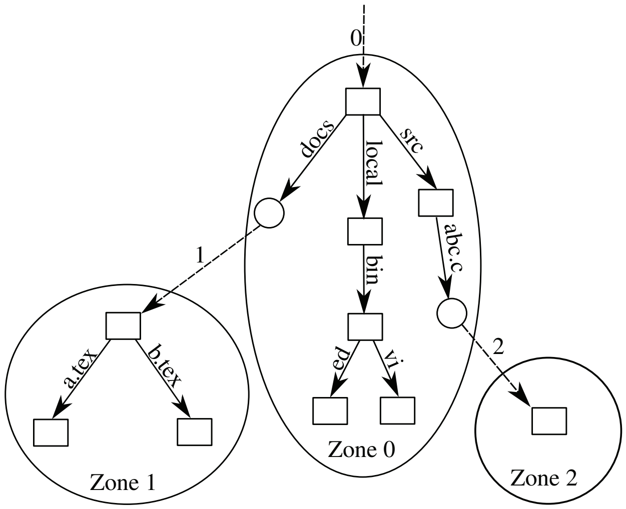
Directories and files within a zone are stored together, enabling fast scans within that zone. Crossing a zone boundary potentially requires a seek to a different part of the tree. Renaming a file under a zone root moves the data, whereas renaming a large file or directory (a zone root) requires only changing a pointer.

Zoning supports a spectrum of trade-off points between the two extremes described above. When zones are restricted to size 1, the BetrFS 0.2 schema is equivalent to an inode-based schema. If we set the zone size bound to infinity (∞), then BetrFS 0.2's schema is equivalent to BetrFS 0.1's schema. At an intermediate setting, BetrFS 0.2 can balance the performance of directory scans and renames.

The default zone size in BetrFS 0.2 is 512 KiB. Intuitively, moving a very small file is sufficiently inexpensive that indirection would save little, especially in a WOD. On the other extreme, once a file system is reading several MB between each seek, the dominant cost is transfer time, not seeking. Thus, one would expect the best zone size to be between tens of KB and a few MB. We also note that this trade-off is somewhat implementation-dependent: the more efficiently a file system can move a set of keys and values, the larger a zone can be without harming rename performance. Section 7 empirically evaluates these trade-offs.

As an effect of zoning, BetrFS 0.2 supports hard links by placing a file with more than 1 link into its own zone.

Metadata and data indexes. The BetrFS 0.2 meta-



(a) An example zone tree in BetrFS 0.2.

Metadata Index

(0,”/”)	→	stat info for “/”
(0,”/docs”)	→	zone 1
(0,”/local”)	→	stat info for “/local”
(0,”/src”)	→	stat info for “/src”
(0,”/local/bin”)	→	stat info for “/local/bin”
(0,”/local/bin/ed”)	→	stat info for “/local/bin/ed”
(0,”/local/bin/vi”)	→	stat info for “/local/bin/vi”
(0,”/src/abc.c”)	→	zone 2
(1,”/”)	→	stat info for “/docs”
(1,”/a.tex”)	→	stat info for “/docs/a.tex”
(1,”/b.tex”)	→	stat info for “/docs/b.tex”
(2,”/”)	→	stat info for “/src/abc.c”

Data Index

(0,“/local/bin/ed”, i)	→	block i of “/local/bin/ed”
(0,“/local/bin/vi”, i)	→	block i of “/local/bin/vi”
(1,“/a.tex”, i)	→	block i of “/docs/a.tex”
(1,“/b.tex”, i)	→	block i of “/docs/b.tex”
(2,“/”, i)	→	block i of “/src/abc.c”

(b) Example metadata and data indices in BetrFS 0.2.

Figure 2: Pictorial and schema illustrations of zone trees in BetrFS 0.2.

data index maps (zone-ID, relative-path) keys to metadata about a file or directory, as shown in Figure 2b. For a file or directory in the same zone, the metadata includes the typical contents of a stat structure, such as owner, modification time, and permissions. For instance, in zone 0, path “/local” maps onto the stat info for this directory. If this key (i.e., relative path within the zone) maps onto a different zone, then the metadata index maps onto the ID of that zone. For instance, in zone 0, path “/docs” maps onto zone-ID 1, which is the root of that zone.

The data index maps (zone-ID, relative-path, block-number) to the content of the specified file block.

Path sorting order. BetrFS 0.2 sorts keys by zone-ID first, and then by their relative path. Since all the items in a zone will be stored consecutively in this sort order, recursive directory scans can visit all the entries within a zone efficiently. Within a zone, entries are sorted by path in a “depth-first-with-children” order, as illustrated in Figure 2b. This sort order ensures that all the entries beneath a directory are stored logically contiguously in the underlying key-value store, followed by recursive listings of the subdirectories of that directory. Thus an application that performs `readdir` on a directory and then recursively scans its sub-directories in the order returned by `readdir` will effectively perform range queries on that zone and each of the zones beneath it.

Rename. Renaming a file or directory that is the root of its zone requires simply inserting a reference to its zone at its new location and deleting the old reference. So, for example, renaming “/src/abc.c” to “/docs/def.c” in Figure 2 requires deleting key (0, “/src/abc.c”) from the metadata index and inserting the mapping (1, “/def.c”) → Zone 2.

Renaming a file or directory that is not the root of its zone requires copying the contents of that file or directory to its new location. So, for example, renaming “/local/bin” to “/docs/tools” requires (1) deleting all the keys of the form (0, “/local/bin/*p*”) in the metadata index, (2) reinserting them as keys of the form (1, “/tools/*p*”), (3) deleting all keys of the form (0, “/local/bin/*p*”, *i*) from the data index, and (4) reinserting them as keys of the form (1, “/tools/*p*”, *i*). Note that renaming a directory never requires recursively moving into a child zone. Thus, by bounding the size of the directory subtree within a single zone, we also bound the amount of work required to perform a rename.

Splitting and merging. To maintain a consistent rename and scan performance trade-off throughout system lifetime, zones must be split and merged so that the following two invariants are upheld:

ZoneMin: Each zone has size at least C_0.

ZoneMax: Each directory that is not the root of its zone has size at most C_1.

The ZoneMin invariant ensures that recursive directory traversals will be able to scan through at least C_0 consecutive bytes in the key-value store before initiating a scan of another zone, which may require a disk seek. The ZoneMax invariant ensures that no directory rename will require moving more than C_1 bytes.

The BetrFS 0.2 design upholds these invariants as follows. Each inode maintains two counters to record the number of data and metadata entries in its subtree.

Whenever a data or metadata entry is added or removed, BetrFS 0.2 recursively updates counters from the corresponding file or directory up to its zone root. If either of a file or directory's counters exceed C_1, BetrFS 0.2 creates a new zone for the entries in that file or directory. When a zone size falls below C_0, that zone is merged with its parent. BetrFS 0.2 avoids cascading splits and merges by merging a zone with its parent only when doing so would not cause the parent to split. To avoid unnecessary merges during a large directory deletion, BetrFS 0.2 defers merging until writing back dirty inodes.

We can tune the trade-off between rename and directory traversal performance by adjusting C_0 and C_1. Larger C_0 will improve recursive directory traversals. However, increasing C_0 beyond the block size of the underlying data structure will have diminishing returns, since the system will have to seek from block to block during the scan of a single zone. Smaller C_1 will improve rename performance. All objects larger than C_1 can be renamed in a constant number of I/Os, and the worst-case rename requires only C_1 bytes be moved. In the current implementation, $C_0 = C_1 = 512$ KiB.

The zone schema enables BetrFS 0.2 to support a spectrum of trade-offs between rename performance and directory traversal performance. We explore these trade-offs empirically in Section 7.

5 Efficient Range Deletion

This section explains how BetrFS 0.2 obtains nearly-flat deletion times by introducing a new *rangecast* message type to the B$^{\varepsilon}$-tree, and implementing several B$^{\varepsilon}$-tree-internal optimizations using this new message type.

BetrFS 0.1 file and directory deletion performance is linear in the amount of data being deleted. Although this is true to some extent in any file system, as the freed disk space will be linear in the file size, the slope for BetrFS 0.1 is alarming. For instance, unlinking a 4GB file takes 5 minutes on BetrFS 0.1!

Two underlying issues are the sheer volume of delete messages that must be inserted into the B$^{\varepsilon}$-tree and missed optimizations in the B$^{\varepsilon}$-tree implementation. Because the B$^{\varepsilon}$-tree implementation does not bake in any semantics about the schema, the B$^{\varepsilon}$-tree cannot infer that two keys are adjacent in the keyspace. Without hints from the file system, a B$^{\varepsilon}$-tree cannot optimize for the common case of deleting large, contiguous key ranges.

5.1 Rangecast Messages

In order to support deletion of a key range in a single message, we added a *rangecast* message type to the B$^{\varepsilon}$-tree implementation. In the baseline B$^{\varepsilon}$-tree implementation, updates of various forms (e.g., insert and delete) are encoded as messages addressed to a single key, which, as explained in §2, are flushed down the path from root-to-leaf. A rangecast message can be addressed to a contiguous range of keys, specified by the beginning and ending keys, inclusive. These beginning and ending keys need not exist, and the range can be sparse; the message will be applied to any keys in the range that do exist. We have currently added rangecast delete messages, but we can envision range insert and upsert [12] being useful.

Rangecast message propagation. When single-key messages are propagated from a parent to a child, they are simply inserted into the child's buffer space in logical order (or in key order when applied to a leaf). Rangecast message propagation is similar to regular message propagation, with two differences.

First, rangecast messages may be applied to multiple children at different times. When a rangecast message is flushed to a child, the propagation function must check whether the range spans multiple children. If so, the rangecast message is transparently split and copied for each child, with appropriate subsets of the original range. If a rangecast message covers multiple children of a node, the rangecast message can be split and applied to each child at different points in time—most commonly, deferring until there are enough messages for that child to amortize the flushing cost. As messages propagate down the tree, they are stored and applied to leaves in the same commit order. Thus, any updates to a key or reinsertions of a deleted key maintain a global serial order, even if a rangecast spans multiple nodes.

Second, when a rangecast delete is flushed to a leaf, it may remove multiple key/value pairs, or even an entire leaf. Because `unlink` uses rangecast delete, all of the data blocks for a file are freed atomically with respect to a crash.

Query. A B$^{\varepsilon}$-tree query must apply all pending modifications in node buffers to the relevant key(s). Applying these modifications is efficient because all relevant messages will be in a node's buffer on the root-to-leaf search path. Rangecast messages maintain this invariant.

Each B$^{\varepsilon}$-tree node maintains a FIFO queue of pending messages, and, for single-key messages, a balanced binary tree sorted by the messages' keys. For rangecast messages, our current prototype checks a simple list of rangecast messages and interleaves the messages with single-key messages based on commit order. This search costs linear in the number of rangecast messages. A faster implementation would store the rangecast messages in each node using an interval tree, enabling it to find all the rangecast messages relevant to a query in $O(k + \log n)$ time, where n is number of rangecast messages in the node and k is the number of those messages relevant to the current query.

Rangecast `unlink` **and** `truncate`**.** In the BetrFS 0.2 schema, 4KB data blocks are keyed by a concatenated tuple of zone ID, relative path, and block number. Unlinking a file involves one delete message to remove the file from the metadata index and, in the same B$^{\varepsilon}$-tree-level transaction, a rangecast delete to remove all of the blocks. Deleting all data blocks in a file is simply encoded by using the same prefix, but from blocks 0 to infinity. Truncating a file works the same way, but can start with a block number other than zero, and does not remove the metadata key.

5.2 B$^{\varepsilon}$-Tree-Internal Optimizations

The ability to group a large range of deletion messages not only reduces the number of total delete messages required to remove a file, but it also creates new opportunities for B$^{\varepsilon}$-tree-internal optimizations.

Leaf Pruning. When a B$^{\varepsilon}$-tree flushes data from one level to the next, it must first read the child, merge the incoming data, and rewrite the child. In the case of a large, sequential write, a large range of obviated data may be read from disk, only to be overwritten. In the case of BetrFS 0.1, unnecessary reads make overwriting a 10 GB file 30–63 MB/s slower than the first write of the file.

The leaf pruning optimization identifies when an entire leaf is obviated by a range delete, and elides reading the leaf from disk. When a large range of consecutive keys and values are inserted, such as overwriting a large file region, BetrFS 0.2 includes a range delete for the key range in the same transaction. This range delete message is necessary, as the B$^{\varepsilon}$-tree cannot infer that the range of the inserted keys are contiguous; the range delete communicates information about the keyspace. On flushing messages to a child, the B$^{\varepsilon}$-tree can detect when a range delete encompasses the child's keyspace. BetrFS 0.2 uses transactions inside the B$^{\varepsilon}$-tree implementation to ensure that the removal and overwrite are atomic: at no point can a crash lose both the old and new contents of the modified blocks. Stale leaf nodes are reclaimed as part of normal B$^{\varepsilon}$-tree garbage collection.

Thus, this leaf pruning optimization avoids expensive reads when a large file is being overwritten. This optimization is both essential to sequential I/O performance and possible only with rangecast delete.

Pac-Man. A rangecast delete can also obviate a significant number of buffered messages. For instance, if a user creates a large file and immediately deletes the file, the B$^{\varepsilon}$-tree may include many obviated insert messages that are no longer profitable to propagate to the leaves.

BetrFS 0.2 adds an optimization to message flushing, where a rangecast delete message can devour obviated messages ahead of it in the commit sequence. We call this optimization "Pac-Man", in homage to the arcade game character known for devouring ghosts. This optimization further reduces background work in the tree, eliminating "dead" messages before they reach a leaf.

6 Optimized Stacking

BetrFS has a stacked file system design [12]; B$^{\varepsilon}$-tree nodes and the journal are stored as files on an ext4 file system. BetrFS 0.2 corrects two points where BetrFS 0.1 was using the underlying ext4 file system suboptimally.

First, in order to ensure that nodes are physically placed together, TokuDB writes zeros into the node files to force space allocation in larger extents. For sequential writes to a new FS, BetrFS 0.1 zeros these nodes and then immediately overwrites the nodes with file contents, wasting up to a third of the disk's bandwidth. We replaced this with the newer `fallocate` API, which can physically allocate space but logically zero the contents.

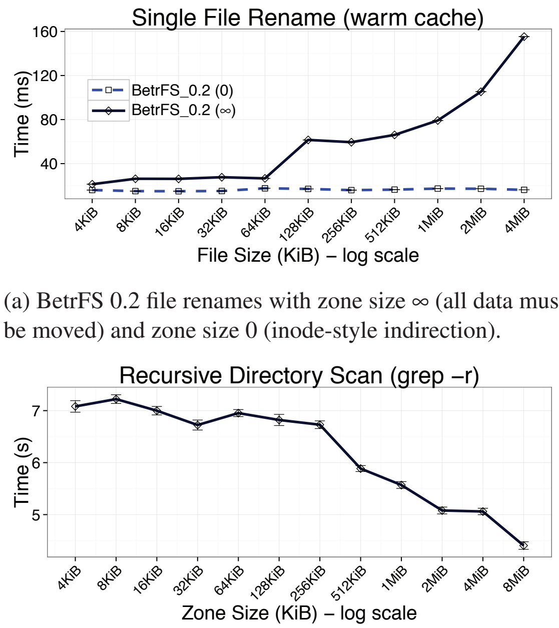
Second, the I/O to flush the BetrFS journal file was being amplified by the ext4 journal. Each BetrFS log flush appended to a file on ext4, which required updating the file size and allocation. BetrFS 0.2 reduces this overhead by pre-allocating space for the journal file and using `fdatasync`.

7 Evaluation

Our evaluation targets the following questions:

- How does one choose the zone size?
- Does BetrFS 0.2 perform comparably to other file systems on the worst cases for BetrFS 0.1?
- Does BetrFS 0.2 perform comparably to BetrFS 0.1 on the best cases for BetrFS 0.1?
- How do BetrFS 0.2 optimizations impact application performance? Is this performance comparable to other file systems, and as good or better than BetrFS 0.1?
- What are the costs of background work in BetrFS 0.2?

All experimental results were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, and a 500 GB, 7200 RPM ATA disk, with a 4096-byte block size. Each file system's block size is 4096 bytes. The system ran Ubuntu 13.10, 64-bit, with Linux kernel version 3.11.10. Each experiment is compared with several file systems, including BetrFS 0.1 [12], btrfs [26], ext4 [20], XFS [31], and zfs [4]. We use the versions of XFS, btrfs, ext4 that are part of the 3.11.10 kernel, and zfs 0.6.3, downloaded from `www.zfsonlinux.org`. The disk was divided into 2 partitions roughly 240 GB each; one for the root FS and the other for experiments. We use default recommended file system settings unless otherwise noted. Lazy inode table and journal initialization were turned off on ext4. Each



(a) BetrFS 0.2 file renames with zone size ∞ (all data must be moved) and zone size 0 (inode-style indirection).

(b) Recursive scans of the Linux 3.11.10 source for "cpu_to_be64" with different BetrFS 0.2 zone sizes.

Figure 3: The impact of zone size on rename and scan performance. Lower is better.

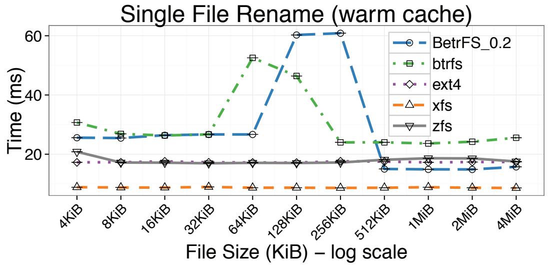


Figure 4: Time to rename single files. Lower is better.

experiment was run a minimum of 4 times. Error bars and ± ranges denote 95% confidence intervals. Unless noted, all benchmarks are cold-cache tests.

7.1 Choosing a Zone Size

This subsection quantifies the impact of zone size on rename and scan performance.

A good zone size limits the worst-case costs of rename but maintains data locality for fast directory scans. Figure 3a shows the average cost to rename a file and fsync the parent directory, over 100 iterations, plotted as a function of size. We show BetrFS 0.2 with an infinite zone size (no zones are created—rename moves all file contents) and 0 (every file is in its own zone—rename is a pointer swap). Once a file is in its own zone, the performance is comparable to most other file systems (16ms on BetrFS 0.2 compared to 17ms on ext4). This is balanced against Figure 3b, which shows grep performance versus zone size. As predicted in Section 4 directory-traversal performance improves as the zone size increases.

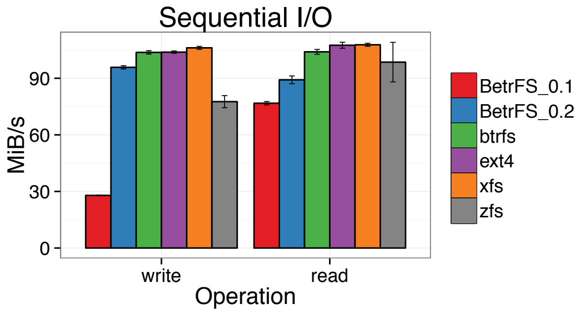


Figure 5: Large file I/O performance. We sequentially read and write a 10GiB file. Higher is better.

We select a default zone size of 512 KiB, which enforces a reasonable bound on worst case rename (compared to an unbounded BetrFS 0.1 worst case), and keeps search performance within 25% of the asymptote. Figure 4 compares BetrFS 0.2 rename time to other file systems. Specifically, worst-case rename performance at this zone size is 66ms, 3.7× slower than the median file system's rename cost of 18ms. However, renames of files 512 KiB or larger are comparable to other file systems, and search performance is 2.2× the best baseline file system and 8× the median. We use this zone size for the rest of the evaluation.

7.2 Improving the Worst Cases

This subsection measures BetrFS 0.1's three worst cases, and shows that, for typical workloads, BetrFS 0.2 is either faster or within roughly 10% of other file systems.

Sequential Writes. Figure 5 shows the throughput to sequentially read and write a 10GiB file (more than twice the size of the machine's RAM). The optimizations described in §3 improve the sequential write throughput of BetrFS 0.2 to 96MiB/s, up from 28MiB/s in BetrFS 0.1. Except for zfs, the other file systems realize roughly 10% higher throughput. We also note that these file systems offer different crash consistency properties: ext4 and XFS only guarantee metadata recovery, whereas zfs, btrfs, and BetrFS guarantee data recovery.

The sequential read throughput of BetrFS 0.2 is improved over BetrFS 0.1 by roughly 12MiB/s, which is attributable to streamlining the code. This places BetrFS 0.2 within striking distance of other file systems.

Rename. Table 1 shows the execution time of several common directory operations on the Linux 3.11.10

File System	find	grep	mv	rm -rf
BetrFS 0.1	0.36 ± 0.0	3.95 ± 0.2	21.17 ± 0.7	46.14 ± 0.8
BetrFS 0.2	0.35 ± 0.0	5.78 ± 0.1	0.13 ± 0.0	2.37 ± 0.2
btrfs	4.84 ± 0.7	12.77 ± 2.0	0.15 ± 0.0	9.63 ± 1.4
ext4	3.51 ± 0.3	49.61 ± 1.8	0.18 ± 0.1	4.17 ± 1.3
xfs	9.01 ± 1.9	61.09 ± 4.7	0.08 ± 0.0	8.16 ± 3.1
zfs	13.71 ± 0.6	43.26 ± 1.1	0.14 ± 0.0	13.23 ± 0.7

Table 1: Time in seconds to complete directory operations on the Linux 3.11.10 source: `find` of the file “wait.c”, `grep` of the string “cpu_to_be64”, `mv` of the directory root, and `rm -rf`. Lower is better.

File System	Time (s)
BetrFS 0.1	0.48 ± 0.1
BetrFS 0.2	0.32 ± 0.0
btrfs	104.18 ± 0.3
ext4	111.20 ± 0.4
xfs	111.03 ± 0.4
zfs	131.86 ± 12.6

Table 2: Time to perform 10,000 4-byte overwrites on a 10 GiB file. Lower is better.

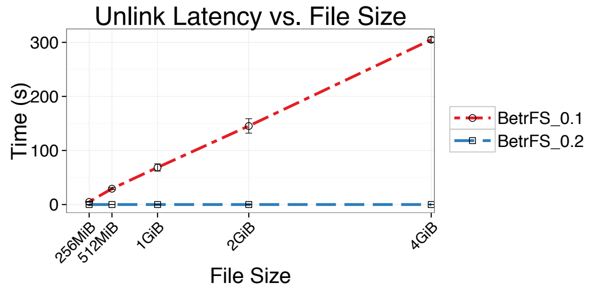


Figure 6: Unlink latency by file size. Lower is better.

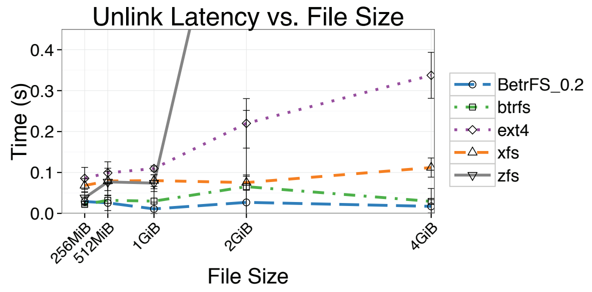


Figure 7: Unlink latency by file size. Lower is better.

source tree. The `rename` test renames the entire source tree. BetrFS 0.1 directory rename is two orders of magnitude slower than any other file system, whereas BetrFS 0.2 is faster than every other file system except XFS. By partitioning the directory hierarchy into zones, BetrFS 0.2 ensures that the cost of a rename is comparable to other file systems.

Unlink. Table 1 also includes the time to recursively delete the Linux source tree. Again, whereas BetrFS 0.1 is an order of magnitude slower than any other file system, BetrFS 0.2 is *faster*. We attribute this improvement to BetrFS 0.2's fast directory traversals and to the effectiveness of range deletion.

We also measured the latency of unlinking files of increasing size. Due to scale, we contrast BetrFS 0.1 with BetrFS 0.2 in Figure 6, and we compare BetrFS 0.2 with other file systems in Figure 7. In BetrFS 0.1, the cost to delete a file scales linearly with the file size. Figure 7 shows that BetrFS 0.2 delete latency is not sensitive to file size. Measurements show that zfs performance is considerably slower and noisier; we suspect that this variance is attributable to unlink incurring amortized housekeeping work.

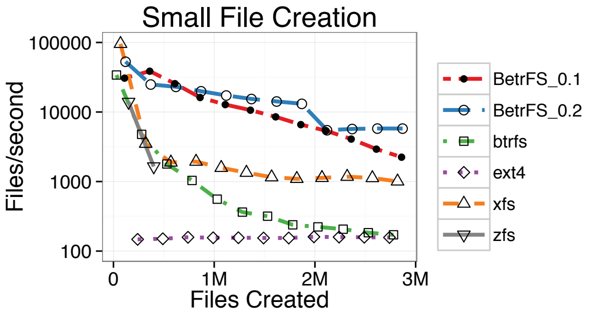


Figure 8: Sustained file creation for 3 million 200-byte files, using 4 threads. Higher is better, y-axis is log scale.

7.3 Maintaining the Best Cases

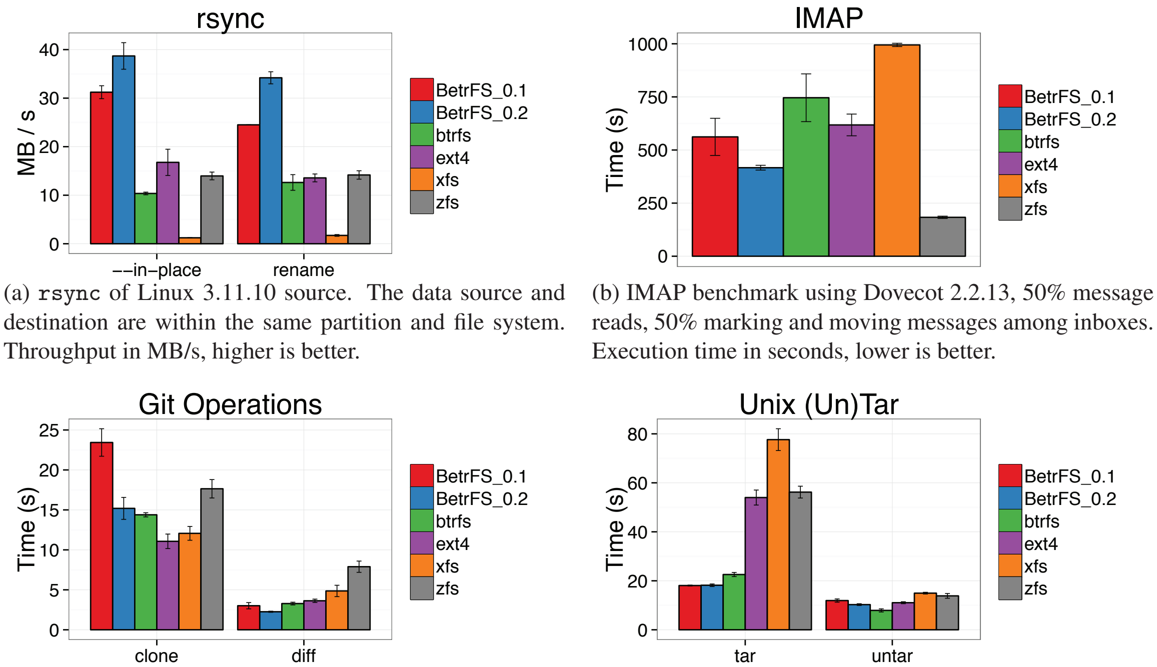
This subsection evaluates the best cases for a write-optimized file system, including small random writes, file creation, and searches. We confirm that our optimizations have not eroded the benefits of write-optimization. In most cases, there is no loss.

Small, random writes. Table 2 shows the execution time of a microbenchmark that issues 10,000 4-byte overwrites at random offsets within a 10GiB file, followed by an `fsync`. BetrFS 0.2 not only retains a two orders-of-magnitude improvement over the other file systems, but improves the latency over BetrFS 0.1 by 34%.

Small file creation. To evaluate file creation, we used the TokuBench benchmark [7] to create three million 200-byte files in a balanced directory tree with a fanout of 128. We used 4 threads, one per core of the machine.

Figure 8 graphs files created per second as a function of the number of files created. In other words, the point at 1 million on the x-axis is the cumulative throughput at the time the millionth file is created. zfs exhausts system memory after creating around a half million files.

The line for BetrFS 0.2 is mostly higher than the line



(a) `rsync` of Linux 3.11.10 source. The data source and destination are within the same partition and file system. Throughput in MB/s, higher is better.

(b) IMAP benchmark using Dovecot 2.2.13, 50% message reads, 50% marking and moving messages among inboxes. Execution time in seconds, lower is better.

(c) Git operations. The BetrFS source repository was `git-clone`d locally, and a `git-diff` was taken between two milestone commits. Lower is better.

(d) Unix `tar` operations. The Linux version 3.11.10 source code was both `tar`ed and un-`tar`ed. Time is in seconds. Lower is better.

Figure 9: Application benchmarks

for BetrFS 0.1, and both sustain throughputs at least 3×, but often an order of magnitude, higher than any other file system (note the y-axis is log scale). Due to TokuBench's balanced directory hierarchy and write patterns, BetrFS 0.2 performs 16,384 zone splits in quick succession at around 2 million files. This leads to a sudden drop in performance and immediate recovery.

Searches. Table 1 shows the time to search for files named "wait.c" (`find`) and to search the file contents for the string "cpu_to_be64" (`grep`). These operations are comparable on both write-optimized file systems, although BetrFS 0.2 `grep` slows by 46%, which is attributable to the trade-offs to add zoning.

7.4 Application Performance

This subsection evaluates the impact of the BetrFS 0.2 optimizations on the performance of several applications, shown in Figure 9. Figure 9a shows the throughput of an `rsync`, with and without the `--in-place` flag. In both cases, BetrFS 0.2 improves the throughput over BetrFS 0.1 and maintains a significant improvement over other file systems. Faster sequential I/O and, in the second case, faster rename, contribute to these gains.

In the case of `git-clone`, sequential write improvements make BetrFS 0.2 performance comparable to other file systems, unlike BetrFS 0.1. Similarly, BetrFS 0.2 marginally improves the performance of `git-diff`, making it clearly faster than the other FSes.

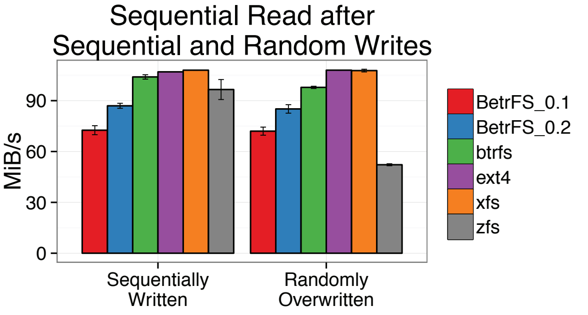


Figure 10: Sequential read throughput after sequentially writing a 10GiB file (left) and after partially overwriting 10,000 random blocks in the file (right). Higher is better.

Both BetrFS 0.2 and zfs outperform other file systems on the Dovecot IMAP workload, although zfs is the fastest. This workload is characterized by frequent small writes and `fsync`s, and both file systems persist small updates quickly by flushing their logs.

On BetrFS 0.2, `tar` is 1% slower than BetrFS 0.1 due to the extra work of splitting zones.

7.5 Background costs

This subsection evaluates the overheads of deferred work attributable to batching in a WOD. To measure the cost of deferred writes, we compare the time to read a sequentially written 10GiB file to the time to read that same file after partially overwriting 10,000 random blocks. Both reads are cold cache, and shown in Figure 10.

This experiment demonstrates that BetrFS 0.2's effective read throughput is nearly identical (87MiB/s vs. 85MiB/s), regardless of how the file was written.

8 Related Work

Zoning. Dynamic subtree partitioning [35] is a technique designed for large-scale distributed systems, like Ceph [36], to reduce metadata contention and balance load. These systems distribute (1) the number of metadata objects and (2) the frequency of metadata accesses, across nodes. Zones instead partition objects according to their aggregate *size* to bound rename costs.

Spyglass [15] introduces a partitioning technique for multi-dimensional metadata indices based on KD-trees. Partitioning techniques have also been used to determine which data goes on slower versus faster media [23], to efficiently maintain inverted document indices [14], or to plug in different storage data structures to optimize for read- or write-intensive workloads [19]. Chunkfs [10] partitions the ext2 file system to improve recovery time. A number of systems also divide disk bandwidth and cache space for performance isolation [33, 34]; speaking generally, these systems are primarily concerned with fairness across users or clients, rather than bounding worst-case execution time. These techniques strike domain-specific trade-offs different from zoning's balance of directory searches and renames.

IceFS [18] uses cubes, a similar concept to zones, to isolate faults, remove physical dependencies in data structures and transactional mechanisms, and allow for finer granularity recovery and journal configuration. Cubes are explicitly defined by users to consist of an entire directory subtree, and can grow arbitrarily large as users add more data. In contrast, zones are completely transparent to users, and dynamically split and merged.

Late-binding log entries. KVFS [30] avoids the journaling overhead of writing most data twice by creating a new VT-tree snapshot for each transaction. When a transaction commits, all in-memory data from the transaction's snapshot VT-tree is committed to disk, and that transaction's VT-tree is added *above* dependent VT-trees. Data is not written twice in this scenario, but the VT-tree may grow arbitrarily tall, making search performance difficult to reason about.

Log-structured file systems [1, 13, 16, 27] avoid the problem of duplicate writes by only writing into a log. This improves write throughput in the best cases, but does not enforce an optimal lower bound on query time.

Physical logging [9] stores before- and after-images of individual database pages, which may be expensive for large updates or small updates to large objects. Logical logging [17] may reduce log sizes when operations have succinct representations, but not for large data inserts.

The zfs intent log combines copy-on-write updates and indirection to avoid log write amplification for large records [21]. We adapt this technique to implement late-binding journaling of large messages (or large groups of related small messages) in BetrFS 0.2.

Previous systems have implemented variations of soft updates [22], where data is written first, followed by metadata, from leaf-to-root. This approach orders writes so that on-disk structures are always a consistent checkpoint. Although soft updates may be possible in a B$^{\varepsilon}$-tree, this would be challenging. Like soft updates, the late-binding journal avoids the problem of doubling large writes, but, unlike soft updates, is largely encapsulated in the block allocator. Late-binding imposes few additional requirements on the B$^{\varepsilon}$-tree itself and does not delay writes of any tree node to enforce ordering. Thus, a late-binding journal is particularly suitable for a WOD.

9 Conclusion

This paper shows that write-optimized dictionaries can be practical not just to accelerate special cases, but as a building block for general-purpose file systems. BetrFS 0.2 improves the performance of certain operations by orders of magnitude and offer performance comparable to commodity file systems on all others. These improvements are the product of fundamental advances in the design of write-optimized dictionaries. We believe some of these techniques may be applicable to broader classes of file systems, which we leave for future work.

The source code for BetrFS 0.2 is available under GPLv2 at `github.com/oscarlab/betrfs`.

Acknowledgments

We thank the anonymous reviewers and our shepherd Eddie Kohler for their insightful comments on earlier drafts of the work. Nafees Abdul, Amit Khandelwal, Nafisa Mandliwala, and Allison Ng contributed to the BetrFS 0.2 prototype. This research was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, CNS-1149229, CNS-1161541, CNS-1228839, IIS-1247750, CCF-1314547, CNS-1526707 and VMware.

References

[1] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2009), pp. 1–14.

[2] 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 ACM symposium on Parallelism in algorithms and architectures (SPAA)* (2007), pp. 81–92.

[3] Bender, M. A., Farach-Colton, M., Jannen, W., Johnson, R., Kuszmaul, B. C., Porter, D. E., Yuan, J., and Zhan, Y. An introduction to B^e-trees and write-optimization. *:login; Magazine 40*, 5 (Oct 2015), 22–28.

[4] Bonwick, J., and Moore, B. ZFS: The Last Word in File Systems. `http://opensolaris.org/os/community/zfs/docs/zfslast.pdf`.

[5] Brodal, G. S., Demaine, E. D., Fineman, J. T., Iacono, J., Langerman, S., and Munro, J. I. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2010), pp. 1448–1456.

[6] Brodal, G. S., and Fagerberg, R. Lower bounds for external memory dictionaries. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2003), pp. 546–554.

[7] Esmet, J., Bender, M. A., Farach-Colton, M., and Kuszmaul, B. C. The TokuFS streaming file system. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)* (2012).

[8] Garimella, N. Understanding and exploiting snapshot technology for data protection. `http://www.ibm.com/developerworks/tivoli/library/t-snaptsm1/`, Apr. 2006.

[9] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[10] Henson, V., van de Ven, A., Gud, A., and Brown, Z. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the USENIX Conference on Hot Topics in System Dependability (HotDep)* (2006).

[11] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M., Farach-Colton, M., Johnson, R., Kuszmaul, B. C., and Porter, D. E. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 301–315.

[12] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M., Farach-Colton, M., Johnson, R., Kuszmaul, B. C., and Porter, D. E. BetrFS: Write-optimization in a kernel file system. *ACM Transactions on Storage (TOS) 11*, 4 (Oct. 2015), 18:1–18:29.

[13] Lee, C., Sim, D., Hwang, J., and Cho, S. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 273–286.

[14] Lester, N., Moffat, A., and Zobel, J. Fast on-line index construction by geometric partitioning. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)* (2005), pp. 776–783.

[15] Leung, A. W., Shao, M., Bisson, T., Pasupathy, S., and Miller, E. L. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2009), pp. 153–166.

[16] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2011), pp. 1–13.

[17] Lomet, D., and Tuttle, M. Logical logging to extend recovery to new domains. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (1999), pp. 73–84.

[18] Lu, L., Zhang, Y., Do, T., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Physical disentanglement in a container-based file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014), pp. 81–96.

[19] Mammarella, M., Hovsepian, S., and Kohler, E. Modular data storage with Anvil. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2009), pp. 147–160.

[20] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. The new ext4 filesystem: Current status and future plans. In *Linux Symposium* (2007).

[21] McKusick, M., Neville-Neil, G., and Watson, R. *The Design and Implementation of the FreeBSD Operating System*. Addison Wesley, 2014.

[22] Mckusick, M. K., and Ganger, G. R. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the USENIX Annual Technical Conference* (1999), pp. 1–17.

[23] Mitra, S., Winslett, M., and Hsu, W. W. Query-based partitioning of documents and indexes for information lifecycle management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2008), pp. 623–636.

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

[25] Ren, K., and Gibson, G. A. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the USENIX Annual Technical Conference* (2013), pp. 145–156.

[26] Rodeh, O., Bacik, J., and Mason, C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (Aug. 2013), 9:1–9:32.

[27] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (Feb. 1992), 26–52.

[28] Santry, D., and Voruganti, K. Violet: A storage stack for IOPS/capacity bifurcated storage environments. In *Proceedings of the USENIX Annual Technical Conference* (2014), pp. 13–24.

[29] Sears, R., and Ramakrishnan, R. bLSM: a general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2012), pp. 217–228.

[30] Shetty, P., Spillane, R. P., Malpani, R., Andrews, B., Seyster, J., and Zadok, E. Building workload-independent storage with VT-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2013), pp. 17–30.

[31] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference* (1996), pp. 1–14.

[32] Tokutek, Inc. TokuDB: MySQL Performance, MariaDB Performance. `http://www.tokutek.com/products/tokudb-for-mysql/`, 2013.

[33] Verghese, B., Gupta, A., and Rosenblum, M. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1998), pp. 181–192.

[34] Wachs, M., Abd-El-Malek, M., Thereska, E., and Ganger, G. R. Argon: Performance insulation for shared storage servers. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2007), pp. 5–5.

[35] Weil, S., Pollack, K., Brandt, S. A., and Miller, E. L. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)* (Nov. 2004).

[36] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 307–320.

[37] Wu, X., Xu, Y., Shao, Z., and Jiang, S. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *Proceedings of the USENIX Annual Technical Conference* (2015), pp. 71–82.

The Composite-file File System: Decoupling the One-to-one Mapping of Files and Metadata for Better Performance

Shuanglong Zhang, Helen Catanese, and An-I Andy Wang
Computer Science Department, Florida State University

Abstract

Traditional file system optimizations typically use a one-to-one mapping of logical files to their physical metadata representations. This mapping results in missed opportunities for a class of optimizations in which such coupling is removed.

We have designed, implemented, and evaluated a composite-file file system, which allows many-to-one mappings of files to metadata, and we have explored the design space of different mapping strategies. Under webserver and software development workloads, our empirical evaluation shows up to a 27% performance improvement. This result demonstrates the promise of composite files.

1. Introduction

File system performance optimization is a well-researched area. However, most optimization techniques (e.g., caching, better data layout) retain the one-to-one mapping of logical files to their physical metadata representations (i.e., each file is associated with its own i-node on UNIX platforms). Such mapping is desirable because metadata constructs are deep-rooted data structures, and many storage components and mechanisms—such as VFS API [MCK90], prefetching, and metadata caching—rely on such constructs. However, this rigid mapping also presents a blind spot for a class of performance optimizations.

We have designed, implemented, and evaluated the composite-file file system (CFFS), where many logical files can be grouped together and associated with a single i-node (plus extra information stored as extended attributes). Such an arrangement is possible because many files accessed together share similar metadata subfields [EDL04], which can be deduplicated. Thus, the CFFS can yield fewer metadata accesses to storage, a source of significant overhead for accessing small files, which still dominates the majority of file references for modern workloads [ROS00; HAR11].

Based on web server and software development workloads, the CFFS can outperform ext4 by up to 27%, suggesting that the approach of relaxing the file-to-metadata mapping is promising.

2. Observations

The following observations led to the CFFS design:

Frequent access to small files: Studies [ROS00; HAR11] show that small files receive the majority of file references. Our in-house analyses of a desktop file system confirmed that >80% of accesses are to files smaller than 32 bytes. Further, ~40% of the access time to access a small file on a disk can be attributable to metadata access. Thus, reducing this access overhead may lead to a large performance gain.

Redundant metadata information: A traditional file is associated with its own physical metadata, which tracks information such as the locations of file blocks, access permissions, etc. However, many files share similar file attributes, as the number of file owners, permission patterns, etc. are limited. Edel et al. [2004] showed up to a 75% metadata compression ratio for a typical workstation. Thus, we see many opportunities to reduce redundant metadata information.

Files accessed in groups: Files tend to be accessed together, as shown by [KRO01, LI04, DIN07, and JIA13]. For example, web access typically involves accessing many associated files. However, optimizations that exploit file grouping may not yield automatic performance gains, as the process of identifying and grouping files incurs overhead.

Limitations of prefetching: While prefetching is an effective optimization, the separate act of fetching each file and associated metadata access can impose a high overhead. For example, accessing 32 small files can incur 50% higher latency than accessing a single file with a size equal to the sum of the 32 files, even with warm caches.

This observation begs the question of whether we can improve performance by consolidating small files that are accessed together. This is achieved through our approach of decoupling the one-to-one mapping of logical files to their physical representation of metadata.

3. Composite-file File System

We introduce the CFFS, which allows multiple small files to be combined and share a single i-node.

3.1. Design Overview

The CFFS introduces an internal physical representation called a ***composite file***, which holds the content of small files that are often accessed together. A composite file is invisible to end users and is associated with a single composite i-node shared among small files. The original information stored in small files' inodes are deduplicated and stored as extended attributes of a composite file. The metadata attributes of individual small files can still be reconstructed, checked, and updated, so that legacy access semantics (e.g., types, permissions, timestamps) are unchanged. The extended attributes also record the locations within the composite file for individual small files. With this representation, the CFFS can translate a physical composite file into logical files.

Which files to combine into a composite file is an important workload-dependent policy decision. As an example, the CFFS has been configured three ways. The first scheme is ***directory-based consolidation***, where all files within a directory (excluding subdirectories) form a composite file. The second scheme is ***embedded-reference consolidation***, where file references within file contents are extracted to identify files that can form composite files. The third is ***frequency-mining-based consolidation***, where file references are analyzed through set frequency mining [AGR94], so that files that are accessed together frequently form composite files.

A composite file exploits legacy VFS prefetching mechanisms because the entire composite file may be prefetched as a unit in a similar manner to the benefit FFS achieved by combining small data blocks into fewer larger blocks [MCK84].

3.2. Data Representation

The content of a composite file is formed by concatenating small files, referred to as ***subfiles***. All subfiles within a composite file share the same i-node, as well as indirect blocks, doubly indirect blocks, etc. The maximum size limit of a composite file is not a concern, as composite files are designed to group small files. If the sum of subfile sizes exceeds the maximum file size limit, we can resort to the use of multiple composite files.

Often, the first subfile in a composite file is the ***entry point***, whose access will trigger the prefetching of the remaining subfiles. For example, when a browser accesses an `html` file, it loads a `css` file and flash script. The `html` file can serve as the entry point and prefetching trigger of this three-subfile composite file. For the frequency-based consolidation, the ordering of subfiles reflects how they are accessed. Although the same group of files may have different access patterns with different entry points, the data layout is based on the most prevalent access pattern.

3.3. Metadata Representations and Operations

Composite file creation: When a composite file is created, the CFFS allocates an i-node and copies and concatenates the contents of the subfiles as its data. The composite file records the composite file offsets and sizes of individual subfiles as well as their deduplicated i-node information into its extended attributes. The original subfiles then are truncated, with their directory entries remapped to the i-node of the composite file extended to also include the subfile ID and their original i-nodes deallocated. Thus, end users still perceive individual logical files in the name space, while individual subfiles can still be located (Figure 3.3.1).

I-node content reconstruction: Deduplicated subfile i-nodes are reconstructed on the fly. By default, a subfile's i-node field inherits the value of the composite file's i-node field, unless otherwise specified in the extended attributes.

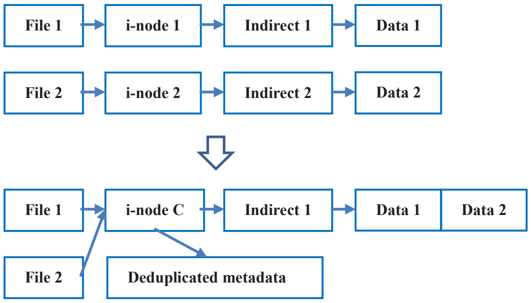


Figure 3.3.1: Creation of the internal composite file (bottom) from the two original files (top).

Permissions: At file open, the permission test is first checked based on the composite i-node. If this fails, no further check is needed. Otherwise, if a subfile has a different permission stored as an extended attribute, the permission will be checked again. Therefore, the composite i-node will have the broadest permissions across all subfiles. For example, if within a composite file, we have a read-only subfile A, and a writable subfile B, the permission for the composite i-node will be read-write. However, when opening subfile A with a write permission, the read-only permission restriction in the extended attribute will catch the violation.

Timestamps: The timestamps of individual subfiles and the composite file are updated with each file operation. However, during checks (e.g., `stat` system calls), we return the timestamps of the subfiles.

Sizes: For data accesses, the offsets are translated and bound-checked via subfile offsets and sizes encoded in the extended attributes. The size of a composite file is the length of the composite file, which can be greater than the total size of its subfiles. For example, if a subfile in the middle of a composite file is deleted, the region is freed, without changing the size of the composite file.

i-node namespace: For i-node numbers larger than a threshold X, upper zero-extended N bits are used for composite i-node numbers, and lower M bits are reserved for subfile IDs. We refer this range of i-node numbers as CFFS unique IDs (CUIDs).

Subfile lookups and renames: If a name in a directory is mapped to a CUID, a subfile's attributes can be looked up via the subfile ID. Renaming will proceed as if a CUID were an i-node number in a non-CFFS system. Since moving a subfile in and out of a composite file will change its CUID, we need to store backpointers [CHI12], to update all names mapped to the CUID.

The changes in CUID may break applications (e.g., backups) that uniquely identify a file by its i-node number. However, today's file systems can also lead to different files sharing the same i-node number at different times; the CFFS design amplifies the reasons that applications should not assume that an i-node number is a unique property of a file.

Subfile and subfile membership updates: When a subfile is added to a composite file, it is appended to the composite file. When a subfile is deleted from a composite file, the corresponding data region within the composite file is marked freed in the extended attributes.

Subfile open/close operations: An open/close call to a subfile is the same as an open/close call to the composite file, with the file-position pointer translated.

Subfile write operations: In-place updates are handled the same way as those in a traditional file system. However, if an update involves growing a subfile in the middle of a composite file and no free space is available at the end of the subfile, we move the updated subfile to the end of the composite file. This scheme exploits the potential temporal locality that a growing subfile is likely to grow again in the near future.

Hardlinks: Different names in directories can be mapped to the same i-node number or CUID.

Space compaction: The composite file compacts its space when half of its allotted size contains no useful data.

Concurrent updates to subfiles within a composite file: Concurrent updates to subfiles within a composite file carry the same semantics as concurrent updates to a normal file. To avoid lock contention, files detected to be involved in concurrent updates might have to be extracted into multiple regular files.

Locking and consistency: The CFFS does not support flock, but we believe it is possible to implement a subfile locking subsystem.

3.4. Identifying Composite File Membership

3.4.1 Directory-based Consolidation

Given that legacy file systems have deep-rooted spatial locality optimizations revolving around directories, a directory is a good approximation of file access patterns and for forming composite files. Currently, this consolidation scheme excludes subdirectories.

The directory-based consolidation can be performed on all directories without tracking and analyzing file references. However, it will not capture file relationships across directories.

3.4.2 Embedded-reference-based Consolidation

Embedded-reference-based consolidation identifies composite file memberships based on embedded file references in files. For example, hyperlinks may be embedded in an `html` file, and a web crawler is likely to access each web page via these links. In this case, we consolidate the original `html` file and the referenced files. Similar ideas apply to compilation. We can extract the dependency rules from `Makefile` and consolidate source files that lead to the generation of the same binary. As file updates may break a dependency, the CFFS could sift periodically through modified files to reconcile composite file membership.

The embedded-reference-based scheme can identify related files accessed across directories, but it may not be easy to extract embedded file references beyond text-based file formats (e.g., `html`, source code). In addition, it requires knowledge of specific file formats.

{A}	5
{B}	2
{C}	2
{D}	4
~~{E}~~	~~1~~

→

{A, B}	2
{A, C}	2
{A, D}	6
{B, C}	2
~~{B, D}~~	~~0~~
~~{C, D}~~	~~0~~

→

{A, B, C}	5
~~{A, B, D}~~	
~~{A, C, D}~~	
~~{B, C, D}~~	

Figure 3.4.3.1: Steps for the Apriori algorithm to identify frequently accessed file sets for a file reference stream E, D, A, D, A, D, A, B, C, A, B, C, A, D.

3.4.3 Frequency-mining-based Consolidation

In our exploration of a frequency-mining-based consolidation, we use a variant of the Apriori algorithm [AGR94]. The key observation is that if a set of files is accessed frequently, its subsets must be as well (the Apriori property). Figure 3.4.3.1 illustrates the algorithm with an access stream to files A, B, C, D, and E.

Initial pass: First, we count the number of accesses for each file, and then we remove files with counts less than a threshold (say two) for further analysis.

Second pass: For the remaining files, we permute, build, and count all possible two-file reference sets. Whenever file A is accessed right after B, or vice versa, we increment the count for file set {A, B}. Sets with counts less than the threshold are removed (e.g., {B, D}).

Third pass: We can generate all three-file reference sets based on the remaining two-file reference sets. However, if a three-file reference set occurs frequently, all its two-file reference sets also need to occur frequently. Thus, file sets such as {A, B, D} are pruned, since {B, D} is eliminated in the second pass.

Termination: As we can no longer generate four-file reference sets, the algorithm ends. Now, if a file can belong to multiple file sets, we return sets {A, B, C} and {A, D} as two frequently accessed sets. Sets such as {A, B} are removed as they are subsets of {A, B, C}.

Variations: An alternative is to use a normalized threshold, or ***support***, which is the percentage of set occurrences (number of the occurrences of a set divided by the total occurrences, ranged between 0 and 1).

Instead of tracking file sets, we can also track file reference sequences to determine the entry point and the content layout of the composite file.

We currently disallow overlapping file sets to avoid the complexity of replication and maintaining consistency. To choose a subfile's membership between two composite files, the decision depends on whether a composite file has more subfiles, higher support, and more recent creation timestamps.

The frequency-mining-based consolidation can identify composite file candidates based on dynamic file references. However, the cost of running it limits its application to more popular file reference sequences.

4. Implementation

The two major components of our prototype are the composite file membership generator tool and the CFFS.

We prototyped the CFFS in user space via the FUSE (v2.9.3) framework [SZE05] (Figure 4.1) running on top of Linux 3.16.7. The CFFS is stacked on ext4, so that we can leverage legacy tools and features such as persistence bootstrapping (e.g., file-system creation utilities), extended attributes, and journaling.

The CFFS periodically consults with the generator tool to create new composite files. We leveraged mechanisms similar to hardlinks to allow multiple file names to be mapped to the same composite i-node. We intercepted all file-system-related calls due to the need to update the timestamps of individual subfiles. We also need to ensure that various accesses use the correct permissions (e.g., `open` and `readdir`), translated subfile offsets and sizes (e.g., `read` and `write`), and timestamps (e.g., `getattr` and `setattr`). The actual composite file, its i-node, and its extended attributes are stored by the underlying ext4 file system. The CFFS is implemented in C++ with ~1,600 semicolons.

For directory-based consolidation, we used a Perl script to list all the files in a directory as composite file members. For the embedded-reference-based scheme, we focus on two scenarios. For the web server workload, we consolidate `html` files and their immediately referenced files. In the case of conflicting composite file memberships, preference is given to `index.html`, and then the `html` that first includes a file. The other is the source code compilation. We used `Makefile` as a guide to consolidate source code files. For the frequency-mining-based scheme, the membership generator tool takes either a `http` access log or a `strace` output. The generator implements the Apriori algorithm, with the support parameter set to 5%. The analysis batch size is set to 50K references. The parameters were chosen based on empirical experience to limit the amount of memory and processing overhead. The generator code contains ~1,200 semicolons.

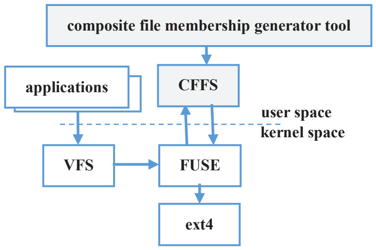


Figure 4.1: CFFS components (shaded) and data path from applications to the underlying ext4.

Table 5.1: Experimental platform.

Processor	2.8GHz Intel® Xeon® E5-1603, L1 cache 64KB, L2 cache 256KB, L3 cache 10MB
Memory	2GBx4, Hyundai, 1067MHz, DDR3
Disk	250GB, 7200 RPM, WD2500AAKX with 16MB cache
Flash	200GB, Intel SSD DC S3700

5. Performance Evaluation

We compared the performance of the CFFS stacked on ext4 via FUSE with the baseline ext4 file system (with the requests routed through an empty FUSE module).

We evaluated our system via replaying two traces. The first is an `http` log gathered from our departmental web server (01/01/2015-03/18/2015). The trace contains 14M file references to 1.0TB of data; of this, 3.1M files are unique, holding 76GB of data. The second trace was gathered via `strace` from a software development workstation (11/20/2014 – 11/30/2014). The trace contained over 240M file-system-related system calls to 24GB of data; of this, 291,133 files are unique with 2.9GB bytes. Between read and write operations, 59% are reads, and 41% are writes.

We conducted multi-threaded, zero-think-time trace replays on a storage device. We also skipped trace intervals with no activities. The replay experiments were performed on a Dell workstation (Table 5.1). Each experiment was repeated 5 times, and results are presented at 90% confidence intervals.

Prior to each experiment, we rebuilt the file system with dummy content. For directory- and embedded-reference-based schemes, composite file memberships are updated continuously. For the frequency-mining-based consolidation, the analysis is performed in batches, but the composite files are updated daily.

5.2. Web Server Trace Replay

HDD performance: Figure 5.2.1 shows the CDF of web server request latency for a disk, measured from the time a request is sent to the time a request is completed.

The original intent of our work is to reduce the number of IOs for small files that are frequently accessed together. However, the benefit of fewer accesses to consolidated metadata displays itself as metadata prefetching for all subfiles, and the composite-file semantics enable cross-file prefetching, resulting in much higher cache-hit rates.

The embedded-reference-based consolidation performed the best, with 62% of requests serviced from the cache, which is 20% higher than ext4. Thus, composite files created based on embedded references capture the access pattern more accurately. The overall replay time was also reduced by ~20%.

The directory-based composite files can also improve the cache-hit rate by 15%, reflecting the effectiveness of directories to capture spatial localities.

The frequency-mining-based consolidation performed worse than the directory-based. We examined the trace and found that 48% of references are made by crawlers, and the rest by users. Thus, the bifurcated traffic patterns for the mining algorithm form less aggressive file groupings, yielding reduced benefits.

SSD Performance: Figure 5.2.2 shows the CDF of web server request latency for an SSD. Compared to a disk, the relative trends are similar, with request latency times for cache misses reduced by two orders of magnitude due to the speed of the SSD. As the main performance gains are caused by higher cache-hit rates and IO avoidance, this 20% benefit is rather independent of the underlying storage media.

5.3. Software Development File-system Trace Replay

For the software development workload replay, it is more difficult to capture the latency of individual file-system call requests, as many are asynchronous (e.g., writes), and calls like `mmap` do not know the number of requests sent to the underlying storage. Thus, we summarize our results with overall elapsed times, which include all overheads of composite file operations, excluding the initial setup cost for the directory- and embedded-reference-based schemes (Figure 5.3.1).

HDD performance: The embedded-reference-based scheme has poor coverage, as many references are unrelated to compilation. Therefore, the elapsed time is closer to that of ext4. Directory-based consolidation achieves a 17% elapsed time reduction, but the frequency-mining-based scheme can achieve 27% because composite files include files across directories.

SSD performance: The relative performance trend for different consolidation settings is similar to that of HDD. Similar to the web traces, the gain is up to 20%.

When comparing the performance improvement gaps between the HDD and SSD experiments, up to an 11% performance gain under HDD cannot be realized by SSD, as an SSD does not incur disk seek overheads.

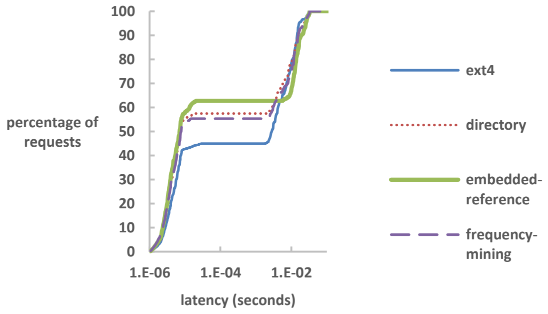


Figure 5.2.1: Web server request latency for HDD.

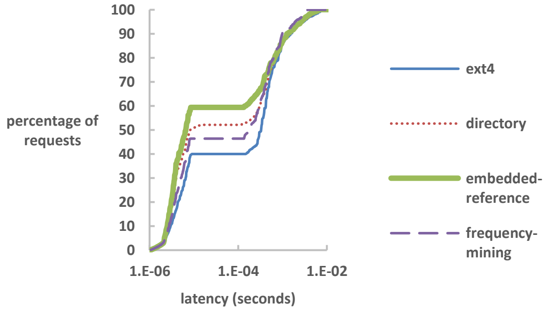


Figure 5.2.2: Web server request latency for SSD.

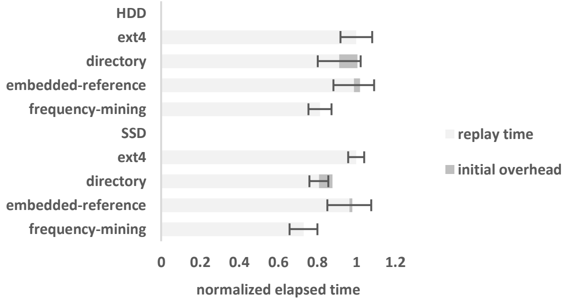


Figure 5.3.1: Elapsed times for 5 runs of the software development file system trace replay. Error bars show 90% confidence intervals.

5.4. Overheads

Directory- and embedded-reference-based schemes: Directory- and embedded-reference-based schemes incur an initial deployment cost to create composite files based on directories and embedded file references. The initial cost of the embedded-reference scheme depends on the number of file types from which file references can be extracted. For our workloads, this cost is anywhere from 1 to 14 minutes (Figure 5.3.1).

As for the incremental cost of updating composite file memberships, adding members involves appending to the composite files. Removing members involves mostly metadata updates. A composite file is not compacted until half its allotted space is unused. As the trace replay numbers already include this overhead, this cost is offset by the benefits.

Frequency-mining-based scheme: The trace gathering overhead is below 0.6%, and the memory overhead for trace analyses is within 200MB for an average of 15M lines of daily logs.

The frequency-mining-based scheme involves learning from recent file references, and it took a few replay days to reap the full benefit of this scheme.

5.5. Discussion and Future Work

Composite files can benefit both read-dominant and read-write workloads using different storage media, suggesting that the performance gains are mostly due to the reduction in the number of IOs (~20%). The performance improvement gaps between the SSD and HDD suggest the performance gains due to reduced disk seeks and modified data layouts are up to ~10%.

Overall, we are intrigued by the relationship among ways to form composite files, the performance effects of consolidating metadata and prefetching enabled by the composite files. Future work will explore additional ways to form composite files and quantify their interplay with different components of performance contributions. Additionally, future work will more fully explore the ramifications of metadata compression, concurrency, and security.

6. Related Work

Small file optimizations: While our research focuses on the many-to-one mapping of logical files and physical metadata, this work is closely related to ways to optimize small file accesses by reducing the number of storage accesses. Early work on this area involve collocating a file's i-node with its first data block [MUL84] and embedding i-nodes in directories [GAN97]. Later, hFS [ZHA07] used separate storage areas to optimize small file and metadata accesses. Btrfs [ROD13] packs metadata and small files into copy-on-write b-trees. TableFS [REN13] packs metadata and small files into a table and flushes 2MB logs of table entry modifications, organized as a log-structured merge tree. The CFFS complements many existing approaches by consolidating i-nodes for files that are often accessed together.

The idea of accessing subfile regions and consolidating metadata is also explored in the parallel and distributed computing domain, where CPUs on multiple computers need to access the same large data file [YU07]. Facebook's photo storage [BEA10] leverages the observation that the permissions of images are largely the same and can be consolidated. However, these mechanisms are tailored for very homogeneous data types. With different ways to form composite files, the CFFS can work with subfiles with more diverse content and access semantics.

Prefetching: While a large body of work can be found to improve prefetching, perhaps C-Miner [LI04] is closest to our work. In particular, C-Miner applied frequent-sequence mining at the block level to optimize the layout of the file and metadata blocks and improve prefetching. However, the CFFS reduces the number of frequently accessed metadata blocks and avoids the need for a large table to map logical to physical blocks. In addition, our file-system-level mining deals with significantly fewer objects and associated overheads. DiskSeen [DIN07] incorporates the knowledge of disk layout to improve prefetching, and the prefetching can cross file and metadata boundaries. The CFFS proactively reduces the number of physical metadata items and alters the storage layout to promote sequential prefetching. Soundararajan et al. [2008] observed that by passing high-level execution contexts (e.g., thread, application ID) to the block layer, the resulting data mining can generate prefetching rules with longer runs under concurrent workloads. Since the CFFS performs data mining at the file-system level, we can use PIDs and IP addresses to detangle concurrent file references. Nevertheless, the CFFS's focus on altering the mapping of logical files to their physical representations, and it can adopt various mining algorithms to consolidate metadata and improve storage layouts.

7. Conclusions

We have presented the design, implementation, and evaluation of a composite-file file system, which explores the many-to-one mapping of logical files and metadata. The CFFS can be configured differently to identify files that are frequently accessed together, and it can consolidate their metadata. The results show up to a 27% performance improvement under two real-world workloads. The CFFS experience shows that the approach of decoupling the one-to-one mapping of files and metadata is promising and can lead to many new optimization opportunities.

Acknowledgement

We thank our shepherd Garth Gibson and anonymous reviewers for their invaluable feedback. We also thank Britton Dennis for contributing some of the preliminary numbers. This work is sponsored by FSU and NSF CNS-144387. Opinions, findings, conclusions, or recommendations expressed in this document do not necessarily reflect the views of FSU, NSF, or the U.S. Government.

References

[ADB05] Abd-El-Malek M, Courtright WV, Cranor C, Ganger GR, Hendricks J, CKlosterman AJ, Mesnier M, Prasad M, Salmon B, Sambasivan RR, Sinnamohideen S, Strunk JD, Thereska E, Wachs M, Wylie JJ. Ursa Minor: Versatile Cluster-based Storage. *Proceedings of the 4th USENIX Conference on File and Storage Technology (FAST)*, 2005.

[AGR94] Agrawal R, Srikant R. Fast Algorithms for Mining Association Rules, *Proceedings of the 20th VLDB Conference*, 1994.

[ALB15] Albrecht R. Web Performance: Cache Efficiency Exercise. https://code.facebook.com/posts/964122680272229/web-performance-cache-efficiency-exercise/, 2015.

[BEA10] Beaver D, Kumar S, Li HC, Vajgel P. Finding a Needle in Haystack: Facebook's Photo Storage. *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[BLO70] Bloom B. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM (CACM)*, 13(7):422-426, July 1970

[CHA13] Chandrasekar S, Dakshinamurthy R, Seshakumar PG, Prabavathy B, Chitra B. A Novel Indexing Scheme for Efficient Handling of Small Files in Hadoop Distributed File System. *Proceedings of 2013 International Conference on Computer Communication and Information*, 2013.

[CHI12] Chidambaram V, Sharma T, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Consistency without Ordering. *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[DIN07] Ding X, Jiang S, Chen F, Davis K, Zhang X. DiskSeen: Exploiting Disk Layout and Access History to Enhance Prefetch. *Proceedings of the 2007 USENIX Annual Technical Conference (ATC)*, 2007.

[DON10] Dong B, Qiu J. Zheng Q, Zhong X, Li J, Li Y. A Novel Approach to Improving the Efficiency of Storing and Accessing Smaller Files on Hadoop: a Case Study by PowerPoint Files. *Proceedings of the 2010 IEEE International Conference on Services Computing*, 2010.

[EDE04] Edel NK, Tuteja D, Miller EL, Brandt SA. *Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, 2004.

[GAN97] Ganger GR, Kaashoek MF. Embedded Inode and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. *Proceedings of the USENIX 1997 Annual Technical Conference (ATC)*, 1997.

[GAR09] Garrison JA, Reddy ALN. Umbrella File System: Storage Management across Heterogeneous Devices. *ACM Transactions on Storage (TOS)*, 5(1), Article 3, 2009.

[HAR11] Harter T, Dragga C, Vaughn M, Arpaci-Dusseau AC, Arpaci-Dusseau RH. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. *Proceedings of 23rd Symposium on Operating Systems Principles (SOSP)*, 2011.

[HEI94] Heidemann JS, Popek GJ. File-System Development with Stackable Layers. *ACM Transactions on Computer Systems (TOS)*, 22(1):58-89.

[JIA13] Jiang S, Ding X, Xu Y, Davis K. A Prefetching Scheme Exploiting Both Data Layout and Access History on Disk. *ACM Transactions on Storage (TOS)*, 9(3), Article No. 10.

[KRO01] Kroeger TM, Long DE. Design and Implementation of a Predictive File Prefetching. *Proceedings of the USENIX 2001 Annual Technical Conference (ATC)*, 2001.

[LI04] Li Z, Chen Z, Srinivasan SM, Zhou YY. C-Miner: Mining Block Correlations in Storage Systems. *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*, 2004.

[MCK84] McKusick MK, Joy WN, Leffler SJ, Fabry RS. A Fast File System for Unix. *ACM Transactions on Computer Systems*, 2(3):181-197, 1984.

[MCK90] McKusick MK, Karels MJ, Bostic K. A Pageable Memory Based Filesystem. *Proceeding of the 1990 USENIX Summer Conference*, June 1990.

[MUL84] Mullender S, Tanenbaum. Immediate Files, *Software Practice and Experience*, 14(4):365-368, 1984.

[REN13] Kai Ren, Garth Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System, *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013

[ROD13] Rodeh O, Bacik J, Mason C. BTRFS: The Linux B-Tree File System. *ACM Transactions on Storage (TOS)*, 9(3), Article No. 9, 2013.

[ROS00] Roselli D, Lorch JR, Anderson TE. A Comparison of File System Workloads. *Proceeding of 2000 USENIX Annual Technical Conference (ATC)*, 2000.

[SOU08] Soundararajan G, Mihailescu M, Amza C. Context-aware Prefetching at the Storage Server. *Proceedings of the 2008 USENIX Annual Technical Conference (ATC)*, 2008.

[SZE05] Szeredi M. Filesystem in Userspace. http://userspace.fuse.sourceforge.net, 2005.

[YU07] Yu W, Vetter J, Canon RS, Jian S. Exploiting Lustre File Joining for Effective Collective IO, *Proceedings of the 7th International Symposium on Cluster Computing and the Grid*, 2007.

[ZHA07] Zhihui Zhang, Kanad Ghose. hFS: A Hybrid File System Prototype fr Improving Small File and metadata Performance, *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

Isotope: Transactional Isolation for Block Storage

Ji-Yong Shin, *Cornell University*
Mahesh Balakrishnan, *Yale University*
Tudor Marian, *Google*
Hakim Weatherspoon, *Cornell University*

Abstract

Existing storage stacks are top-heavy and expect little from block storage. As a result, new high-level storage abstractions – and new designs for existing abstractions – are difficult to realize, requiring developers to implement from scratch complex functionality such as failure atomicity and fine-grained concurrency control. In this paper, we argue that pushing transactional isolation into the block store (in addition to atomicity and durability) is both viable and broadly useful, resulting in simpler high-level storage systems that provide strong semantics without sacrificing performance. We present Isotope, a new block store that supports ACID transactions over block reads and writes. Internally, Isotope uses a new multiversion concurrency control protocol that exploits fine-grained, sub-block parallelism in workloads and offers both strict serializability and snapshot isolation guarantees. We implemented several high-level storage systems over Isotope, including two key-value stores that implement the LevelDB API over a hashtable and B-tree, respectively, and a POSIX filesystem. We show that Isotope's block-level transactions enable systems that are simple (100s of lines of code), robust (i.e., providing ACID guarantees), and fast (e.g., 415 MB/s for random file writes). We also show that these systems can be composed using Isotope, providing applications with transactions across different high-level constructs such as files, directories and key-value pairs.

1 Introduction

With the advent of multi-core machines, storage systems such as filesystems, key-value stores, graph stores and databases are increasingly parallelized over dozens of cores. Such systems run directly over raw block storage but assume very little about its interface and semantics; usually, the only expectations from the block store are durability and single-operation, single-block linearizability. As a result, each system implements complex code to layer high-level semantics such as atomicity and isolation over the simple block address space. While multiple systems have implemented transactional atomicity within the block store [18, 24, 46, 6, 19], concurrency control has traditionally been delegated to the storage system above the block store.

In this paper, we propose the abstraction of a transactional block store that provides isolation in addition to atomicity and durability. A number of factors make isolation a prime candidate for demotion down the stack.

1) Isolation is *general*; since practically every storage system has to ensure safety in the face of concurrent accesses, an isolation mechanism implemented within the block layer is broadly useful.
2) Isolation is *hard*, especially for storage systems that need to integrate fine-grained concurrency control with coarse-grained durability and atomicity mechanisms (e.g., see ARIES [40]); accordingly, it is better provided via a single, high-quality implementation within the block layer.
3) Block-level transactions allow storage systems to effortlessly provide end-user applications with transactions over high-level constructs such as files or key-value pairs.
4) Block-level transactions are oblivious to software boundaries at higher levels of the stack, and can seamlessly span multiple layers, libraries, threads, processes, and interfaces. For example, a single transaction can encapsulate an end application's accesses to an in-process key-value store, an in-kernel filesystem, and an out-of-process graph store.
5) Finally, multiversion concurrency control (MVCC) [17] provides superior performance and liveness in many cases but is particularly hard to implement for storage systems since it requires them to maintain multiversioned state; in contrast, many block stores (e.g., log-structured designs) are already internally multiversioned.

Block-level isolation is enabled and necessitated by recent trends in storage. Block stores have evolved over time. They are increasingly implemented via a combination of host-side software and device firmware [9, 3]; they incorporate multiple, heterogeneous physical devices under a single address space [59, 56]; they leverage new NVRAM technologies to store indirection metadata; and they provide sophisticated functionality such as virtualization [9, 61], tiering [9], deduplication and wear-leveling. Unfortunately, storage systems such as filesystems continue to assume minimum functionality from the block store, resulting in redundant, complex, and inefficient stacks where layers constantly tussle with each other [61]. A second trend that argues for pushing functionality from the filesystem to a lower layer is the increasing importance of alternative abstractions that can be implemented directly over block storage, such as graphs, key-value pairs [8], tables, caches [53], tracts [42], byte-addressable [14] and write-once [15] address spaces, etc.

To illustrate the viability and benefits of block-level isolation, we built Isotope, a transactional block store that provides isolation (with a choice of strict serializability or snapshot isolation) in addition to atomicity and durability. Isotope is implemented as an in-kernel software module running over commodity hardware, exposing a conventional block read/write interface augmented with *beginTX*/*endTX* IOCTLs to demarcate transactions. Transactions execute speculatively and are validated by Isotope on *endTX* by checking for conflicts. To minimize the possibility of conflict-related aborts, applications can provide information to Isotope about which sub-parts of each 4KB block are read or written, allowing Isotope to perform conflict detection at sub-block granularity.

Internally, Isotope uses an in-memory multiversion index over a persistent log to provide each transaction with a consistent, point-in-time snapshot of a block address space. Reads within a transaction execute against this snapshot, while writes are buffered in RAM by Isotope. When *endTX* is called, Isotope uses a new MVCC commit protocol to determine if the transaction commits or aborts. The commit/abort decision is a function of the timestamp-ordered stream of recently proposed transactions, as opposed to the multiversion index; as a result, the protocol supports arbitrarily fine-grained conflict detection without requiring a corresponding increase in the size of the index. When transactions commit, their buffered writes are flushed to the log, which is implemented on an array of physical drives [56], and reflected in the multiversion index. Importantly, aborted transactions do not result in any write I/O to persistent storage.

Storage systems built over Isotope are simple, stateless, shim layers that focus on mapping some variable-sized abstraction – such as files, tables, graphs, and key-value pairs – to a fixed-size block API. We describe several such systems in this paper, including a key-value store based on a hashtable index, one based on a B-tree, and a POSIX user-space filesystem. These systems do not have to implement their own fine-grained locking for concurrency control and logging for failure atomicity. They can expose transactions to end applications without requiring any extra code. Storage systems that reside on different partitions of an Isotope volume can be composed with transactions into larger end applications.

Block-level isolation does have its limitations. Storage systems built over Isotope cannot share arbitrary, in-memory soft state such as read caches across transaction boundaries, since it is difficult to update such state atomically based on the outcome of a transaction. Instead, they rely on block-level caching in Isotope by providing hints about which blocks to cache. We found this approach well-suited for both the filesystem application (which cached inode blocks, indirection blocks and allocation maps) and the key-value stores (which cached their index data structures). In addition, information is invariably lost when functionality is implemented at a lower level of the stack: Isotope cannot leverage properties such as commutativity and idempotence while detecting conflicts.

This paper makes the following contributions:

- We revisit the end-to-end argument for storage stacks with respect to transactional isolation, in the context of modern hardware and applications.
- We propose the abstraction of a fully transactional block store that provides isolation, atomicity and durability. While others have explored block-level transactional atomicity [18, 24, 46, 19], this is the first proposal for block-level transactional isolation.
- We realize this abstraction in a system called Isotope via a new MVCC protocol. We show that Isotope exploits sub-block concurrency in workloads to provide a high commit rate for transactions and high I/O throughput.
- We describe storage systems built using Isotope transactions – two key-value stores and a filesystem – and show that they are simple, fast, and robust, as well as composable via Isotope transactions into larger end applications.

2 Motivation

Block-level isolation is an idea whose time has come. In the 90s, the authors of Rio Vista (a system that provided atomic transactions over a persistent memory abstraction) wrote in [36]: "*We believe features such as serializability are better handled by higher levels of software... adopting any concurrency control scheme would penalize the majority of applications, which are single-threaded and do not need locking.*" Today, applications run on dozens of cores and are multi-threaded by default; isolation is a universal need, not a niche feature.

Isolation is simply the latest addition to a long list of features provided by modern block stores: caching, tiering, mapping, virtualization, deduplication, and atomicity. This explosion of features has been triggered partly by the emergence of software-based block layers, ranging from flash FTLs [3] to virtualized volume managers [9]. In addition, the block-level indirection necessary for many of these features has been made practical and inexpensive by hardware advances in the last decade. In the past, smart block devices such as HP AutoRAID [65] were restricted to enterprise settings due to their reliance on battery-backed RAM; today, SSDs routinely implement indirection in FTLs, using supercapacitors to flush metadata and data on a power failure. Software block stores in turn can store metadata on these SSDs, on raw flash, or on derivatives such as flash-backed RAM [34] and Auto-Commit Memory [7].

What about the end-to-end argument? We argue that block-level isolation passes the litmus test imposed by the end-to-end principle [49] for pushing functionality down the stack: it is broadly useful, efficiently implementable at a lower layer of the stack with negligible performance overhead, and leverages machinery that already exists at that lower layer. The argument regarding utility is obvious: pushing functionality down the stack is particularly useful when it is general enough to be used by the majority of applications, which is undeniably the case for isolation or concurrency control. However, the other motivations for a transactional block store require some justification:

Isolation is hard. Storage systems typically implement pessimistic concurrency control via locks, opening the door to a wide range of aberrant behavior such as deadlocks and livelocks. This problem is exacerbated when developers attempt to extract more parallelism via fine-grained locks, and when these locks interact with coarse-grained failure atomicity and durability mechanisms [40]. Transactions can provide a simpler programming model that supplies isolation, atomicity and durability via a single abstraction. Additionally, transactions decouple the policy of isolation – as expressed through *beginTX/endTX* calls – from the concurrency control mechanism used to implement it under the hood.

Isolation is harder when exposed to end applications. Storage systems often provide concurrency control APIs over their high-level storage abstractions; for example, NTFS offers transactions over files, while Linux provides file-level locking. Unfortunately, these high-level concurrency control primitives often have complex, weakened, and idiosyncratic semantics [44]; for instance, NTFS provides transactional isolation for accesses to the same file, but not for directory modifications, while a Linux *fcntl* lock on a file is released when any file descriptor for that file is closed by a process [1]. The complex semantics are typically a reflection of a complex implementation, which has to operate over high-level constructs such as files and directories. In addition, composability is challenging if each storage system implements isolation independently: for example, it is impossible to do a transaction over an NTFS file and a Berkeley DB key-value pair.

Isolation is even harder when multiversion concurrency control is required. In many cases, pessimistic concurrency control is slow and prone to liveness bugs; for example, when locks are exposed to end applications directly or via a transactional interface, the application could hang while holding a lock. Optimistic concurrency control [35] works well in this case, ensuring that other transactions can proceed without waiting for the hung process. Multiversion concurrency control works even better, providing transactions with stable, consistent snapshots (a key property for arbitrary applications that can crash if exposed to inconsistent snapshots [31]); allowing read-only transactions to always commit [17]; and enabling weaker but more performant isolation levels such as snapshot isolation [16].

```
/*** Transaction API ***/
int beginTX();
int endTX();
int abortTX();
//POSIX read/write commands
/*** Optional API ***/
//release ongoing transaction and return handle
int releaseTX();
//take over a released transaction
int takeoverTX(int tx_handle);
//mark byte range accessed by last read/write
int mark_accessed(off_t blknum, int start, int size);
//request caching for blocks
int please_cache(off_t blknum);
```

Figure 1: The Isotope API.

However, switching to multiversion concurrency control can be difficult for storage systems due to its inherent need for multiversion state. High-level storage systems are not always intrinsically multiversioned (with notable exceptions such as WAFL [33] and other copy-on-write filesystems), making it difficult for developers to switch from pessimistic locking to a multiversion concurrency control scheme. Multiversioning can be particularly difficult to implement for complex data structures used by storage systems such as B-trees, requiring mechanisms such as tombstones [26, 48].

In contrast, multiversioning is relatively easy to implement over the static address space provided by a block store (for example, no tombstones are required since addresses can never be 'deleted'). Additionally, many block stores are already multiversioned in order to obtain write sequentiality: examples include log-structured disk stores, shingled drives [11] and SSDs.

3 The Isotope API

The basic Isotope API is shown in Figure 1: applications can use standard POSIX calls to issue reads and writes to 4KB blocks, bookended by *beginTX/endTX* calls. The *beginTX* call establishes a snapshot; all reads within the transaction are served from that snapshot. Writes within the transaction are speculative. Each transaction can view its own writes, but the writes are not made visible to other concurrent transactions until the transaction commits. The *endTX* call returns true if the transaction commits, and false otherwise. The *abortTX* allows the application to explicitly abort the transaction. The application can choose one of two isolation levels on startup: strict serializability or snapshot isolation.

The Isotope API implicitly associates transaction IDs with user-space threads, instead of augmenting each call signature in the API with an explicit transaction ID that the application supplies. We took this route to allow applications to use the existing, highly optimized POSIX calls to read and write data to the block store. The control API for starting, committing and aborting transactions is implemented via IOCTLs. To allow transactions to execute across different threads or processes, Isotope provides additional APIs via IOCTLs: *releaseTX* disconnects the association between the current thread and the transaction, and returns a temporary transaction handle. A different thread can call *takeoverTX* with this handle to associate itself with the transaction.

Isotope exposes two other optional calls via IOCTLs. After reading or writing a 4KB block within a transaction, applications can call *mark_accessed* to explicitly specify the accessed byte range within the block. This information is key for fine-grained conflict detection; for example, a filesystem might mark a single inode within an inode block, or a single byte within a data allocation bitmap. Note that this information cannot be inferred implicitly by comparing the old and new values of the 4KB block; the application might have overwritten parts of the block without changing any bits. The second optional call is *please_cache*, which lets the application request Isotope to cache specific blocks in RAM; we discuss this call in detail later in the paper. Figure 2 shows a snippet of application code that uses the Isotope API (the *setattr* function from a filesystem).

If a read or write is issued outside a transaction, it is treated as a singleton transaction. In effect, Isotope behaves like a conventional block device if the reads and writes issued to it are all non-transactional. In addition, Isotope can preemptively abort transactions to avoid buggy or malicious applications from hoarding resources within the storage subsystem. When a transaction is preemptively aborted, any reads, writes, or control calls issued within it will return error codes, except for *endTX*, which will return false, and *abortTX*.

Transactions can be nested – i.e., a *beginTX*/*endTX* pair can have other pairs nested within it – with the simple semantics that the internal transactions are ignored. A nested *beginTX* does not establish a new snapshot, and a nested *endTX* always succeeds without changing the persistent state of the system. A nested *abortTX* causes any further activity in the transaction to return error codes until all the enclosing *abortTX*/*endTX* have been called. This behavior is important for allowing storage systems to expose transactions to end-user applications. In the example of the filesystem, if an end-user application invokes *beginTX* (either directly on Isotope or through a filesystem-provided API) before calling the *setattr* function in Figure 2 multiple times, the internal transactions within each *setattr* call are ignored and the entire ensemble of operations will commit or abort.

```
isofs_inode_num ino;
unsigned char *buf;
//allocate buf, set ino to parameter
...
int blknum = inode_to_block(ino);
txbegin:
beginTX();
if(!read(blknum, buf)){
        abortTX();
        return EIO;
}
mark_accessed(blknum, off, sizeof(inode));
//update attributes
...
if(!write(blknum, buf)){
        abortTX();
        return EIO;
}
mark_accessed(blknum, off, sizeof(inode));
if(!endTX()) goto txbegin;
```

Figure 2: Example application: *setattr* code for a filesystem built over Isotope.

3.1 Composability

As stated earlier, a primary benefit of a transactional block store is its obliviousness to the structure of the software stack running above it, which can range from a single-threaded application to a composition of multithreaded application code, library storage systems, out-of-process daemons and kernel modules. The Isotope API is designed to allow block-level transactions to span arbitrary compositions of different types of software modules. We describe some of these composition patterns in the context of a simple photo storage application called ImgStore, which stores photos and their associated metadata in a key-value store.

In the simplest case, ImgStore can store images and various kinds of metadata as key-value pairs in IsoHT, which in turn is built over a Isotope volume using transactions. Here, a single transaction-oblivious application (ImgStore) runs over a single transaction-aware library-based storage system (IsoHT).

Cross-Layer: ImgStore may want to atomically update multiple key-value pairs in IsoHT; for example, when a user is tagged in a photo, ImgStore may want to update a photo-to-user mapping as well as a user-to-photo mapping, stored under two different keys. To do so, ImgStore can encapsulate calls to IsoHT within Isotope *beginTX*/*endTX* calls, leveraging nested transactions.

Cross-Thread: In the simplest case, ImgStore executes each transaction within a single thread. However, if ImgStore is built using an event-driven library that requires transactions to execute across different threads, it can use the *releaseTX*/*takeoverTX* calls.

Cross-Library: ImgStore may find that IsoHT works well for certain kinds of accesses (e.g., retrieving a specific image), but not for others such as range queries (e.g., finding photos taken between March 4 and May

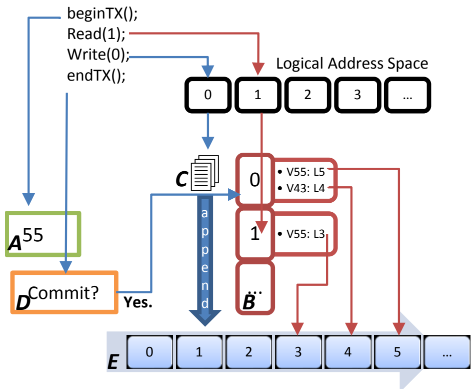


Figure 3: Isotope consists of (A) a timestamp counter, (B) a multiversion index, (C) a write buffer, (D) a decision algorithm, and (E) a persistent log.

10, 2015). Accordingly, it may want to spread its state across two different library key-value stores, one based on a hashtable (IsoHT) and another on a B-tree (IsoBT) for efficient range queries. When a photo is added to the system, ImgStore can transactionally call *put* operations on both stores. This requires the key-value stores to run over different partitions on the same Isotope volume.

Cross-Process: For various reasons, ImgStore may want to run IsoHT in a separate process and access it via an IPC mechanism; for example, to share it with other applications on the same machine, or to isolate failures in different codebases. To do so, ImgStore has to call *releaseTX* and pass the returned transaction handle via IPC to IsoHT, which then calls *takeoverTX*. This requires IsoHT to expose a transaction-aware IPC interface for calls that occur within a transactional context.

4 Design and Implementation

Figure 3 shows the major components of the Isotope design. Isotope internally implements an in-memory multiversion index (*B* in the figure) over a persistent log (*E*). Versioning is provided by a timestamp counter (*A*) which determines the snapshot seen by a transaction as well as its commit timestamp. This commit timestamp is used by a decision algorithm (*D*) to determine if the transaction commits or not. Writes issued within a transaction are buffered (*C*) during its execution, and flushed to the log if the transaction commits. We now describe the interaction of these components.

When the application calls *beginTX*, Isotope creates an in-memory intention record for the speculative transaction: a simple data structure with a start timestamp and a read/write-set. Each entry in the read/write-set consists of a block address, a bitmap that tracks the accessed status of smaller fixed-size chunks or fragments within the block (by default, the fragment size is 16 bytes, resulting in a 256-bit bitmap for each 4KB block), and an additional 4KB payload only in the write-set. These bitmaps are never written persistently and are only maintained in-memory for currently executing transactions. After creating the intention record, the *beginTX* call sets its start timestamp to the current value of the timestamp counter (*A* in Figure 3) without incrementing it.

Until *endTX* is called, the transaction executes speculatively against the (potentially stale) snapshot, without any effect on the shared or persistent state of the system. Writes update the write-set and are buffered in-memory (*C* in Figure 3) without issuing any I/O. A transaction can read its own buffered writes, but all other reads within the transaction are served from the snapshot corresponding to the start timestamp using the multiversion index (*B* in Figure 3). The *mark_accessed* call modifies the bitmap for a previously read or written block to indicate which bits the application actually touched. Multiple *mark_accessed* calls have a cumulative effect on the bitmap. At any point, the transaction can be preemptively aborted by Isotope simply by discarding its intention record and buffered writes. Subsequent reads, writes, and *endTX* calls will be unable to find the record and return an error code to the application.

All the action happens on the *endTX* call, which consists of two distinct phases: *deciding* the commit/abort status of the transaction, and *applying* the transaction (if it commits) to the state of the logical address space. Regardless of how it performs these two phases, the first action taken by *endTX* is to assign the transaction a commit timestamp by reading and incrementing the global counter. The commit timestamp of the transaction is used to make the commit decision, and is also used as the version number for all the writes within the transaction if it commits. We use the terms timestamp and version number interchangeably in the following text.

4.1 Deciding Transactions

To determine whether the transaction commits or aborts, *endTX* must detect the existence of conflicting transactions. The isolation guarantee provided – strict serializability or snapshot isolation – depends on what constitutes a conflicting transaction. We first consider a simple strawman scheme that provides strict serializability and implements conflict detection as a function of the multiversion index. Here, transactions are processed in commit timestamp order, and for each transaction the multiversion index is consulted to check if any of the logical blocks in its read-set has a version number greater than the current transaction's start timestamp. In other words, we check whether any of the blocks read by the transaction has been updated since it was read.

This scheme is simple, but suffers from a major drawback: the granularity of the multiversion index has to match the granularity of conflict detection. For example, if we want to check for conflicts at 16-byte grain, the index has to track version numbers at 16-byte grain as well; this blows up the size of the in-memory index by

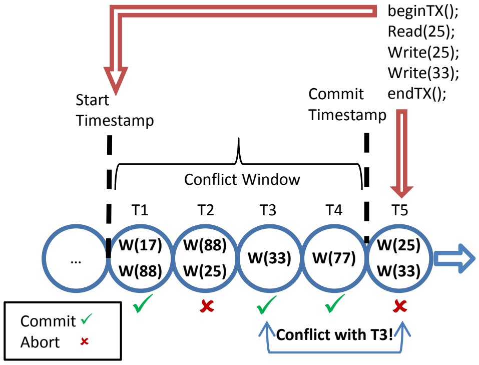


Figure 4: Conflict detection under snapshot isolation: a transaction commits if no other committed transaction in its conflict window has an overlapping write-set.

256X compared to a conventional block-granular index. As a result, this scheme is not well-suited for fine-grained conflict detection.

To perform fine-grained conflict detection while avoiding this blow-up in the size of the index, Isotope instead implements conflict detection as a function over the temporal stream of prior transactions (see Figure 4). Concretely, each transaction has a conflict window of prior transactions between its start timestamp and its commit timestamp.

- For strict serializability, the transaction T aborts if any committed transaction in its conflict window modified an address that T *read*; else, T commits.
- For snapshot isolation, the transaction T aborts if any committed transaction in its conflict window modified an address that T *wrote*; else, T commits.

In either case, the commit/abort status of a transaction is a function of a window of transactions immediately preceding it in commit timestamp order.

When *endTX* is called on T, a pointer to its intention record is inserted into the slot corresponding to its commit timestamp in an in-memory array. Since the counter assigns contiguous timestamps, this array has no holes; each slot is eventually occupied by a transaction. At this point, we do not yet know the commit/abort status of T and have not issued any write I/O, but we have a start timestamp and a commit timestamp for it. Each slot is guarded by its own lock.

To decide if T commits or aborts, we simply look at its conflict window of transactions in the in-memory array (i.e., the transactions between its start and commit timestamps). We can decide T's status once all these transactions have decided. T commits if each transaction in the window either aborts or has no overlap between its read/write-set and T's read/write-set (depending on the transactional semantics). Since each read/write-set stores fine-grained information about which fragments of the block are accessed, this scheme provides fine-grained conflict detection without increasing the size of the multiversion index.

Defining the commit/abort decision for a transaction as a function of other transactions is a strategy as old as optimistic concurrency control itself [35], but choosing an appropriate implementation is non-trivial. Like us, Bernstein et al. [48] formulate the commit/abort decision for distributed transactions in the Hyder system as a function of a conflict window over a totally ordered stream of transaction intentions. Unlike us, they explicitly make a choice to use the spatial state of the system (i.e., the index) to decide transactions. A number of factors drive our choice in the opposite direction: we need to support writes at arbitrary granularity (e.g., an inode) without increasing index size; our intention log is a local in-memory array and not distributed or shared across the network, drastically reducing the size of the conflict window; and checking for conflicts using read/write-sets is easy since our index is a simple address space.

4.2 Applying Transactions

If the outcome of the decision phase is commit, *endTX* proceeds to apply the transaction to the logical address space. The first step in this process is to append the writes within the transaction to the persistent log. This step can be executed in parallel for multiple transactions, as soon as each one's decision is known, since the existence and order of writes on the log signifies nothing: the multiversion index still points to older entries in the log. The second step involves changing the multiversion index to point to the new entries. Once the index has been changed, the transaction can be acknowledged and its effects are visible.

One complication is that this protocol introduces a lost update anomaly. Consider a transaction that reads a block (say an allocation bitmap in a filesystem), examines and changes the first bit, and writes it back. A second transaction reads the same block concurrently, examines and changes the last bit, and writes it back. Our conflict detection scheme will correctly allow both transactions to commit. However, each transaction will write its own version of the 4KB bitmap, omitting the other's modification; as a result, the transaction with the higher timestamp will destroy the earlier transaction's modification. To avoid such lost updates, the *endTX* call performs an additional step for each transaction before appending its buffered writes to the log. Once it knows that the current transaction can commit, it scans the conflict window and *merges* updates made by prior committed transactions to the blocks in its write-set.

4.3 Implementation Details

Isotope is implemented as an in-kernel software module in Linux 2.6.38; specifically, as a device mapper that exposes multiple physical block devices as a single virtual disk, at the same level of the stack as software RAID. Below, we discuss the details of this implementation.

Log implementation: Isotope implements the log (i.e., E in Figure 3) over a conventional address space with a counter marking the tail (and additional bookkeeping information for garbage collection, which we discuss shortly). From a correctness and functionality standpoint, Isotope is agnostic to how this address space is realized. For good performance, it requires an implementation that works well for a logging workload where writes are concentrated at the tail, while reads and garbage collection can occur at random locations in the body. A naive solution is to use a single physical disk (or a RAID-0 or RAID-10 array of disks), but garbage collection activity can hurt performance significantly by randomizing the disk arm. Replacing the disks with SSDs increases the cost-to-capacity ratio of the array without entirely eliminating the performance problem [58].

As a result, we use a design where a log is chained across multiple disks or SSDs (similar to Gecko [56]). Chaining the log across drives ensures that garbage collection – which occurs in the body of the log/chain – is separated from the first-class writes arriving at the tail drive of the log/chain. In addition, a commodity SSD is used as a read cache with an affinity for the tail drive of the chain, preventing application reads from disrupting write sequentiality at the tail drive. In essence, this design 'collars' the throughput of the log, pegging write throughput to the speed of a single drive, but simultaneously eliminating the throughput troughs caused by concurrent garbage collection and read activity.

Garbage collection (GC): Compared to conventional log-structured stores, GC is slightly complicated in Isotope by the need to maintain older versions of blocks. Isotope tracks the oldest start timestamp across all ongoing transactions and makes a best-effort attempt to not garbage collect versions newer than this timestamp. In the worst case, any non-current versions can be discarded without compromising safety, by first preemptively aborting any transactions reading from them. The application can simply retry its transactions, obtaining a new, current snapshot. This behavior is particularly useful for dealing with the effects of rogue transactions that are never terminated by the application. The alternative, which we did not implement, is to set a flag that preserves a running transaction's snapshot by blocking new writes if the log runs out of space; this may be required if it's more important for a long-running transaction to finish (e.g., if it's a critical backup) than for the system to be online for writes.

Caching: The *please_cache* call in Isotope allows the application to mark the blocks it wants cached in RAM. To implement caching, Isotope annotates the multiversion index with pointers to cached copies of block versions. This call is merely a hint and provides no guarantees to the application. In practice, our implementation uses a simple LRU scheme to cache a subset of the blocks if the application requests caching indiscriminately.

Index persistence: Thus far, we have described the multiversion index as an in-memory data structure pointing to entries on the log. Changes to the index have to be made persistent so that the state of the system can be reconstructed on failures. To obtain persistence and failure atomicity for these changes, we use a *metadata log*. The size of this log can be limited via periodic checkpoints.

A simple option is to store the metadata log on battery-backed RAM, or on newer technologies such as PCM or flash-backed RAM (e.g., Fusion-io's AutoCommit Memory [7]). In the absence of special hardware on our experimental testbed, we instead used a commodity SSD. Each transaction's description in the metadata log is quite compact (i.e., the logical block address and the physical log position of each write in it, and its commit timestamp). To avoid the slowdown and flash wear-out induced by logging each transaction separately as a synchronous page write, we batch multiple committed transactions together [25], delaying the final step of modifying the multiversion index and acknowledging the transaction to the application. We do not turn off the write cache on the SSD, relying on its ability to flush data on power failures using supercapacitors.

Memory overhead: A primary source of memory overhead in Isotope is the multiversion index. A single-version index that maps a 2TB logical address space to an 4TB physical address space can be implemented as a simple array that requires 2GB of RAM (i.e., half a billion 4-byte entries), which can be easily maintained in RAM on modern machines. Associating each address with a version (without supporting access to prior versions) doubles the space requirement to 4GB (assuming 4-byte timestamps), which is still feasible. However, multiversioned indices that allow access to past versions are more expensive, due to the fact that multiple versions need to be stored, and because a more complex data structure is required instead of an array with fixed-size values. These concerns are mitigated by the fact that Isotope is not designed to be a fully-fledged multiversion store; it only stores versions from the recent past, corresponding to the snapshots seen by executing transactions.

Accordingly, Isotope maintains a pair of indices: a single-version index in the form of a simple array and a multiversion index implemented as a hashtable. Each entry in the single-version index either contains a valid physical address if the block has only one valid, non-GC'ed version, a null value if the block has never been written, or a constant indicating the existence of multiple versions. If a transaction issues a read and encounters this constant, the multiversion index is consulted. An address is moved from the single-version index to the multiversion index when it goes from having one version to two; it is moved back to the single-version index when its older version(s) are garbage collected (as described earlier in this section).

The multiversion index consists of a hashtable that maps each logical address to a linked list of its existing versions, in timestamp order. Each entry contains forward and backward pointers, the logical address, the physical address, and the timestamp. A transaction walks this linked list to find the entry with the highest timestamp less than its snapshot timestamp. In addition, the entry also has a pointer to the in-memory cached copy, as described earlier. If an address is cached, the first single-version index is marked as having multiple versions even if it does not, forcing the transaction to look at the hashtable index and encounter the cached copy. In the future, we plan on applying recent work on compact, concurrent maps [28] to further reduce overhead.

Rogue Transactions: Another source of memory overhead in Isotope is the buffering of writes issued by in-progress transactions. Each write adds an entry to the write-set of the transaction containing the 4KB payload and a $\frac{4K}{C}$ bit bitmap, where C is the granularity of conflict detection (e.g., with 16-byte detection, the bitmap is 256 bits). Rogue transactions that issue a large number of writes are a concern, especially since transactions can be exposed to end-user applications. To handle this, Isotope provides a configuration parameter to set the maximum number of writes that can be issued by a transaction (set to 256 by default); beyond this, writes return an error code. Another parameter sets the maximum number of outstanding transactions a single process can have in-flight (also set to 256). Accordingly, the maximum memory a rogue process can use within Isotope for buffered writes is roughly 256MB. When a process is killed, its outstanding transactions are preemptively aborted.

Despite these safeguards, it is still possible for Isotope to run out of memory if many processes are launched concurrently and each spams the system with rogue, never-ending transactions. In the worst case, Isotope can always relieve memory pressure by preemptively aborting transactions. Another option which we considered is to flush writes to disk before they are committed; since the metadata index does not point to them, they won't be visible to other transactions. Given that the system is only expected to run out of memory in pathological cases where issuing I/O might worsen the situation, we didn't implement this scheme.

Note that the in-memory array that Isotope uses for conflict detection is not a major source of memory overhead; pointers to transaction intention records are inserted into this array in timestamp order only after the application calls *endTX*, at which point it has relinquished control to Isotope and cannot prolong the transaction. As a result, the lifetime of an entry in this array is short and limited to the duration of the *endTX* call.

Application	Original	Basic APIs	Optional APIs
	with locks	(lines modified)	(lines added)
IsoHT	591	591 (15)	617 (26)
IsoBT	1,229	1,229 (12)	1,246 (17)
IsoFS	997	997 (19)	1,022 (25)

Table 1: Lines of code for Isotope storage systems.

5 Isotope Applications

To illustrate the usability and performance of Isotope, we built four applications using Isotope transactions: IsoHT, a key-value store built over a persistent hashtable; IsoBT, a key-value store built over a persistent B-tree; IsoFS, a user-space POSIX filesystem; and ImgStore, an image storage service that stores images in IsoHT, and a secondary index in IsoBT. These applications implement each call in their respective public APIs by following a simple template that wraps the entire function in a single transaction, with a retry loop in case the transaction aborts due to a conflict (see Figure 2).

5.1 Transactional Key-Value Stores

Library-based or 'embedded' key-value stores (such as LevelDB or Berkeley DB) are typically built over persistent, on-disk data structures. We built two key-value stores called IsoHT and IsoBT, implemented over an on-disk hashtable and B-tree data structure, respectively. Both key-value stores support basic put/get operations on key-value pairs, while IsoBT additionally supports range queries. Each API call is implemented via a single transaction of block reads and writes to an Isotope volume.

We implemented IsoHT and IsoBT in three stages. First, we wrote code without Isotope transactions, using a global lock to guard the entire hashtable or B-tree. The resulting key-value stores are functional but slow, since all accesses are serialized by the single lock. Further, they do not provide failure atomicity: a crash in the middle of an operation can catastrophically violate data structure integrity.

In the second stage, we simply replaced the acquisitions/releases on the global lock with Isotope *beginTX/endTX/ abortTX* calls, without changing the overall number of lines of code. With this change, the key-value stores provide both fine-grained concurrency control (at block granularity) and failure atomicity. Finally, we added optional *mark_accessed* calls to obtain sub-block concurrency control, and *please_cache* calls to cache the data structures (e.g., the nodes of the B-tree, but not the values pointed to by them). Table 1 reports on the lines of code (LOC) counts at each stage for the two key-value stores.

5.2 Transactional Filesystem

IsoFS is a simple user-level filesystem built over Isotope accessible via FUSE [2], comprising 1K lines of C code. Its on-disk layout consists of distinct regions for storing inodes, data, and an allocation bitmap for each. Each inode has an indirect pointer and a double indirect pointer, both of which point to pages allocated from the data region. Each filesystem call (e.g., *setattr*, *lookup*, or *unlink*) uses a single transaction to access and modify multiple blocks. The only functionality implemented by IsoFS is the mapping and allocation of files and directories to blocks; atomicity, isolation, and durability are handled by Isotope.

IsoFS is stateless, caching neither data nor metadata across filesystem calls (i.e., across different transactions). Instead, IsoFS tells Isotope which blocks to cache in RAM. This idiom turned out to be surprisingly easy to use in the context of a filesystem; we ask Isotope to cache all bitmap blocks on startup, each inode block when an inode within it is allocated, and each data block that's allocated as an indirect or double indirect block. Like the key-value stores, IsoFS was implemented in three stages and required few extra lines of code to go from a global lock to using the Isotope API (see Table 1).

IsoFS trivially exposes transactions to end applications over files and directories. For example, a user might create a directory, move a file into it, edit the file, and rename the directory, only to abort the entire transactions and revert the filesystem to its earlier state. One implementation-related caveat is that we were unable to expose transactions to end applications of IsoFS via the FUSE interface, since FUSE decouples application threading from filesystem threading and does not provide any facility for explicitly transferring a transaction handle on each call. Accordingly, we can only expose transactions to the end application if IsoFS is used directly as a library within the application's process.

5.3 Experience

Composability: As we stated earlier, Isotope-based storage systems are trivially composable: a single transaction can encapsulate calls to IsoFS, IsoHT and IsoBT. To demonstrate the power of such composability, we built ImgStore, the image storage application described in Section 3. ImgStore stores images in IsoHT, using 64-bit IDs as keys. It then stores a secondary index in IsoBT, mapping dates to IDs. The implementation of ImgStore is trivially simple: to add an image, it creates a transaction to put the image in IsoHT, and then updates the secondary index in IsoBT. The result is a storage system that – in just 148 LOC – provides hashtable-like performance for gets while supporting range queries.

Isolation Levels: Isotope provides both strict serializability and snapshot isolation; our expectation was that developers would find it difficult to deal with the semantics of the latter. However, our experience with IsoFS, IsoHT and IsoBT showed otherwise. Snapshot isolation provides better performance than strict serializability, but introduces the write skew anomaly [16]: if two concurrent transactions read two blocks and each updates one of the blocks (but not the same one), they will both commit despite not being serializable in any order. The write skew anomaly is problematic for applications if a transaction is expected to maintain an integrity constraint that includes some block it does not write to (e.g., if the two blocks in the example have to sum to less than some constant). In the case of the storage systems we built, we did not encounter these kinds of constraints; for instance, no particular constraint holds between different bits on an allocation map. As a result, we found it relatively easy to reason about and rule out the write skew anomaly.

Randomization: Our initial implementations exhibited a high abort rate due to deterministic behavior across different transactions. For example, a simple algorithm for allocating a free page involved getting the first free bit from the allocation bitmap; as a result, multiple concurrent transactions interfered with each other by trying to allocate the same page. To reduce the abort rate, it was sufficient to remove the determinism in simple ways; for example, we assigned each thread a random start offset into the allocation bitmap.

6 Performance Evaluation

We evaluate Isotope on a machine with an Intel Xeon CPU with 24 hyper-threaded cores, 24GB RAM, three 10K RPM disks of 600GB each, an 128GB SSD for the OS and two other 240GB SSDs with SATA interfaces. In the following experiments, we used two primary configurations for Isotope's persistent log: a three-disk chained logging instance with a 32GB SSD read cache in front, and a 2-SSD chained logging instance. In some of the experiments, we compare against conventional systems running over RAID-0 configurations of 3 disks and 2 SSDs, respectively. In the chained logging configurations, all writes are logged to the single tail drive, while reads are mostly served by the other drives (and the SSD read cache for the disk-based configuration). The performance of this logging design under various workloads and during GC activity has been documented in [56]. In all our experiments, GC is running in the background and issuing I/Os to the drives in the body of the chain to compact segments, without disrupting the tail drive.

Our evaluation consists of two parts. First, we focus on the performance and overhead of Isotope, showing that it exploits fine-grained concurrency in workloads and provides high, stable throughput. Second, we show that Isotope applications – in addition to being simple and robust – are fast, efficient, and composable into larger applications.

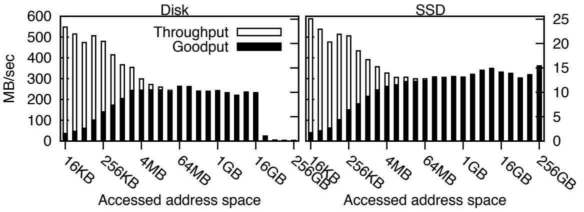


Figure 5: Without fine-grained conflict detection, Isotope performs well under low contention workloads.

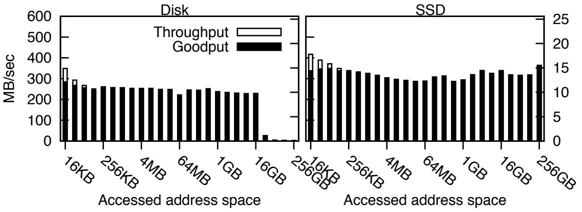


Figure 6: With fine-grained conflict detection, Isotope performs well even under high block-level contention.

6.1 Isotope Performance

To understand how Isotope performs depending on the concurrency present in the workload, we implemented a synthetic benchmark. The benchmark executes a simple type of transaction that reads three randomly chosen blocks, modifies a random 16-byte segment within each block (aligned on a 16-byte boundary), and writes them back. This benchmark performs identically with strict serializability and snapshot isolation, since the read-set exactly matches the write-set.

In the following experiments, we executed 64 instances of the micro benchmark concurrently, varying the size of the address space accessed by the instances to vary contention. The blocks are chosen from a specific prefix of the address space, which is a parameter to the benchmark; the longer this prefix, the bigger the fraction of the address space accessed by the benchmark, and the less skewed the workload. The two key metrics of interest are transaction goodput (measured as the number of successfully committed transactions per second, as well as the total number of bytes read or written per second by these transactions) and overall transaction throughput; their ratio is the commit rate of the system. Each data point in the following graphs is averaged across three runs; in all cases, the minimum and the maximum run were within 10% of the average.

Figure 5 shows the performance of this benchmark against Isotope without fine-grained conflict detection; i.e., the benchmark does not issue *mark_accessed* calls for the 16-byte segments it modifies. On the x-axis, we increase the fraction of the address space accessed by the benchmark. On the left y axis, we plot the rate at which data is read and written by transactions; on the right y-axis, we plot the number of transactions/sec. On both disk and SSD, transactional contention cripples performance on the left part of the graph: even though the benchmark attempts to commit thousands of transactions/sec, all of them access a small number of blocks, leading to low goodput. Note that overall transaction throughput is very high when the commit rate is low: aborts are cheap and do not result in storage I/O.

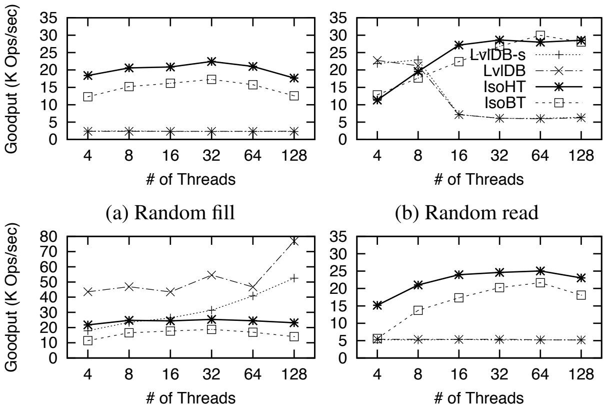
Conversely, disk contention hurts performance on the right side of Figure 5-Left: since the blocks read by each transaction are distributed widely across the address space, the 32GB SSD read cache is ineffective in serving reads and the disk arm is randomized and seeking constantly. As a result, the system provides very few transactions per second (though with a high commit rate). In the middle of the graph is a sweet spot where Isotope saturates the disk at roughly 120 MB/s of writes, where the blocks accessed are concentrated enough for reads to be cacheable in the SSD (which supplies 120 MB/s of reads, or 30K 4KB IOPS), while distributed enough for writes to not trigger frequent conflicts.

We can improve performance on the left side of the graphs in Figure 5 via fine-grained conflict detection. In Figure 6, the benchmark issues *mark_accessed* calls to tell Isotope which 16-byte fragment it is modifying. The result is high, stable goodput even when all transactions are accessing a small number of blocks, since there is enough fragment-level concurrency in the system to ensure a high commit rate. Using the same experiment but with smaller and larger data access and conflict detection granularities than 16 bytes showed similar trends. Isotope's conflict detection was not CPU-intensive: we observed an average CPU utilization of 5.96% without fine-grained conflict detection, and 6.17% with it.

6.2 Isotope Application Performance

As described earlier, we implemented two key-value stores over Isotope: IsoHT using a hashtable index and IsoBT using a B-tree index, respectively. IsoBT exposes a fully functional LevelDB API to end applications; IsoHT does the same minus range queries. To evaluate these systems, we used the LevelDB benchmark [5] as well as the YCSB [21] benchmark. We ran the fill-random, read-random, and delete-random workloads of the LevelDB benchmark and YCSB workload-A traces (50% reads and 50% updates following a zipf distribution on keys). All these experiments are on the 2-SSD configuration of Isotope. For comparison, we ran LevelDB on a RAID-0 array of the two SSDs, in both synchronous mode ('LvlDB-s') and asynchronous mode ('LvlDB'). LevelDB was set to use no compression and the default write cache size of 8MB. For all the workloads, we used a value size of 8KB and varied the number of threads issuing requests from 4 to 128. Results with different value sizes (from 4KB to 32KB) showed similar trends.

For operations involving writes (Figure 7-(a), (c), and (d)), IsoHT and IsoBT goodput increases with the num-



(a) Random fill

(b) Random read

(c) Random delete

(d) YCSB workload-A

Figure 7: IsoHT and IsoBT outperform LevelDB for data operations while providing stronger consistency.

ber of threads, but dips slightly beyond 64 threads due to an increased transaction conflict rate. For the read workload (Figure 7-(b)), throughput increases until the underlying SSDs are saturated. Overall, IsoHT has higher goodput than IsoBT, since it touches fewer metadata blocks per operation. We ran these experiments with Isotope providing snapshot isolation, since it performed better for certain workloads and gave sufficiently strong semantics for building the key-value stores. With strict serializability, for instance, the fill workload showed nearly identical performance, whereas the delete workload ran up to 25% slower.

LevelDB's performance is low for fill operations due to sorting and multi-level merging (Figure 7-(a)), and its read performance degrades as the number of concurrent threads increases because of the CPU contention in the skip list, cache thrashing, and internal merging operations (Figure 7-(b)). Still, LevelDB's delete is very efficient because it only involves appending a small delete intention record to a log, whereas IsoBT/IsoHT has to update a full 4KB block per delete (Figure 7-(c)).

The point of this experiment is not to show IsoHT/IsoBT is better than LevelDB, which has a different internal design and is optimized for specific workloads such as sequential reads and bulk writes. Rather, it shows that systems built over Isotope with little effort can provide equivalent or better performance than an existing system that implements its own concurrency control and failure atomicity logic.

6.2.1 Composability

To evaluate the composability of Isotope-based storage systems, we ran the YCSB workload on ImgStore, our image storage application built over IsoHT and IsoBT. In our experiments, ImgStore transactionally stored a 16KB payload (corresponding to an image) in IsoHT and a small date-to-ID mapping in IsoBT. To capture the various ways in which Isotope storage systems can be composed (see Section 3), we implemented several versions of ImgStore: cross-library, where ImgStore accesses the two key-value stores as in-process libraries, with each transaction executing within a single user-space thread; cross-thread, where ImgStore accesses each key-value store using a separate thread, and requires transactions to execute across them; and cross-process, where each key-value store executes within its own process and is accessed by ImgStore via socket-based IPC. Figure 8 shows the resulting performance for all three versions. It shows that the cost of the extra *takeoverTX*/*releaseTX* calls required for cross-thread transactions is negligible. As one might expect, cross-process transactions are slower due to the extra IPC overhead. Additionally, ImgStore exhibits less concurrency than IsoHT or IsoBT (peaking at 32 threads), since each composite transaction conflicts if either of its constituent transactions conflict.

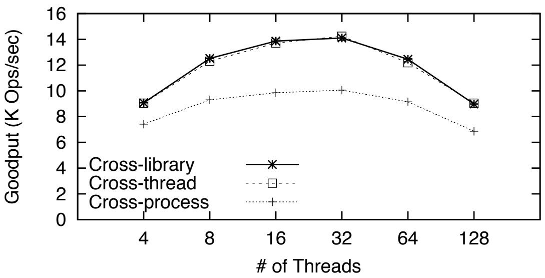


Figure 8: YCSB over different compositions of IsoBT and IsoHT.

6.2.2 Filesystem Performance

Next, we compare the end-to-end performance of IsoFS running over Isotope using the IOZone [4] write/rewrite benchmark with 8 threads. Each thread writes to its own file using a 16KB record size until the file size reaches 256MB; it then rewrites the entire file sequentially; and then rewrites it randomly. We ran this workload against IsoFS running over Isotope, which converted each 16KB write into a transaction involving four 4KB Isotope writes, along with metadata writes. We also ran ext2 and ext3 over Isotope; these issued solitary, non-transactional reads and writes, which were interpreted by Isotope as singleton transactions (in effect, Isotope operated as a conventional log-structured block store, so that ext2 and ext3 are not penalized for random I/Os). We ran ext3 in 'ordered' mode, where metadata is journaled but file contents are not.

Figure 9 plots the throughput observed by IOZone: on disk, IsoFS matches or slightly outperforms ext2 and ext3, saturating the tail disk on the chain. On SSD, IsoFS is faster than ext2 and ext3 for initial writes, but is bottlenecked by FUSE on rewrites. When we ran IsoFS directly using a user-space benchmark that mimics IOZone ('IsoFS-lib'), throughput improved to over 415MB/s. A

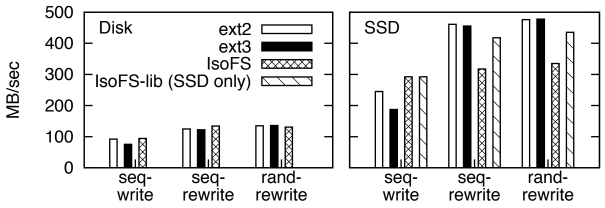


Figure 9: IOZone over IsoFS and ext2/ext3.

secondary point made by this graph is that Isotope does not slow down applications that do not use its transactional features (the high performance is mainly due to the underlying logging scheme, but ext2 and ext3 still saturate disk and SSD for rewrites), satisfying a key condition for pushing functionality down the stack [49].

7 Related Work

The idea of transactional atomicity for multi-block writes was first proposed in Mime [18], a log-structured storage system that provided atomic multi-sector writes. Over the years, multiple other projects have proposed block-level or page-level atomicity: the Logical Disk [24] in 1993, Stasis [54] in 2006, TxFlash [46] in 2008, and MARS [19] in 2013. RVM [51] and Rio Vista [36] proposed atomicity over a persistent memory abstraction. All these systems explicitly stopped short of providing full transactional semantics, relying on higher layers to implement isolation. To the best of our knowledge, no existing single-machine system has implemented transactional isolation at the block level, or indeed any concurrency control guarantee beyond linearizability.

On the other hand, distributed filesystems have often relied on the underlying storage layer to provide concurrency control. Boxwood [37], Sinfonia [12], and CalvinFS [62] presented simple NFS designs that leveraged transactions over distributed implementations of high-level data structures and a shared address space. Transaction isolation has been proposed for shared block storage accessed over a network [13] and for key-value stores [60]. Isotope can be viewed as an extension of similar ideas to single-machine, multi-core systems that does not require consensus or distributed rollback protocols. Our single-machine IsoFS implementation has much in common with the Boxwood, Sinfonia, and CalvinFS NFS implementations which ran against clusters of storage servers.

Isotope also fits into a larger body of work on smart single-machine block devices, starting with Loge [27] and including HP AutoRAID [65]. Some of this work has focused on making block devices smarter without changing the interface [57], while other work has looked at augmenting the block interface [18, 64, 30], modifying it [67], and even replacing it with an object-based interface [38]. In a distributed context, Parallax [39] and Strata [23] provide virtual disks on storage clusters. A number of filesystems are multiversion, starting with WAFL [33], and including many others [50, 41, 22]. Underlying these systems is research on multiversion data structures [26]. Less common are multiversion block stores such as Clotho [29] and Venti [47].

A number of filesystems have been built over a full-fledged database. Inversion [43] is a conventional filesystem built over the POSTGRES database, while Amino [66] is a transactional filesystem (i.e., exposing transactions to users) built over Berkeley DB. WinFS [10] was built over a relational engine derived from SQL Server. This route requires storage system developers to adopt a complex interface – one that does not match or expose the underlying grain of the hardware – in order to obtain benefits such as isolation and atomicity. In contrast, Isotope retains the simple block storage interface while providing isolation and atomicity.

TxOS [45] is a transactional operating system that provides ACID semantics over syscalls, include file accesses. In contrast, Isotope is largely OS-agnostic and can be ported easily to commodity operating systems, or even implemented under the OS as a hardware device. In addition, Isotope supports the easy creation of new systems such as key-value stores and filesystems that run directly over block storage.

Isotope is also related to the large body of work on software transactional memory (STM) [55, 32] systems, which typically provide isolation but not durability or atomicity. Recent work has leveraged new NVRAM technologies to add durability to the STM abstraction: Mnemosyne [63] and NV-Heaps [20] with PCM and Hathi [52] with commodity SSDs. In contrast, Isotope aims for transactional secondary storage, rather than transactional main-memory.

8 Conclusion

We described Isotope, a transactional block store that provides isolation in addition to atomicity and durability. We showed that isolation can be implemented efficiently within the block layer, leveraging the inherent multi-versioning of log-structured block stores and application-provided hints for fine-grained conflict detection. Isotope-based systems are simple and fast, while obtaining database-strength guarantees on failure atomicity, durability, and consistency. They are also composable, allowing application-initiated transactions to span multiple storage systems and different abstractions such as files and key-value pairs.

Acknowledgments

This work is partially funded and supported by a SLOAN Research Fellowship received by Hakim Weatherspoon, DARPA MRC and CSSG (D11AP00266) and NSF (1422544, 1053757, 0424422, 1151268, 1047540). We would like to thank our shepherd, Sage Weil, and the anonymous reviewers for their comments.

References

[1] fcntl man page.

[2] Filesystem in userspace. http://fuse.sourceforge.net.

[3] Fusion-io. www.fusionio.com.

[4] Iozone filesystem benchmark. http://www.iozone.org.

[5] LevelDB benchmarks. http://leveldb.googlecode.com/svn/trunk/doc/benchmark.html.

[6] SanDisk Fusion-io atomic multi-block writes. http://www.sandisk.com/assets/docs/accelerate-myql-open-source-databases-with-sandisk-nvmfs-and-fusion-iomemory-sx300-application-accelerators.pdf.

[7] SanDisk Fusion-io auto-commit memory. http://web.sandisk.com/assets/white-papers/MySQL_High-Speed_Transaction_Logging.pdf.

[8] Seagate kinetic open storage platform. http://www.seagate.com/solutions/cloud/data-center-cloud/platforms/.

[9] Storage spaces. http://technet.microsoft.com/en-us/library/hh831739.aspx.

[10] Winfs. http://blogs.msdn.com/b/winfs/.

[11] A. Aghayev and P. Desnoyers. Skylight a window on shingled disk operation. In *USENIX FAST*, pages 135–149, 2015.

[12] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *ACM SIGOPS Operating Systems Review*, 41(6):159–174, 2007.

[13] K. Amiri, G. A. Gibson, and R. Golding. Highly concurrent shared storage. In *IEEE ICDCS*, pages 298–307, 2000.

[14] A. Badam and V. S. Pai. SSDAlloc: hybrid SSD/RAM memory management made easy. In *USENIX NSDI*, 2011.

[15] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. CORFU: A shared log design for flash clusters. In *USENIX NSDI*, pages 1–14, 2012.

[16] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[17] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[18] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical report, HPL-CSP-92-9, Hewlett-Packard Laboratories, 1992.

[19] J. Coburn, T. Bunker, R. K. Gupta, and S. Swanson. From ARIES to MARS: Reengineering transaction management for next-generation, solid-state drives. In *SOSP*, 2013.

[20] 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. *ACM SIGARCH Computer Architecture News*, 39(1):105–118, 2011.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *ACM SoCC*, pages 143–154, 2010.

[22] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: A user-level versioning file system for linux. In *USENIX ATC, FREENIX Track*, 2004.

[23] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield. Strata: scalable high-performance storage on virtualized non-volatile memory. In *USENIX FAST*, pages 17–31, 2014.

[24] W. De Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. *ACM SIGOPS Operating Systems Review*, 27(5):15–28, 1993.

[25] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *ACM SIGMOD*, pages 1–8, 1984.

[26] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121. ACM, 1986.

[27] R. M. English and A. A. Stepanov. Loge: a self-organizing disk controller. In *USENIX Winter*, 1992.

[28] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. *USENIX NSDI*, 2013.

[29] M. Flouris and A. Bilas. Clotho: Transparent Data Versioning at the Block I/O Level. In *MSST*, pages 315–328, 2004.

[30] G. R. Ganger. *Blurring the line between OSes and storage devices*. School of Computer Science, Carnegie Mellon University, 2001.

[31] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.

[32] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2010.

[33] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an nfs file server appliance. In *USENIX Winter*, volume 94, 1994.

[34] J. Jose, M. Banikazemi, W. Belluomini, C. Murthy, and D. K. Panda. Metadata persistence using storage class memory: experiences with flash-backed dram. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, page 3. ACM, 2013.

[35] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[36] D. E. Lowell and P. M. Chen. Free transactions with rio vista. *ACM SIGOPS Operating Systems Review*, 31(5):92–101, 1997.

[37] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, volume 4, pages 8–8, 2004.

[38] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *Communications Magazine, IEEE*, 41(8):84–90, 2003.

[39] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: virtual disks for virtual machines. *ACM SIGOPS Operating Systems Review*, 42(4):41–54, 2008.

[40] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[41] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. In *USENIX FAST*, 2004.

[42] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *USENIX OSDI*, 2012.

[43] M. A. Olson. The design and implementation of the inversion file system. In *USENIX Winter*, pages 205–218, 1993.

[44] A. Pennarun. Everything you never wanted to know about file locking. `http://apenwarr.ca/log/?m=201012#13`.

[45] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP*, pages 161–176. ACM, 2009.

[46] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *USENIX OSDI*, 2008.

[47] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *USENIX FAST*, 2002.

[48] C. Reid, P. A. Bernstein, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[49] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[50] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. *ACM SIGOPS Operating Systems Review*, 33(5):110–123, 1999.

[51] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems (TOCS)*, 12(1):33–57, 1994.

[52] M. Saxena, M. A. Shah, S. Harizopoulos, M. M. Swift, and A. Merchant. Hathi: durable transactions for memory using flash. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 33–38. ACM, 2012.

[53] M. Saxena, M. M. Swift, and Y. Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *ACM EuroSys*, pages 267–280, 2012.

[54] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *USENIX OSDI*, 2006.

[55] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[56] J.-Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon. Gecko: Contention-oblivious disk arrays for cloud storage. In *USENIX FAST*, pages 213–225, 2013.

[57] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *USENIX FAST*, 2003.

[58] D. Skourtis, D. Achlioptas, N. Watkins, C. Maltzahn, and S. Brandt. Flash on rails: consistent flash performance through redundancy. In *USENIX ATC*, pages 463–474, 2014.

[59] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD Lifetimes with Disk-Based Write Caches. In *USENIX FAST*, 2010.

[60] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *ACM SOSP*, 2011.

[61] L. Stein. Stupid File Systems Are Better. In *HotOS*, 2005.

[62] A. Thomson and D. J. Abadi. Calvinfs: Consistent wan replication and scalable metadata management for distributed file systems. In *USENIX FAST*, pages 1–14, 2015.

[63] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, 39(1):91–104, 2011.

[64] R. Y. Wang, T. E. Anderson, and D. A. Patterson. Virtual log based file systems for a programmable disk. *Operating systems review*, 33:29–44, 1998.

[65] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The hp autoraid hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, 1996.

[66] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending acid semantics to the file system. *ACM Transactions on Storage (TOS)*, 3(2):4, 2007.

[67] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *USENIX FAST*, 2012.

BTrDB: Optimizing Storage System Design for Timeseries Processing

Michael P Andersen
University of California, Berkeley
m.andersen@cs.berkeley.edu

David E. Culler
University of California, Berkeley
culler@cs.berkeley.edu

Abstract

The increase in high-precision, high-sample-rate telemetry timeseries poses a problem for existing timeseries databases which can neither cope with the throughput demands of these streams nor provide the necessary primitives for effective analysis of them. We present a novel abstraction for telemetry timeseries data and a data structure for providing this abstraction: a time-partitioning version-annotated copy-on-write tree. An implementation in Go is shown to outperform existing solutions, demonstrating a throughput of 53 million inserted values per second and 119 million queried values per second on a four-node cluster. The system achieves a 2.9x compression ratio and satisfies statistical queries spanning a year of data in under 200ms, as demonstrated on a year-long production deployment storing 2.1 trillion data points. The principles and design of this database are generally applicable to a large variety of timeseries types and represent a significant advance in the development of technology for the Internet of Things.

1 Introduction

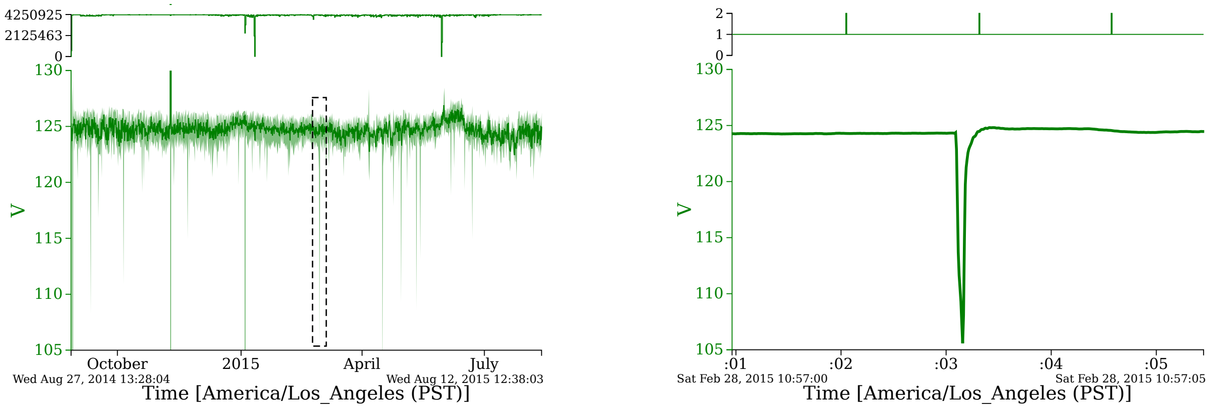
A new class of distributed system with unique storage requirements is becoming increasingly important with the rise of the Internet of Things. It involves collecting, distilling and analyzing – in near real-time and historically – time-correlated telemetry from a large number of high-precision networked sensors with fairly high sample rates. This scenario occurs in monitoring the internal dynamics of electric grids, building systems, industrial processes, vehicles, structural health, and so on. Often, it provides situational awareness of complex infrastructure. It has substantially different characteristics from either user-facing focus and click data, which is pervasive in modern web applications, smart metering data, which collects 15-minute interval data from many millions of meters, or one-shot dedicated instrument logging.

We focus on one such source of telemetry – microsynchophasors, or uPMUs. These are a new generation of small, comparatively cheap and extremely high-precision power meters that are to be deployed in the distribution tier of the electrical grid, possibly in the millions. In the distributed system shown in Figure 2, each device produces 12 streams of 120 Hz high-precision values with timestamps accurate to 100 ns (the limit of GPS). Motivated by the falling cost of such data sources, we set out to construct a system supporting more than 1000 of these devices per backing server – more than 1.4 million inserted points per second, and several times this in expected reads and writes from analytics. Furthermore, this telemetry frequently arrives out of order, delayed and duplicated. In the face of these characteristics, the storage system must guarantee the consistency of not only the raw streams, but all analytics derived from them. Additionally, fast response times are important for queries across time scales from years to milliseconds.

These demands exceed the capabilities of current timeseries data stores. Popular systems, such as KairosDB [15], OpenTSDB [20] or Druid [7], were designed for complex multi-dimensional data at low sample rates and, as such, suffer from inadequate throughput and timestamp resolution for these telemetry streams, which have comparatively simple data and queries based on time extents. These databases all advertise reads and writes of far less than 1 million values per second per server, often with order-of-arrival and duplication constraints, as detailed in Section 2.

As a solution to this problem, a novel, ground-up, use-inspired time-series database abstraction – BTrDB – was constructed to provide both higher sustained throughput for raw inserts and queries, as well as advanced primitives that accelerate the analysis of the expected 44 quadrillion datapoints per year *per server*.

The core of this solution is a new abstraction for time series telemetry data (Section 3) and a data structure that provides this abstraction: a *time-partitioning, multi-*



(a) Statistical summary of a year of voltage data to locate voltage sags, representing 50 billion readings, with min, mean, and max shown. The data density (the plot above the main plot) is 4.2 million points per pixel column.

(b) The voltage sag outlined in (a) plotted over 5 seconds. The data density (the plot above the main plot) is roughly one underlying data point per pixel column

Figure 1: Locating interesting events in typical uPMU telemetry streams using statistical summaries

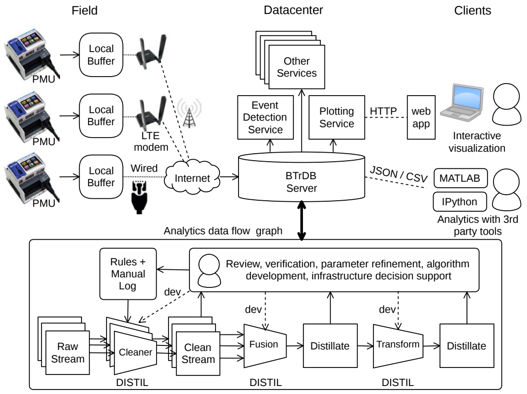


Figure 2: uPMU network storage and query processing system

resolution, version-annotated, copy-on-write tree, as detailed in Section 4. A design for a database using this data structure is presented in Section 5.

An open-source 4709-line Go implementation of BTrDB demonstrates the simplicity and efficacy of this method, achieving 53 million inserted values per second and 119 million queried values per second on a four node cluster with the necessary fault-tolerance and consistency guarantees. Furthermore, the novel analytical primitives allow the navigation of a year worth of data comprising billions of datapoints (e.g. Figure 1a) to locate and analyse a sub-second event (e.g. Figure 1b) using a sequence of statistical queries that complete in 100-200ms, a result not possible with current tools. This is discussed in Section 6.

2 Related work

Several databases support high-dimensionality time-series data, including OpenTSDB [20], InfluxDB [12], KairosDB [15] and Druid [7]. In terms of raw telemetry, these databases are all limited to millisecond-precision timestamps. This is insufficient to capture phase angle samples from uPMUs, which require sub-microsecond-precision timestamps. While all of these are capable of storing scalar values, they also support more advanced "event" data, and this comes at a cost. Druid advertises that "large production clusters" have reached "1M+ events per second" on "tens of thousands of cores." Published results for OpenTSDB show $<$ 1k operations per second per node [4][10]. MapR has shown OpenTSDB running on MapR-DB, modified to support batch inserts and demonstrated 27.5 million inserted values per second per node (bypassing parts of OpenTSDB) and 375 thousand reads per second per node [28][8]; unfortunately this performance is with 1 byte values and counter-derived second-precision timestamps, which somewhat undermines its utility [27].

A study evaluating OpenTSDB and KairosDB [10] with real PMU data showed that KairosDB significantly outperforms OpenTSDB, but only achieves 403,500 inserted values per second on a 36 node cluster. KairosDB gives an example of 133 k inserted values per second [14] using bulk insert in their documentation. Rabl et. al [24] performed an extensive benchmark comparing Project Voldemort [23], Redis [26], HBase [3], Cassandra [2], MySQL [21] and VoltDB [31][29]. Cassandra exhibited the highest throughput, inserting 230k records per second on a twelve node cluster. The records used were large (75 bytes), but even if optimistically normalised to the size of our records (16 bytes) it only yields roughly 1M inserts per second, or 89K inserts per second per node. The other five candidates exhibited lower throughput. Datastax [6] performed a similar benchmark [9] comparing MongoDB [19], Cassandra [2], HBase [3] and Couchbase [5]. Here too, Cassandra outperformed the competition obtaining 320k inserts/sec and 220k reads/sec on a 32 node cluster.

Recently, Facebook's in-memory Gorilla database [22] takes a similar approach to BTrDB - simplifying the data model to improve performance. Unfortunately, it has second-precision timestamps, does not permit out-of-order insertion and lacks accelerated aggregates.

In summary, we could find no databases capable of handling 1000 uPMUs per server node (1.4 million inserts/s per node and 5x that in reads), even without considering the requirements of the analytics. Even if existing databases *could* handle the raw throughput, and timestamp precision of the telemetry, they lack the ability to satisfy queries over large ranges of data efficiently. While many time series databases support aggregate queries, the computation requires on-the-fly iteration of the base data (e.g. OpenTSDB, Druid) - untenable at 50 billion samples per year per uPMU. Alternatively, some timeseries databases offer precomputed aggregates (e.g, InfluxDB, RespawnDB [4]), accelerating these queries, but they are unable to guarantee the consistency of the aggregates when data arrives out of order or is modified. The mechanisms to guarantee this consistency exist in most relational databases, but those fare far worse in terms of throughput.

Thus, we were motivated to investigate a clean slate design and implementation of a time-series database with the necessary capabilities – high throughput, fixed-response-time analytics irrespective of the underlying data size and eventual consistency in a graph of interdependent analytics despite out of order or duplicate data. This was approached in an integrated fashion from the block or file server on up. Our goal was to develop a multi-resolution storage and query engine for many higher bandwidth ($>$ 100 Hz) streams that provides the above functionality essentially "for free", in that it operates at the full line rate of the underlying network or storage infrastructure for affordable cluster sizes ($<$ 6 servers).

BTrDB has promising functionality and performance. On four large EC2 nodes it achieves over 119M queried values per second ($>$10GbE line rate) and over 53M inserted values per second of 8 byte time and 8 byte value pairs, while computing statistical aggregates. It returns results of 2K points summarizing anything from the raw values (9 ms) to 4 billion points (a year) in 100-250ms. It does this while maintaining the provenance of all computed values and consistency of a network of streams. The system storage overhead is negligible, with an all-included compression ratio of 2.9x – a significant improvement on existing compression techniques for synchrophasor data streams.

3 Time Series Data Abstraction

The fundamental abstraction provided by BTrDB is a consistent, write-once, ordered sequence of time-value pairs. Each stream is identified by a UUID. In typical uses, a substantial collection of metadata is associated with each stream. However, the nature of the metadata varies widely amongst uses and many good solutions exist for querying metadata to obtain a collection of streams. Thus, we separate the lookup (or directory) function entirely from the time series data store, identifying each stream solely by its UUID. All access is performed on a temporal segment of a version of a stream. All time stamps are in nanoseconds with no assumptions on sample regularity.

InsertValues(UUID, [(time, value)]) creates a new version of a stream with the given collection of (time,value) pairs inserted. Logically, the stream is maintained in time order. Most commonly, points are appended to the end of the stream, but this cannot be assumed: readings from a device may be delivered to the store out of order, duplicates may occur, holes may be backfilled and corrections may be made to old data – perhaps as a result of recalibration. These situations routinely occur in real world practice, but are rarely supported by timeseries databases. In BTrDB, each insertion of a collection of values creates a new version, leaving the old version unmodified. This allows new analyses to be performed on old versions of the data.

The most basic access method, **GetRange(UUID, StartTime, EndTime, Version) → (Version, [(Time, Value)])** retrieves all the data between two times in a given version of the stream. The 'latest' version can be indicated, thereby eliminating a call to **GetLatestVersion(UUID) → Version** to obtain the latest version for a stream prior to querying a range. The exact version number is returned along with the data to facilitate a repeatable query in future. BTrDB does not provide operations to resample the raw points in a stream on a particular schedule or to align raw samples across streams because performing these manipulations correctly ultimately depends on a semantic model of the data. Such operations are well supported by mathematical environments, such as Pandas [17], with appropriate control over interpolation methods and so on.

Although this operation is the only one provided by most historians, with trillions of points, it is of limited utility. It is used in the final step after having isolated an important window or in performing reports, such as disturbances over the past hour. Analyzing raw streams in their entirety is generally impractical; for example, each uPMU produces nearly 50 billion samples per year.

The following access methods are far more powerful for broad analytics and for incremental generation of computationally refined streams.

In visualizing or analyzing huge segments of data **GetStatisticalRange(UUID, StartTime, EndTime, Version, Resolution) → (Version, [(Time, Min, Mean, Max, Count)])** is used to retrieve statistical records between two times at a given temporal resolution. Each record covers $2^{resolution}$ nanoseconds. The start time and end time are on $2^{resolution}$ boundaries and result records are periodic in that time unit; thus summaries are aligned across streams. Unaligned windows can also be queried, with a marginal decrease in performance.

GetNearestValue(UUID, Time, Version, Direction) → (Version, (Time, Value)) locates the nearest point to a given time, either forwards or backwards. It is commonly used to obtain the 'current', or most recent to now, value of a stream of interest.

In practice, raw data streams feed into a graph of distillation processes in order to clean and filter the raw data and then combine the refined streams to produce useful data products, as illustrated in Figure 2. These distillers fire repeatedly, grab new data and compute output segments. In the presence of out of order arrival and loss, without support from the storage engine, it can be complex and costly to determine which input ranges have changed and which output extents need to be computed, or recomputed, to maintain consistency throughout the distillation pipeline.

To support this, **ComputeDiff(UUID, FromVersion, ToVersion, Resolution) → [(StartTime, EndTime)]** provides the time ranges that contain differences between the given versions. The size of the changeset returned can be limited by limiting the number of versions between **FromVersion** and **ToVersion** as each version has a maximum size. Each returned time range will be larger than $2^{resolution}$ nanoseconds, allowing the caller to optimize for batch size.

As utilities, **DeleteRange(UUID, StartTime, EndTime)**: create a new version of the stream with the given range deleted and **Flush(UUID)** ensure the given stream is flushed to replicated storage.

4 Time partitioned tree

To provide the abstraction described above, we use a *time-partitioning copy-on-write version-annotated k-ary tree*. As the primitives API provides queries based on time extents, the use of a tree that partitions time serves the role of an index by allowing rapid location of specific points in time. The base data points are stored in the leaves of the tree, and the depth of the tree is defined by the interval between data points. A uniformly sampled telemetry stream will have a fixed tree depth irrespective of how much data is in the tree. All trees logically represent a huge range of time (from $-2^{60}ns$ to $3*2^{60}ns$ as measured from the Unix epoch, or approximately 1933 to 2079 with nanosecond precision) with big holes at the front and back ends and smaller holes between each of the points. Figure 3 illustrates a time-partitioning tree for 16 ns. Note the hole between 8 and 12 ns.

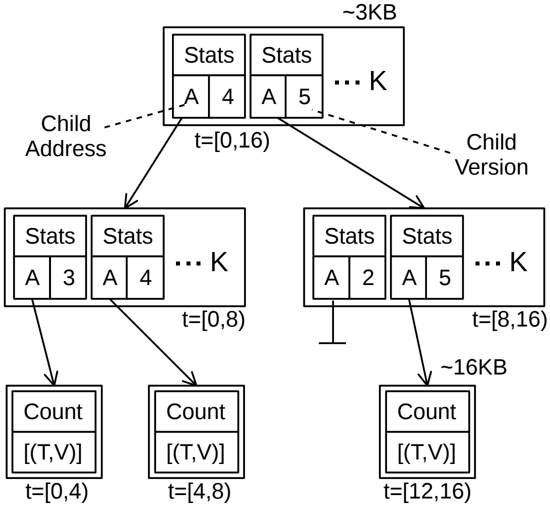


Figure 3: An example of a time-partitioning tree with version-annotated edges. Node sizes correspond to a K=64 implementation

To retain historic data, the tree is *copy on write*: each insert into the tree forms an overlay on the previous tree accessible via a new root node. Providing historic data queries in this way ensures that all versions of the tree require equal effort to query – unlike log replay mechanisms which introduce overheads proportional to how much data has changed or how old is the version that is being queried. Using the storage structure as the index ensures that queries to any version of the stream have an index to use, and reduces network round trips.

Each link in the tree is annotated with the version of the tree that introduced that link, also shown in Figure 3. A null child pointer with a nonzero version annotation implies the version is a deletion. The time extents that were modified between two versions of the tree can be walked by loading the tree corresponding to the later version, and descending into all nodes annotated with the start version or higher. The tree need only be walked to the depth of the desired difference resolution, thus *ComputeDiff()* returns its results without reading the raw data. This mechanism allows consumers of a stream to query and process new data, regardless of where the changes were made, without a full scan and with only 8 bytes of state maintenance required - the 'last version processed'.

Each internal node holds scalar summaries of the subtrees below it, along with the links to the subtrees. Statistical aggregates are computed as nodes are updated, following the modification or insertion of a leaf node. The statistics currently supported are *min*, *mean*, *max* and *count*, but any operation that uses intermediate results from the child subtrees without requiring iteration over the raw data can be used. Any associative operation meets this requirement.

This approach has several advantages over conventional discrete rollups. The summary calculation is free in terms of IO operations – the most expensive part of a distributed storage system. All data for the calculation is already in memory, and the internal node needs to be copied anyway, since it contains new child addresses. Summaries do increase the size of internal nodes, but even so, internal nodes are a tiny fraction of the total footprint ($< 0.3\%$ for a single version of a $K = 64$ k-ary tree). Observable statistics are guaranteed to be consistent with the underlying data, because failure during their calculation would prevent the root node from being written and the entire overlay would be unreachable.

When querying a stream for statistical records, the tree need only be traversed to the depth corresponding to the desired resolution, thus the response time is proportional to the number of returned records describing the temporal extent, not the length of the extent nor the number of datapoints within it. Records from disparate streams are aligned in time, so time-correlated anaylsis can proceed directly. For queries requiring specific non-power-of-two windows, the operation is still dramatically accelerated by using the precomputed statistics to fill in the middle of each window, only requiring a “drill down” on the side of each window, so that the effort to generate a window is again proportional to the log of the length of time it covers, not linear in the underlying data.

Although conceptually a binary tree, an implementation may trade increased query-time computation for decreased storage and IO operations by using a k-ary tree and performing just-in time computation of the statistical metrics for windows that lie between actual levels of the tree. If k is too large, however, the on-the-fly computation impacts increases the variability of statistical query latencies, as discussed in Section 6.3.

To allow fetching nodes from the tree in a single IO operation, all addresses used in the tree are “native” in that they are directly resolvable by the storage layer without needing a translation step. If an indirect address were used it would require either a costly remote lookup in a central map, or complex machinery to synchronize a locally stored map. Multiple servers can execute reads on the same stream at a time, so all servers require an up-to-date view of this mapping. Native addresses remove this problem entirely, but they require care to maintain, as discussed below.

The internal blocks have a base size of $2 \times 8 \times K$ for the child addresses and child pointer versions. On top of that, the statistics require $4 \times 8 \times K$ for *min*, *mean*, *max* and *count* making them 3KB in size for $K = 64$. The leaf nodes require 16 bytes per (time, value) pair, and a 16 byte length value. For $N_{leaf} = 1024$ they are 16KB big. Both of these blocks are compressed, as discussed below.

5 System design

The overall system design of BTrDB, shown in Figure 4, is integrally tied to the multi-resolution COW tree data structure described above, but also represents a family of trade-offs between complexity, performance and reliability. This design prioritizes simplicity first, performance second and then reliability, although it does all three extremely well. The ordering is the natural evolution of developing a database that may require frequent changes to match a dynamically changing problem domain and workload (simplicity leads to an easily modifiable design). Performance requirements originate from the unavoidable demands placed by the devices we are deploying and, as this system is used in production, reliability needs to be as high as possible, without sacrificing the other two goals.

The design consists of several modules: request handling, transaction coalescence, COW tree construction and merge, generation link, block processing, and block storage. The system follows the SEDA [34] paradigm with processing occuring in three resource control stages – request, write and storage – with queues capable of exerting backpressure decoupling them.

5.1 Request processing stage

At the front end, flows of insertion and query requests are received over multiple sockets, either binary or HTTP. Each stream is identified by UUID. Operations on many streams may arrive on a single socket and those for a particular stream may be distributed over multiple sockets. Inserts are collections of time-value pairs, but need not be in order.

Insert and query paths are essentially separate. Read requests are comparatively lightweight and are handled in a thread of the session manager. These construct and traverse a partial view of the COW tree, as described above, requesting blocks from the block store. The block store in turn requests blocks from a reliable storage provider (Ceph in our implementation) and a cache of recently used blocks. Read throttling is achieved by the storage stage limiting how many storage handles are given to the session thread to load blocks. Requests hitting the cache are only throttled by the socket output.

On the insert path, incoming data is demultiplexed into per-stream coalescence buffers by UUID. Session man-

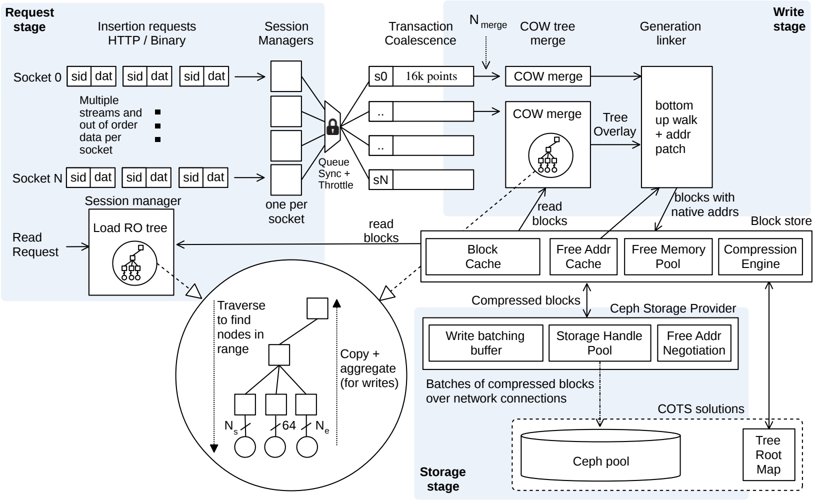


Figure 4: An overview of BTrDB showing the three SEDA-style stages, and composing modules

agers compete for a shortly held map lock and then grab a lock on the desired buffer. This sync point provides an important write-throttling mechanism, as discussed in Section 7.3. Each stream is buffered until either a certain time interval elapses or a certain number of points arrive, which triggers a commit by the write stage. These parameters can be adjusted according to target workload and platform, with the obvious trade-offs in stream update delay, number of streams, memory pressure, and ratio of tree and block overhead per version commit. These buffers need not be very big; we see excellent storage utilization in production where the buffers are configured for a maximum commit of 5 seconds or 16k points and the average commit size is 14400 points.

5.2 COW merge

A set of threads in the write stage pick up buffers awaiting commit and build a writable tree. This process is similar to the tree build done by a read request, except that all traversed nodes are modified as part of the merge, so must remain in memory. Copying existing nodes or creating new nodes requires a chunk of memory which is obtained from a **free pool** in the block store. At this point the newly created blocks have temporary addresses – these will be resolved by the linker later to obtain the index-free native addressing.

5.3 Block store

The block store allocates empty blocks, stores new blocks and fetches stored blocks. It also provides compression/decompression of storage blocks, and a cache. Empty blocks, used in tree merges, are satisfied primarily from the free pool to avoid allocations. After blocks are evicted from the block cache and are about to be garbage collected, they are inserted back into this pool.

Fields such as a block's address, UUID, resolution (tree depth) and time extent are useful for traversing the tree, but can be deduced from context when a block is read from disk, so are stripped before the block enters the compression engine.

The **block cache** holds all the blocks that pass through the block store with the least recently used blocks evicted first. It consumes a significant (tunable) portion of the memory footprint. Cache for a time series store may not seem an obvious win, other than for internal nodes in the COW tree, but it is extremely important for near-real-time analytics. As the majority of our read workload consists of processes waiting to consume any changes to a set of streams – data that just passed through the system – a cache of recently used blocks dramatically improves performance.

5.4 Compression engine

Part of the block store, the **compression engine** compresses the min, mean, max, count, address and version fields in internal nodes, as well as the time and value fields in leaf nodes. It uses a method we call *delta-delta* coding followed by Huffman coding using a fixed tree. Typical delta coding works by calculating the difference between every value in the sequence and storing that using variable-length symbols (as the delta is normally smaller than the absolute values [18]). Unfortunately with high-precision sensor data, this process does not work well because nanosecond timestamps produce very large deltas, and even linearly-changing values produce sequences of large, but similar, delta values.

In lower precision streams, long streams of identical deltas are typically removed with run-length encoding that removes sequences of identical deltas. Unfortunately noise in the lower bits of high precision sensor values prevents run-length encoding from successfully compacting the sequence of deltas. This noise, however, only adds a small jitter to the delta values. They are otherwise very similar. Delta-delta compression replaces run-length encoding and encodes each delta as the difference from the mean of a window of previous delta values. The result is a sequence of only the jitter values. Incidentally this works well for the addresses and version numbers, as they too are linearly increasing values with some jitter in the deltas. In the course of system development, we found that this algorithm produces better results, with a simpler implementation, than the residual coding in FLAC [13] which was the initial inspiration.

This method is lossless only if used with integers. To overcome this, the double floating point values are broken up into mantissa and exponent and delta-delta compressed as independent streams. As the exponent field rarely changes, it is elided if the delta-delta value is zero.

While a quantitative and comparative analysis of this compression algorithm is beyond the scope of this paper, its efficacy is shown in Section 6.

5.5 Generation linker

The generation linker receives a new tree overlay from the COW merge process, sorts the new tree nodes from deepest to shallowest and sends them to the block store individually, while resolving the temporary addresses to addresses native to the underlying storage provider. As nodes reference only nodes deeper than themselves, which have been written already, any temporary address encountered can be immediately resolved to a native address.

This stage is required because most efficient storage providers – such as append-only logs – can only write an object of arbitrary size to certain addresses. In the case of a simple file, arbitrarily sized objects can only be written to the tail, otherwise they overwrite existing data. Once the size of the object is known, such as after the linker sends the object to the block store and it is compressed, a new address can be derived from the previous one. The nature of the storage may limit how many addresses can be derived from a given initial address. For example, if the maximum file size is reached and a new file needs to be used.

For some storage providers, obtaining the first address is an expensive operation e.g. in a cluster operation, this could involve obtaining a distributed lock to ensure uniqueness of the generated addresses. For this reason the block store maintains a pool of pre-created initial addresses.

5.6 Root map

The *root map* is used before tree construction in both reads and writes. It resolves a UUID and a version to a storage "address." When the blocks for a new version have been acknowledged as durably persisted by the storage provider, a new mapping for the version is inserted into this root map. It is important that the map is fault tolerant as it represents a single point of failure. Without this mapping, no streams can be accessed. If the latest version entry for a stream is removed from the map, it is logically equivalent to rolling back the commit. Incidentally, as the storage costs of a small number of orphaned versions are low, this behaviour can be used deliberately to obtain cheap single-stream transaction semantics without requiring support code in the database.

The demands placed by these inserts / requests are much lower than those placed by the actual data, so many off-the-shelf solutions can provide this component of the design. We use MongoDB as it has easy-to-use replication.

One side effect of the choice of an external provider is that the latency in resolving this first lookup is present in all queries – even ones that hit the cache for the rest of the query. Due to the small size of this map, it would be reasonable to replicate the map on all BTrDB nodes and use a simpler storage solution to reduce this latency. All the records are the same size, and the version numbers increment sequentially, so a flat file indexed by offset would be acceptable.

5.7 Storage provider

The storage provider component wraps an underlying durable storage system and adds write batching, prefetching, and a pool of connection handles. In BTrDB, a tree commit can be done as a single write, as long as addresses can be generated for all the nodes in the commit without performing intermediate communication with the underlying storage. Throttling to the underlying storage is implemented here, for reasons described in Section 7.3.

As the performance of a storage system generally decreases with the richness of its features, BTrDB is designed to require only three very simple properties from the underlying storage:

1. It must be able to provide one or more free "addresses" that an arbitrarily large object can be written to later. Only a small finite number of these addresses need be outstanding at a time.

2. Clients must be able to derive another free "address" from the original address, and the size of the object that was written to it.

3. Clients must be able to read back data given just the "address" and a length.

Additional properties may be required based on the desired characteristics of the BTrDB deployment as a whole, for example distributed operation and durable writes. Sans these additional requirements, even a simple file is sufficient as a storage provider: (1) is the current size of the file, (2) is addition and (3) is a random read.

Note that as we are appending and reading with a file-like API, almost every distributed file system automatically qualifies as acceptable, such as HDFS, GlusterFS [25], CephFS [32], MapR-FS, etc.

Note also that if arbitrary but unique "addresses" are made up, then any database offering a key-value API would also work, e.g. , Cassandra, MongoDB, RADOS [33] (the object store under CephFS), HBase or BigTable. Most of these offer capabilities far beyond what is required by BTrDB, however, usually at a performance or space cost.

Although we support file-backed storage, we use Ceph RADOS in production. Initial addresses are read from a monotonically increasing integer stored in a RADOS object. Servers add a large increment to this integer while holding a distributed lock (provided by Ceph). The server then has a range of numbers it knows are unique. The high bits are used as a 16MB RADOS object identifier, while the low bits are used as an offset within that object. The address pool in the block store decouples the latency of this operation from write operations.

6 Quasi-production implementation

An implementation of BTrDB has been constructed using Go [11]. This language was chosen as it offers primitives that allow for rapid development of highly SMP-scalable programs in a SEDA [34] paradigm – namely channels and goroutines. As discussed above, one of the primary tenets of BTrDB is performance through simplicity: the entire implementation sans test code and auto-generated libraries is only 4709 lines.

Various versions of BTrDB have been used in a year-long deployment to capture data from roughly 35 microsynchrophasors deployed in the field, comprising 12 streams of 120 Hz data each. Data from these devices streams in over LTE and wired connections, as shown in Figure 2, leading to unpredictable delays, out-of-order chunk delivery and many duplicates (when the GPS-derived time synchronizes to different satellites). Many of the features present in BTrDB were developed to support the storage and analysis of this sensor data.

The hardware configuration for this deployment is shown in Figure 5. The compute server runs BTrDB (in a single node configuration). It also runs the DISTIL analytics framework [1] and a MongoDB replica set master. The MongoDB database is used for the root map along with sundry metadata, such as engineering units for the streams and configuration parameters for DISTIL algorithms. The storage server is a single-socket server containing 28 commodity 4TB 5900 RPM spinning-metal drives. The IO capacity of this server may seem abnormally low for a high performance database, but it is typical for data warehousing applications. BTrDB's IO pattern was chosen with this type of server in mind: 1MB reads and writes with excellent data locality for the primary analytics workload.

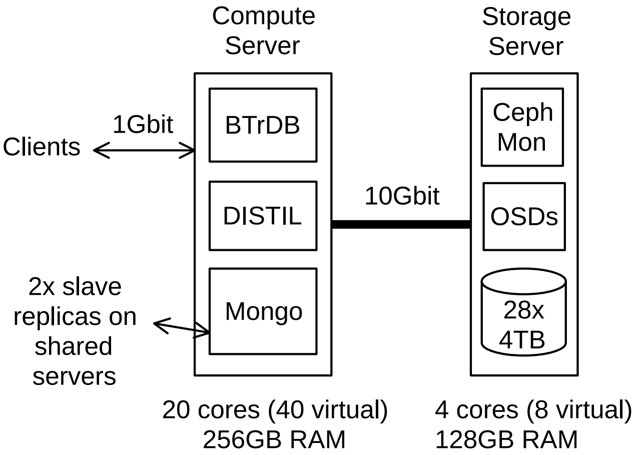

Compute Server
Storage Server
Clients
1Gbit
BTrDB
DISTIL
Mongo
10Gbit
Ceph Mon
OSDs
28x 4TB
2x slave replicas on shared servers
20 cores (40 virtual) 256GB RAM
4 cores (8 virtual) 128GB RAM

Figure 5: The architecture of our production system

6.1 Golang – the embodiment of SEDA

SEDA advocates constructing reliable, high-performance systems via decomposition into independent stages separated by queues with admission control. Although not explicitly referencing this paradigm, Go encourages the partitioning of complex systems into logical units of concurrency, connected by *channels*, a Go primitive roughly equal to a FIFO with atomic enqueue and dequeue operations. In addition, Goroutines – an extremely lightweight thread-like primitive with userland scheduling – allow for components of the system to be allocated pools of goroutines to handle events on the channels connecting a system in much the same way that SEDA advocates event dispatch. Unlike SEDA's Java implementation, however, Go is actively maintained, runs at near native speeds and can elegantly manipulate binary data.

6.2 Read throttling

As discussed below, carefully applied backpressure is necessary to obtain good write performance. In contrast, we have not yet found the need to explicitly throttle reads, despite having a higher read load than write load. The number of blocks that are kept in memory to satisfy a read is fewer than for a write. If the nodes are not already in the block cache (of which 95% are), they are needed only while their subtree is traversed, and can be freed afterwards. This differs from a write, where all the traversed blocks will be copied and must therefore be kept in memory until the linker has patched them and written them to the storage provider. In addition, Go channels are used to stream query data directly to the socket as it is read. If the socket is too slow, the channel applies back pressure to the tree traversal so that nodes are not fetched until the data has somewhere to go. For this reason, even large queries do not place heavy memory pressure on the system.

6.3 Real-data quantitative evaluation

Although the version of BTrDB running on production is lacking the performance optimizations implemented on the version evaluated in Section 7, it can provide insight into the behavior of the database with large, real data sets. At the time of writing, we have accumulated more than 2.1 trillion data points over 823 streams, of which 500 billion are spread over 506 streams feeding from instruments deployed in the field. The remaining 1.6 trillion points were produced by the DISTIL analysis framework. Of this analysis data, roughly 1.1 trillion points are in extents that were invalidated due to algorithm changes, manual flagging or replaced data in input streams. This massive dataset allows us to assess several aspects of the design.

Compression: A concern with using a copy-on-write tree data structure with "heavyweight" internal nodes is that the storage overheads may be unacceptable. With real data, the compression more than compensates for this overhead. The total size of the instrument data in the production Ceph pool (not including replication) is 2.757 TB. Dividing this by the number of raw data points equates to 5.514 bytes per reading *including all statistical and historical overheads*. As the raw tuples are 16 bytes, we have a compression ratio of 2.9x despite the costs of the time-partitioning tree. Compression is highly data dependent, but this ratio is better than the results of in-depth parametric studies of compression on similar synchrophasor telemetry [16][30].

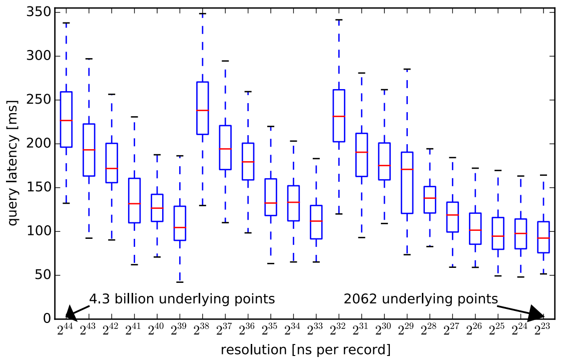


Figure 6: Query latencies for 2048 statistical records covering a varying time extent (1 year to 5 seconds), queried from a single node

Statistical queries: As these queries come into play with larger data sets, they are best evaluated on months of real data, rather than the controlled study in Section 7. These queries are typically used in event detectors to locate areas of interest – the raw data is too big to navigate with ease – and for visualization. To emulate this workload, we query a year's worth of voltage data – the same data illustrated in Figure 1a – to locate a voltage sag (the dashed box) and then issue progressively finer-grained queries until we are querying a 5 second window (Figure 1b). Automated event detectors typically skip several levels of resolution between queries, but this pattern is typical of data exploration where a user is zooming in to the event interactively. This process is repeated 300 times, with pauses between each sequence to obtain distributions on the query response times. The results can be found in Figure 6. Typically these distributions would be tighter, but the production server is under heavy load.

Each query is for the same number of statistical records (2048), but the number of data points that these records represent grows exponentially as the resolution becomes coarser (right to left in Figure 6). In a typical on-the-fly rollup database, the query time would grow exponentially as well, but with BTrDB it remains roughly constant within a factor of three. The implementation's choice of K ($64 = 2^6$) is very visible in the query response times. The query can be satisfied directly from the internal nodes with no on-the-fly computation every 6 levels of resolution. In between these levels, BTrDB must perform a degree of aggregation – visible in the query latency – to return the statistical summaries, with the most work occurring just before the next tier of the tree (2^{44}, 2^{38}, 2^{32}). Below 2^{27} the data density is low enough ($<$ 16 points per pixel column) that the query is being satisfied from the leaves.

Cache hit ratios: Although cache behavior is workload dependent, our mostly-automated-analysis is likely representative of most use-cases. Over 22 days, the block cache has exhibited a 95.93% hit rate, and the Ceph read-behind prefetch cache exhibited a 95.22% hit rate.

6.4 Analysis pipeline

The raw data acquired from sensors in the field is eventually used for decision support; grid state estimation; island detection and reverse power flow detection, to name a few examples. To obtain useful information the data must first go through a pipeline consisting of multiple transformation, fusion and synthesis stages, as shown in Figure 2.

All the stages of the pipeline are implemented with the same analysis framework and all consist of the same sequence of operations: find changes in the inputs, compute which ranges of the outputs need to be updated, fetch the data required for the computation, compute the outputs, and insert them. Finally, if this process completes successfully, the version numbers of the inputs that the distiller has now "caught up to" are written to durable storage (the same MongoDB replica set used for the root map). This architecture allows for fault tolerance without mechanisms, as each computation is idempotent: the output range corresponding to a given input range is deleted and replaced for each run of a DISTIL stage. If any error occurs, simply rerun the stage until it completes successfully, before updating the "input → last version" metadata records for the input streams.

This illustrates the power of the BTrDB *CalculateDiff()* primitive: an analysis stream can be "shelved," i.e., not kept up to date, and when it becomes necessary later it can be brought up-to date just-in-time with guaranteed consistency, even if the changes to the dependencies have occurred at random times throughout the stream. Furthermore the consumer obtains this with just 8 bytes of state per stream. The mechanism allows changes in a stream to propagate to all streams dependent on it, even if the process materializing the dependent stream is not online or known to the process making the change upstream. Achieving this level of consistency guarantee in existing systems typically requires a journal of outstanding operations that must be replayed on downstream consumers when they reappear.

7 Scalability Evaluation

To evaluate the design principles and implementation of BTrDB in a reproducible manner, we use a configuration of seven Amazon EC2 instances. There are four primary servers, one metadata server and two load generators. These machines are all c4.8xlarge instances. These were chosen as they are the only available instance type with both 10GbE network capabilities and EBS optimization. This combination allows the scalability of BTrDB to be established in the multiple-node configuration where network and disk bandwidth are the limiting factors.

Metric	Mean	Std. dev.
Write bandwidth [MB/s]	833	151
Write latency [ms]	34.3	40.0
Read bandwidth [MB/s]	1174	3.8
Read latency [ms]	22.0	18.7

Table 2: The underlying Ceph pool performance at max bandwidth

Ceph version 0.94.3 was used to provide the storage pool over 16 Object Store Daemons (OSDs). It was configured with a size (replication factor) of two. The bandwidth characteristics of the pool are shown in Table 2. It is important to note the latency of operations to Ceph, as this establishes a lower bound on cold query latencies, and interacts with the transaction coalescence backpressure mechanism. The disk bandwidth on a given BTrDB node to one of the OSD volumes measured using `dd` was approximately 175MB/s. This matched the performance of the OSD reported by `ceph tell osd.N bench`.

To keep these characteristics roughly constant, the number of Ceph nodes is kept at four, irrespective of how many of the servers are running BTrDB for a given experiment, although the bandwidth and latency of the pool does vary over time. As the Ceph CRUSH data placement rules are orthogonal to the BTrDB placement rules, the probability of a RADOS request hitting a local OSD is 0.25 for all experiments.

7.1 Throughput

The throughput of BTrDB in raw record tuples per second is measured for inserts, cold queries (after flushing the BTrDB cache) and warm queries (with a preheated BTrDB cache). Each tuple is an 8 byte time stamp and an 8 byte value. Warm and cold cache performance is characterized independently, because it allows an estimation of performance under different workloads after estimating the cache hit ratio.

#BTrDB	Streams	Total points	#Conn	Insert [mil/s]	Cold Query [mil/s]	Warm Query [mil/s]
1	50	500 mil	30	16.77	9.79	33.54
2	100	1000 mil	60	28.13	17.23	61.44
3	150	1500 mil	90	36.68	22.05	78.47
4	200	2000 mil	120	53.35	33.67	119.87

Table 1: Throughput evaluation as number of servers and size of load increases

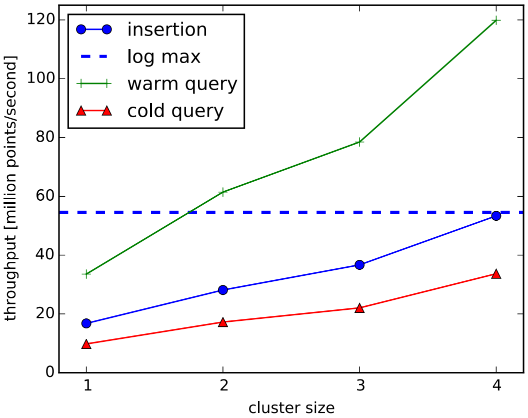


Figure 7: Throughput as the number of BTrDB nodes increases. The horizontal dashed line indicates the independently benchmarked write bandwidth of the underlying storage system.

	When insertion was	
Throughput [million pt/s] for	Chrono.	Random
Insert	28.12	27.73
Cold query in chrono. order	31.41	31.67
Cold query in same order	-	32.61
Cold query in random order	29.67	28.26
Warm query in chrono. order	114.1	116.2
Warm query in same order	-	119.0
Warm query in random order	113.7	117.2

Table 3: The influence of query/insert order on throughput

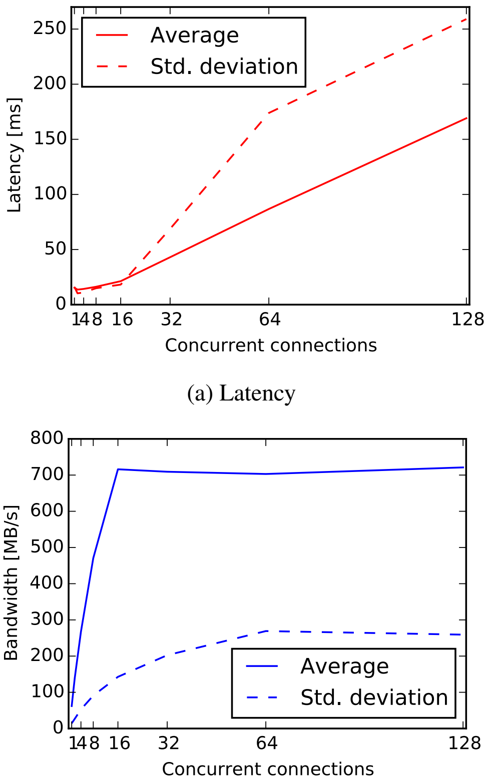
Inserts and queries are done in 10 kilorecord chunks, although there is no significant change in performance if this is decreased to 2 kilorecords.

Figure 7 shows that insert throughput scales linearly, to approximately 53 million records per second with four nodes. The horizontal dashed line is calculated as the maximum measured pool bandwidth (823MB/s) divided by the raw record size (16 bytes). This is the bandwidth that could be achieved by simply appending the records to a log in Ceph without any processing. This shows that despite the functionality that BTrDB offers, and the additional statistical values that must be stored, BTrDB performs on par with an ideal data logger.

The warm query throughput of 119 million readings per second is typical for tailing-analytics workloads where distillers process recently changed data. This throughput equates to roughly 1815 MB/s of network traffic, or 907MB/s per load generator.

7.2 Data and operation ordering

BTrDB allows data to be inserted in arbitrary order, and queried in arbitrary order. To characterize the effect of insertion and query order on throughput, measurements with randomized operations were performed. The workload consists of two hundred thousand inserts/queries of 10k points each (2 billion points in total). Two datasets were constructed, one where the data was inserted chronologically and one where the data was inserted randomly. After this, the performance of cold and warm queries in chronological order and random order were tested on both datasets. For the case of random insert, queries in the same (non-chronological) order as the insert were also tested. Note that operations were randomized at the granularity of the requests; within each request the 10k points were still in order. The results are presented in Table 3. The differences in throughput are well within experimental noise and are largely insignificant. This out-of-order performance is an important result for a database offering insertion speeds near that of an in-order append-only log.



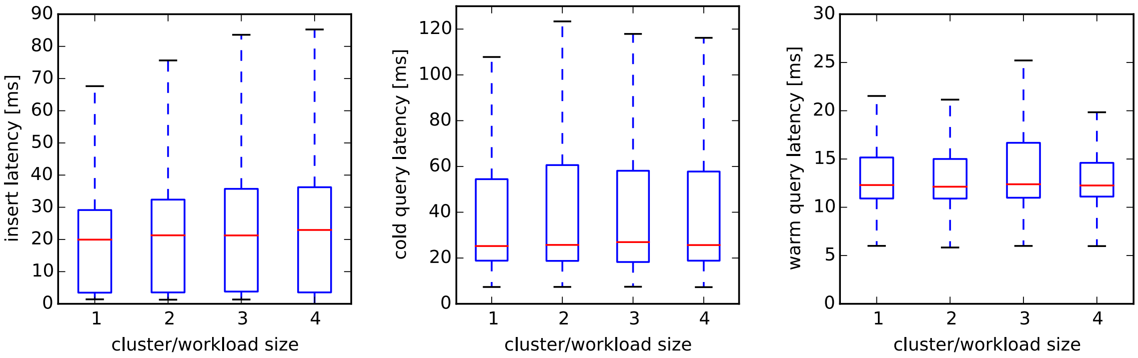
(a) Latency

(b) Aggregate bandwidth

Figure 9: Ceph pool performance characteristics as the number of concurrent connections increases

7.3 Latency

Although BTrDB is designed to trade a small increase in latency for a large increase in throughput, latency is still an important metric for evaluation of performance under load. The load generators record the time taken for each



(a) Insert latencies

(b) Cold query latencies

(c) Warm query latencies

Figure 8: Operation latencies as server count and workload is increased linearly

insert or query operation. Figure 8 gives an overview of the latency of operations as the workload and number of servers grows. Ideally all four points would be equal indicating perfect scaling. The range of latencies seen for insert operations increases as the cluster approaches the maximum bandwidth of Ceph. This is entirely Ceph's latency being presented to the client as backpressure. When a transaction coalescence buffer is full or being committed, no data destined for that stream is admitted to the database. Furthermore, a fixed number of tree merges are allowed at a time, so some buffers may remain full for some time. Although this appears counter-intuitive, in fact it increases system performance. Applying this backpressure early prevents Ceph from reaching pathological latencies. Consider Figure 9a where it is apparent that not only does the Ceph operation latency increase with the number of concurrent write operations, but it develops a long fat tail, with the standard deviation exceeding the mean. Furthermore, this latency buys nothing, as Figure 9b shows that the aggregate bandwidth plateaus after the number of concurrent operations reaches 16 – the number of OSDs.

With Ceph's latency characteristics in mind, BTrDB's write latency under maximum load is remarkable. A four node cluster inserting more than 53 million points per second exhibits a third quartile latency of 35ms: less than one standard deviation above the raw pool's latency.

7.4 Limitations and future work

The tests on EC2 show that the throughput and latency characteristics of the system are defined primarily by the underlying storage system. This is the ideal place to be, as it renders most further optimization in the timeseries database tier irrelevant.

The exception to this is optimizations that reduce the number of IO operations. We have already optimized the write path to the point of one write operation per commit. Nevertheless, there are significant performance gains to be had by optimizing the read path. One such avenue is to improve the block cache policy, reducing read ops. At present, the cache evicts the least recently used blocks. More complex policies could yield improved cache utilization: for example, if clients query only the most recent version of a stream, then all originals of blocks that were copied during a tree merge operation could be evicted from the cache. If most clients are executing statistical queries, then leaf nodes (which are 5x bigger than internal nodes) can be prioritized for eviction. Furthermore, as blocks are immutable, a distributed cache would not be difficult to implement as no coherency algorithm is required. Querying from memory on a peer BTrDB server would be faster than hitting disk via Ceph.

8 Conclusion

BTrDB provides a novel set of primitives, especially fast difference computation and rapid, low-overhead statistical queries that enable analysis algorithms to locate subsecond transient events in data comprising billions of datapoints spanning months – all in a fraction of a second. These primitives are efficiently provided by a time-partitioning version-annotated copy-on-write tree, which is shown to be easily implementable. A Go implementation is shown to outperform existing timeseries databases, operating at 53 million inserted values per second, and 119 million queried values per second with a four node cluster. The principles underlying this database are potentially applicable to a wide range of telemetry timeseries, and with slight modification, are applicable to all timeseries for which statistical aggregate functions exist and which are indexed by time.

Acknowledgments

The authors would like to thank Amazon and the UC Berkeley AMPLab for providing computing resources. In addition, this research is sponsored in part by the U.S. Department of Energy ARPA-E program (DE-AR0000340), National Science Foundation CPS-1239552, and Fulbright Scholarship program.

References

[1] ANDERSEN, M. P., KUMAR, S., BROOKS, C., VON MEIER, A., AND CULLER, D. E. DISTIL: Design and Implementation of a Scalable Synchrophasor Data Processing System. *Smart Grid, IEEE Transactions on* (2015).

[2] APACHE SOFTWARE FOUNDATION. Apache Cassandra home page. `http://cassandra.apache.org/`, 9 2015.

[3] APACHE SOFTWARE FOUNDATION. Apache HBase home page. `http://hbase.apache.org/`, 9 2015.

[4] BUEVICH, M., WRIGHT, A., SARGENT, R., AND ROWE, A. Respawn: A distributed multi-resolution time-series datastore. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th* (2013), IEEE, pp. 288–297.

[5] COUCHBASE. CouchBase home page. `http://www.couchbase.com/`, 9 2015.

[6] DATASTAX. DataStax home page. `http://www.datastax.com/`, 9 2015.

[7] DRUID. Druid.io. `http://druid.io/`, 9 2015.

[8] DUNNING, TED. MapR: High Performance Time Series Databases. `http://www.slideshare.net/NoSQLmatters/ted-dunning-very-high-bandwidth-time-series-database-implementation-nosql-matters-barcelona-2014`, 11 2014.

[9] ENDPOINT. Benchmarking Top NoSQL Databases. Tech. rep., Endpoint, apr 2015. `http://www.datastax.com/wp-content/themes/datastax-2014-08/files/NoSQL_Benchmarks_EndPoint.pdf`.

[10] GOLDSCHMIDT, T., JANSEN, A., KOZIOLEK, H., DOPPELHAMER, J., AND BREIVOLD, H. P. Scalability and robustness of time-series databases for cloud-native monitoring of industrial processes. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on* (2014), IEEE, pp. 602–609.

[11] GOOGLE. The Go Programming Language. `https://golang.org/`, 9 2015.

[12] INFLUXDB. InfluxDB home page. `https://influxdb.com/`, 9 2015.

[13] JOSH COALSON. Free Lossless Audio Codec. `https://xiph.org/flac/`, 9 2015.

[14] KAIROSDB. KairosDB import/export documentation. `https://kairosdb.github.io/kairosdocs/ImportExport.html`, 9 2014.

[15] KAIROSDB. KairosDB home page. `http://github.com/kairosdb/kairosdb`, 9 2015.

[16] KLUMP, R., AGARWAL, P., TATE, J. E., AND KHURANA, H. Lossless compression of synchronized phasor measurements. In *Power and Energy Society General Meeting, 2010 IEEE* (2010), IEEE, pp. 1–7.

[17] LAMBDA FOUNDRY. Pandas home page. `http://pandas.pydata.org/`, 9 2015.

[18] LELEWER, D. A., AND HIRSCHBERG, D. S. Data compression. *ACM Computing Surveys (CSUR) 19*, 3 (1987), 261–296.

[19] MONGODB INC. MongoDB home page. `https://www.mongodb.org/`, 9 2015.

[20] OPENTSDB. OpenTSDB home page. `http://opentsdb.net/`, 9 2015.

[21] ORACLE CORPORATION. MySQL home page. `https://www.mysql.com/`, 9 2015.

[22] PELKONEN, T., FRANKLIN, S., TELLER, J., CAVALLARO, P., HUANG, Q., MEZA, J., AND VEERARAGHAVAN, K. Gorilla: a fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1816–1827.

[23] PROJECT VOLDEMORT. Project Voldemort home page. `http://www.project-voldemort.com/`, 9 2015.

[24] RABL, T., GÓMEZ-VILLAMOR, S., SADOGHI, M., MUNTÉS-MULERO, V., JACOBSEN, H.-A., AND MANKOVSKII, S. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment 5*, 12 (2012), 1724–1735.

[25] REDHAT. Gluster home page. `http://www.gluster.org/`, 9 2015.

[26] REDIS LABS. Redis home page. `http://redis.io/`, 9 2015.

[27] SCOTT, JIM. MapR-DB OpenTSDB bulk inserter code. `https://github.com/mapr-demos/opentsdb/commit/c732f817498db317b8078fa5b53441a9ec0766ce`, 9 2014.

[28] SCOTT, JIM. MapR: Loading a time series database at 100 million points per second. `https://www.mapr.com/blog/loading-time-series-database-100-million-points-second`, 9 2014.

[29] STONEBRAKER, M., AND WEISBERG, A. The voltdb main memory dbms. *IEEE Data Eng. Bull. 36*, 2 (2013), 21–27.

[30] TOP, P., AND BRENEMAN, J. Compressing Phasor Measurement data. In *North American Power Symposium (NAPS), 2013* (Sept 2013), pp. 1–4.

[31] VOLTDB INC. VoltDB home page. `https://voltdb.com/`, 9 2015.

[32] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 307–320.

[33] WEIL, S. A., LEUNG, A. W., BRANDT, S. A., AND MALTZAHN, C. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07* (2007), ACM, pp. 35–44.

[34] WELSH, M., CULLER, D., AND BREWER, E. Seda: an architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review 35*, 5 (2001), 230–243.

Environmental Conditions and Disk Reliability in Free-Cooled Datacenters

Ioannis Manousakis,[†][*] *Sriram Sankar*,[‡] *Gregg McKnight*,[δ] *Thu D. Nguyen*,[†] *Ricardo Bianchini*[δ]

[†]*Rutgers University* [‡]*GoDaddy* [δ]*Microsoft*

Abstract

Free cooling lowers datacenter costs significantly, but may also expose servers to higher and more variable temperatures and relative humidities. It is currently unclear whether these environmental conditions have a significant impact on hardware component reliability. Thus, in this paper, we use data from nine hyperscale datacenters to study the impact of environmental conditions on the reliability of server hardware, with a particular focus on disk drives and free cooling. Based on this study, we derive and validate a new model of disk lifetime as a function of environmental conditions. Furthermore, we quantify the tradeoffs between energy consumption, environmental conditions, component reliability, and datacenter costs. Finally, based on our analyses and model, we derive server and datacenter design lessons.

We draw many interesting observations, including (1) relative humidity seems to have a dominant impact on component failures; (2) disk failures increase significantly when operating at high relative humidity, due to controller/adaptor malfunction; and (3) though higher relative humidity increases component failures, software availability techniques can mask them and enable free-cooled operation, resulting in significantly lower infrastructure and energy costs that far outweigh the cost of the extra component failures.

1 Introduction

Datacenters consume a massive amount of energy. A recent study [18] estimates that they consume roughly 2% and 1.5% of the electricity in the United States and world-wide, respectively. In fact, a single hyperscale datacenter may consume more than 30MW [7].

These staggering numbers have prompted many efforts to reduce datacenter energy consumption. Perhaps the most successful of these efforts have involved reducing the energy consumption of the datacenter cooling infrastructure. In particular, three important techniques have helped reduce the cooling energy: (1) increasing the hardware operating temperature to reduce the need for cool air inside the datacenter; (2) building datacenters where their cooling can directly leverage the outside air, reducing the need for energy-hungry (and expensive) water chillers; and (3) eliminating the hot air recirculation within the datacenter by isolating the cold air from the hot air. By using these and other techniques, large datacenter operators today can report yearly Power Usage Effectiveness (PUE) numbers in the 1.1 to 1.2 range, meaning that only 10% to 20% of the total energy goes into non-IT activities, including cooling. The low PUEs of these modern ("direct-evaporative-cooled" or simply "free-cooled"[1]) datacenters are substantially lower than those of older generation datacenters [14, 15].

Although lowering cooling costs and PUEs would seem like a clear win, increasing the operating temperature and bringing the outside air into the datacenter may have unwanted consequences. Most intriguingly, these techniques may decrease hardware component reliability, as they expose the components to aggressive environmental conditions (e.g., higher temperature and/or higher relative humidity). A significant decrease in hardware reliability could actually increase rather than decrease the total cost of ownership (TCO).

Researchers have not yet addressed the tradeoffs between cooling energy, datacenter environmental conditions, hardware component reliability, and overall costs in modern free-cooled datacenters. For example, the prior work on the impact of environmental conditions on hardware reliability [10, 25, 27] has focused on older (non-free-cooled) datacenters that maintain lower and more stable temperature and relative humidity at each spatial spot in the datacenter. Because of their focus on these datacenters, researchers have not addressed the reliability impact of relative humidity in energy-efficient, free-cooled datacenters at all.

[*] This work was done while Ioannis was at Microsoft.

[1] Throughout the paper, we refer to free cooling as the direct use of outside air to cool the servers. Some authors use a broader definition.

Understanding these tradeoffs and impacts is the topic of this paper. First, we use data collected from the operations of nine world-wide Microsoft datacenters for 1.5 years to 4 years to study the impact of environmental conditions (absolute temperature, temperature variation, relative humidity, and relative humidity variation) on the reliability of server hardware components. Based on this study and the dominance of disk failures, we then derive and validate a new model of disk lifetime as a function of both temperature and relative humidity. The model leverages data on the impact of relative humidity on corrosion rates. Next, we quantify the tradeoffs between energy consumption, environmental conditions, component reliability, and costs. Finally, based on our dataset and model, we derive server and datacenter design lessons.

We draw many observations from our dataset and analyses, including (1) disks account for the vast majority (89% on average) of the component failures regardless of the environmental conditions; (2) relative humidity seems to have a *much stronger* impact on disk failures than absolute temperature in current datacenter operating conditions, even when datacenters operate within ASHRAE's "allowable" conditions [4] (i.e., 10-35°C inlet air temperature and 20–80% relative humidity); (3) temperature variation and relative humidity variation are *negatively* correlated with disk failures, but this is a consequence of these variations tending to be strongest when relative humidity is low; (4) disk failure rates increase significantly during periods of high relative humidity, i.e. these periods exhibit temporal clustering of failures; (5) disk controller/connectivity failures increase significantly when operating at high relative humidity (the controller and the adaptor are the only parts that are exposed to the ambient conditions); (6) in high relative humidity datacenters, server designs that place disks in the back of enclosures can reduce the disk failure rate significantly; and (7) though higher relative humidity increases component failures, relying on software techniques to mask them and operate in this mode also significantly reduces infrastructure and energy costs, and *more than compensates* for the cost of the additional failures.

Note that, unlike disk vendors, we do not have access to a large isolated chamber where thousands of disks can be exposed to different environmental conditions in a controlled and repeatable manner.[2] Instead, we derive the above observations from multiple statistical analyses of large-scale commercial datacenters under their real operating conditions, as in prior works [10, 25, 27]. To increase confidence in our analyses and inferences, we personally inspected two free-cooled datacenters that experience humid environments and observed many samples of corroded parts.

In summary, we make the following contributions:

- We study the impact of relative humidity and relative humidity variation on hardware reliability (with a strong focus on disk drives) in datacenters.
- We study the tradeoffs between cooling energy, environmental conditions, hardware reliability, and cost in datacenters.
- Using data from nine datacenters, more than 1M disks, and 1.5–4 years, we draw many interesting observations. Our data suggests that the impact of temperature and temperature variation on disk reliability (the focus of the prior works) is *much less* significant than that of relative humidity in modern cooling setups.
- Using our disk data and a corrosion model, we also derive and validate a new model of disk lifetime as a function of environment conditions.
- From our observations and disk lifetime model, we draw a few server and datacenter design lessons.

2 Related Work

Environmentals and their impact on reliability. Several works have considered the impact of the cooling infrastructure on datacenter temperatures and humidities, e.g. [3, 22, 23]. However, they did not address the hardware reliability implications of these environmental conditions. The reason is that meaningful reliability studies require large server populations in datacenters that are monitored for multiple years. Our paper presents the largest study of these issues to date.

Other authors have had access to such large real datasets for long periods of time: [5, 10, 20, 25, 27, 28, 29, 35]. A few of these works [5, 28, 29, 35] considered the impact of age and other factors on hardware reliability, but did not address environmental conditions and their potential effects. The other prior works [10, 25, 27] have considered the impact of absolute temperature and temperature variation on the reliability of hardware components (with a significant emphasis on disk drives) in datacenters with fairly stable temperature and relative humidity at each spatial spot, i.e. non-air-cooled datacenters. Unfortunately, these prior works are inconclusive when it comes to the impact of absolute temperature and temperature variations on hardware reliability. Specifically, El-Sayed *et al.* [10] and Pinheiro *et al.* [25] found a smaller impact of absolute temperature on disk lifetime than previously expected, whereas Sankar *et al.* [27] found a significant impact. El-Sayed *et al.* also found temperature variations to have a more significant impact than absolute temperature on Latent Sector Errors

[2] From these experiments, vendors derive recommendations for the ideal operating conditions for their parts. Unfortunately, it is very difficult in practice to guarantee consistent operation within those conditions, as server layouts vary and the environment inside servers is difficult to control exactly, especially in free-cooled datacenters.

(LSEs), a common type of disk failure that renders sectors inaccessible. None of the prior studies considered relative humidity or relative humidity variations.

Our paper adds to the debate about the impact of absolute temperature and temperature variation. However, our paper suggests that this debate may actually be moot in modern (air-cooled) datacenters. In particular, our results show that relative humidity is a more significant factor than temperature. For this reason, we also extend an existing disk lifetime model with a relative humidity term, and validate it against our real disk reliability data.

Other tradeoffs. Prior works have also considered the impact of the cooling infrastructure and workload placement on cooling energy and costs, e.g. [1, 6, 8, 17, 19, 21, 23]. However, they did not address the impact of environmental conditions on hardware reliability (and the associated replacement costs). A more complete understanding of these tradeoffs requires a more comprehensive study, like the one we present in this paper. Specifically, we investigate a broader spectrum of tradeoffs, including cooling energy, environmental conditions, hardware reliability, and costs. Importantly, we show that the increased hardware replacement cost in free-cooled datacenters is *far outweighed* by their infrastructure and operating costs savings.

However, we do not address effects that our dataset does not capture. In particular, techniques devised to reduce cooling energy (increasing operating temperature and using outside air) may increase the energy consumption of the IT equipment, if server fans react by spinning faster. They may also reduce performance, if servers throttle their speed as a result of the higher operating temperature. Prior research [10, 32] has considered these effects, and found that the cooling energy benefits of these techniques outweigh the downsides.

3 Background

Datacenter cooling and environmentals. The cooling infrastructure of hyperscale datacenters has evolved over time. The first datacenters used water chillers with computer room air handlers (CRAHs). CRAHs do not feature the integrated compressors of traditional computer room air conditioners (CRACs). Rather, they circulate the air carrying heat from the servers to cooling coils carrying chilled water. The heat is then transferred via the water back to the chillers, which transfer the heat to another water loop directed to a cooling tower, before returning the chilled water back inside the datacenter. The cooling tower helps some of the water to evaporate (dissipating heat), before it loops back to the chillers. Chillers are expensive and consume a large amount of energy. However, the environmental conditions inside the datacenter can be precisely controlled (except for hot spots that may develop due to poor air flow design). Moreover, these datacenters do not mix outside and inside air. We refer to these datacenters as *chiller-based.*

Technology	Temp/RH Control	CAPEX	PUE
Chillers	Precise / Precise	\$2.5/W	1.7
Water-side	Precise / Precise	\$2.8/W	1.19
Free-cooled	Medium / Low	\$0.7/W	1.12

Table 1: Typical temperature and humidity control, CAPEX [11], and PUEs of the cooling types [13, 34].

An improvement over this setup allows the chillers to be bypassed (and turned off) when the cooling towers alone are sufficient to cool the water. Turning the chillers off significantly reduces energy consumption. When the cooling towers cannot lower the temperature enough, the chillers come back on. These datacenters tightly control the internal temperature and relative humidity, like their chiller-based counterparts. Likewise, there is still no mixing of outside and inside air. We refer to these datacenters as *water-side economized.*

A more recent advance has been to use large fans to blow cool outside air into the datacenter, while filtering out dust and other air pollutants. Again using fans, the warm return air is guided back out of the datacenter. When the outside temperature is high, these datacenters apply an evaporative cooling process that adds water vapor into the airstream to lower the temperature of the outside air, before letting it reach the servers. To increase temperature (during excessively cold periods) and/or reduce relative humidity, these datacenters intentionally recirculate some of the warm return air. This type of control is crucial because rapid reductions in temperature (more than $20°C$ per hour, according to ASHRAE [4]) may cause condensation inside the datacenter. This cooling setup enables the forgoing of chillers and cooling towers altogether, thus is the cheapest to build. However, these datacenters may also expose the servers to warmer and more variable temperatures, and higher and more variable relative humidities than other datacenter types. We refer to these datacenters as direct-evaporative-cooled or simply *free-cooled.*

A survey of the popularity of these cooling infrastructures can be found in [16]. Table 1 summarizes the main characteristics of the cooling infrastructures in terms of their ability to control temperature and relative humidity, and their estimated cooling infrastructure costs [11] and PUEs. For the PUE estimates, we assume Uptime Institute's surveyed average PUE of 1.7 [34] for chiller-based cooling. For the water-side economization PUE, we assume that the chiller only needs to be active 12.5% of the year, i.e. during the day time in the summer. The PUE of free-cooled datacenters depends on their locations, but we assume a single value (1.12) for simplicity. This value is in line with those reported by hyperscale datacenter operators. For example, Facebook's free-cooled datacenter in Prineville, Oregon reports an yearly average PUE of 1.08 with peaks around 1.14 [13]. All PUE estimates assume 4% overheads due to factors other than cooling.

Hardware lifetime models. Many prior reliability models associated component lifetime with temperature. For example, [30] considered several CPU failure modes that result from high temperatures. CPU manufacturers use high temperatures and voltages to accelerate the onset of early-life failures [30]. Disk and other electronics vendors do the same to estimate mean times to failure (mean lifetimes). The Arrhenius model is often used to calculate an *acceleration factor* (AF_T) for the lifetime [9].

$$AF_T = e^{\frac{E_a}{k}\cdot\left(\frac{1}{\overline{T_b}}-\frac{1}{\overline{T_e}}\right)} \tag{1}$$

where E_a is the activation energy (in eV) for the device, k is Boltzmann's constant ($8.62 \cdot 10^{-5}$ eV/K), $\overline{T_b}$ is the average baseline operating temperature (in K) of the device, and $\overline{T_e}$ is the average elevated temperature (in K).

The acceleration factor can be used to estimate how much higher the failure rate will be during a certain period. For example, if the failure rate is typically 2% over a year (i.e., 2% of the devices fail in a year) at a baseline temperature, and the acceleration factor is 2 at a higher temperature, the estimate for the accelerated rate will be 4% (2% × 2) for the year. In other words, $\overline{FR_T} = AF_T \times \overline{FR_{Tb}}$, where $\overline{FR_T}$ is the average failure rate due to elevated temperature, and $\overline{FR_{Tb}}$ is the average failure rate at the baseline temperature. Prior works [10, 27] have found the Arrhenius model to approximate disk failure rates accurately, though El-Sayed *et al.* [10] also found accurate linear fits to their failure data.

The Arrhenius model computes the acceleration factor assuming steady-state operation. To extrapolate the model to periods of changing temperature, existing models compute a weighted acceleration factor, where the weights are proportional to the length of the temperature excursions [31]. We take this approach when proposing our extension of the model to relative humidity and free-cooled datacenters. Our validation of the extended model (Section 6) shows very good accuracy for our dataset.

4 Methodology

In this section, we describe the main characteristics of our dataset and the analyses it enables. We purposely omit certain sensitive information about the datacenters, such as their locations, numbers of servers, and hardware vendors, due to commercial and contractual reasons. Nevertheless, the data we do present is plenty to make our points, as shall become clear in later sections.

DC Tag	**Cooling**	**Months**	**Refresh Cycles**	**Disk Popul.**
CD1	*Chiller*	48	2	117 K
CD2	*Water-Side*	48	2	146 K
CD3	*Free-Cooled*	27	1	24 K
HD1	*Chiller*	24	1	16 K
HD2	*Water-Side*	48	2	100 K
HH1	*Free-Cooled*	24	1	168 K
HH2	*Free-Cooled*	22	1	213 K
HH3	*Free-Cooled*	24	1	124 K
HH4	*Free-Cooled*	18	1	161 K
Total				**1.07 M**

Table 2: Main datacenter characteristics. The "C" and "D" tags mean cool and dry. An "H" as the first letter of the tag means hot, whereas an "H" as the second letter means humid.

Data sources. We collect data from nine hyperscale Microsoft datacenters spread around the world for periods from 1.5 to 4 years. The data includes component health and failure reports, traces of environmental conditions, traces of component utilizations, cooling energy data, and asset information.

The datacenters use a variety of cooling infrastructures, exhibiting different environmental conditions, hardware component reliabilities, energy efficiencies, and costs. The three first columns from the left of Table 2 show each datacenter's tag, its cooling technology, and the length of data we have for it. The tags correspond to the environmental conditions inside the datacenters (see caption for details), not their cooling technology or location. We classify a datacenter as "hot" ("H" as the first letter of its tag) if at least 10% of its internal temperatures over a year are above $24°C$, whereas we classify it as "humid" ("H" as the second letter of the tag) if at least 10% of its internal relative humidities over a year are above 60%. We classify a datacenter that is not "hot" as "cool", and one that is not "humid" as "dry". Although admittedly arbitrary, our naming convention and thresholds reflect the general environmental conditions in the datacenters accurately. For example, HD1 (hot and dry) is a state-of-the-art chiller-based datacenter that precisely controls temperature at a high setpoint. More interestingly, CD3 (cool and dry) is a free-cooled datacenter so its internal temperatures and relative humidities vary more than in chiller-based datacenters. However, because it is located in a cold region, the temperatures and relative humidities can be kept fairly low the vast majority of the time.

To study hardware component reliability, we gather failure data for CPUs, memory modules (DIMMs), power supply units (PSUs), and hard disk drives. The two rightmost columns of Table 2 list the number of disks we consider from each datacenter, and the number of "refresh" cycles (servers are replaced every 3 years) in each datacenter. We filter the failure data for entries with the following properties: (1) the entry was classified with a maintenance tag; (2) the component is in either *Failing* or *Dead* state according to the datacenter-wide health monitoring system; and (3) the entry's error message names the failing component. For example, a disk error will generate either a `SMART` (Self-Monitoring, Analysis, and Reporting Technology) report or a failure to detect the disk on its known `SATA` port. The nature of the error allows further classification of the underlying failure mode.

Defining exactly when a component has failed permanently is challenging in large datacenters [25, 28]. However, since many components (most importantly, hard disks) exhibit recurring errors before failing permanently and we do not want to double-count failures, we consider a component to have failed on the first failure reported to the datacenter-wide health monitoring system. This failure triggers manual intervention from a datacenter technician. After this first failure and manual repair, we count no other failure against the component. For example, we consider a disk to have failed on the first LSE that gets reported to the health system and requires manual intervention; this report occurs after the disk controller itself has already silently reallocated many sectors (e.g., 2000+ sectors for many disks in our dataset). Though this failure counting may seem aggressive at first blush (a component may survive a failure report and manual repair), note that others [20, 25] have shown high correlations of several types of `SMART` errors, like LSEs, with permanent failures. Moreover, the disk Annualized Failure Rates (AFRs) that we observe for chiller-based datacenters are in line with previous works [25, 28].

Detailed analyses. To correlate the disk failures with their environmental conditions, we use detailed data from one of the hot and humid datacenters (HH1). The data includes server inlet air temperature and relative humidity values, as well as outside air conditions with a granularity of 15 minutes. The dataset does not contain the temperature of all the individual components inside each server, or the relative humidity inside each box. However, we can accurately use the inlet values as the environmental conditions at the disks, because the disks are placed at the front of the servers (right at their air inlets) in HH1. For certain analyses, we use CD3 and HD1 as bases for comparison against HH1. Although we do not have information on the disks' manufacturing batch, our cross-datacenter comparisons focus on disks that differ mainly in their environmental conditions.

To investigate potential links between the components' utilizations and their failures, we collect historical average utilizations for the processors and disks in a granularity of 2 hours, and then aggregate them into lifetime average utilization for each component.

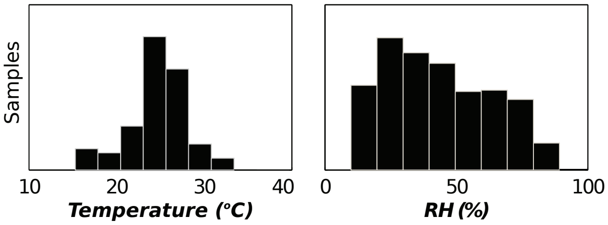


Figure 1: HH1 temperature and relative humidity distributions. Both of these environmentals vary widely.

We built a tool to process all these failure, environmental, and utilization data. After collecting, filtering, and deduplicating the data, the tool computes the AFRs and timelines for the component failures. With the timelines, it also computes daily and monthly failure rates. For disks, the tool also breaks the failure data across models and server configurations, and does disk error classification. Finally, the tool derives linear and exponential reliability models (via curve fitting) as a function of environmental conditions, and checks their accuracy versus the observed failure rates.

5 Results and Analyses

In this section, we first characterize the temperatures and relative humidities in a free-cooled datacenter, and the hardware component failures in the nine datacenters. We then perform a detailed study of the impact of environmental conditions on the reliability of disks. We close the section with an analysis of the hardware reliability, cooling energy, and cost tradeoffs.

5.1 Environmentals in free-cooled DCs

Chiller-based datacenters precisely control temperature and relative humidity, and keep them stable at each spatial spot in the datacenter. For example, HD1 exhibits a stable $27°C$ temperature and a 50% average relative humidity. In contrast, the temperature and relative humidity at each spatial spot in the datacenter vary under free cooling. For example, Figure 1 shows the temperature and relative humidity distributions measured at a spatial spot in HH1. The average temperature is $24.5°C$ (the standard deviation is $3.2°C$) and the average relative humidity is 43% (the standard deviation is 20.3%). Clearly, HH1 exhibits wide ranges, including a large fraction of high temperatures (greater than $24°C$) and a large fraction of high relative humidities (greater than 60%).

5.2 Hardware component failures

In light of the above differences in environmental conditions, an important question is whether hardware component failures are distributed differently in different types

DC Tag	Cooling	AFR	Increase wrt $AFR = 1.5\%$
CD1	*Chiller*	1.5%	0%
CD2	*Water-Side*	2.1%	40%
CD3	*Free-Cooled*	1.8%	20%
HD1	*Chiller*	2.0%	33%
HD2	*Water-Side*	2.3%	53%
HH1	*Free-Cooled*	3.1%	107%
HH2	*Free-Cooled*	5.1%	240%
HH3	*Free-Cooled*	5.1%	240%
HH4	*Free-Cooled*	5.4%	260%

Table 3: Disk AFRs. HH1-HH4 incur the highest rates.

of datacenters. We omit the full results due to space limitations, but highlight that disk drive failures dominate with 76%–95% (89% on average) of all hardware component failures, regardless of the component models, environmental conditions, and cooling technologies. As an example, disks, DIMMs, CPUs, and PSUs correspond to 83%, 10%, 5%, and 2% of the total failures, respectively, in HH1. The other datacenters exhibit a similar pattern. Disks also dominate in terms of failure rates, with AFRs ranging from 1.5% to 5.4% (average 3.16%) in our dataset. In comparison, the AFRs of DIMMs, CPUs, and PSUs were 0.17%, 0.23%, and 0.59%, respectively. Prior works had shown that disk failures are the most common for stable environmental conditions, e.g. [27]. Our data shows that they also dominate in modern, hotter and more humid datacenters. Interestingly, as we discuss in Section 7, the placement of the components (e.g., disks) inside each server affects their failure rates, since the temperature and relative humidity vary as air flows through the server.

Given the dominance of disk failures and rates, we focus on them in the remainder of the paper.

5.3 Impact of environmentals

Disk failure rates. Table 3 presents the disk AFRs for the datacenters we study, and how much they differ relative to the AFR of one of the datacenters with stable temperature and relative humidity at each spatial spot (CD1). We repeat the cooling technology information from Table 2 for clarity. The data includes a small number of disk models in each datacenter. For example, CD3 and HD1 have two disk models, whereas HH1 has two disk models that account for 95% of its disks. More importantly, the most popular model in CD3 (55% of the total) and HD1 (85%) are the same. The most popular model in HH1 (82%) is from the same disk manufacturer and series as the most popular model in CD3 and HD1, and has the same rotational speed, bus interface, and form factor; the only differences between the models are their storage and cache capacities. In more detailed studies below, we compare the impact of environmentals on these two models directly.

We make several observations from these data:

1. The datacenters with consistently or frequently dry internal environments exhibit the lowest AFRs, regardless of cooling technologies. For example, CD1 and HD1 keep relative humidities stable at 50%.
2. High internal relative humidity increases AFRs by 107% (HH1) to 260% (HH4), compared to CD1. Compared to HD1, the increases range from 55% to 170%. For example, HH1 exhibits a wide range of relative humidities, with a large percentage of them higher than 50% (Figure 1).
3. Free cooling does not necessarily lead to high AFR, as CD3 shows. Depending on the local climate (and with careful humidity control), free-cooled datacenters can have AFRs as low as those of chiller-based and water-side economized datacenters.
4. High internal temperature does not directly correlate to the high range of AFRs (greater than 3%), as suggested by datacenters HD1 and HD2.

The first two of these observations are indications that relative humidity may have a significant impact on disk failures. We cannot state a stronger result based solely on Table 3, because there are many differences between the datacenters, their servers and environmental conditions. Thus, in the next few pages, we provide more detailed evidence that consistently points in the same direction.

Causes of disk failures. The first question then becomes why would relative humidity affect disks if they are encapsulated in a sealed package? Classifying the disk failures in terms of their causes provides insights into this question. To perform the classification, we divide the failures into three categories [2]: (1) mechanical (prefail) issues; (2) age-related issues; and (3) controller and connectivity issues. In Table 4, we list the most common errors in our dataset. Pre-fail and old-age errors are reported by `SMART`. In contrast, `IOCTL ATA PASS THROUGH` (inability to issue an ioctl command to the controller) and `SMART RCV DRIVE DATA` (inability to read the `SMART` data from the controller) are generated in the event of an unresponsive disk controller.

In Figure 2, we present the failure breakdown for the popular disk model in HD1 (top), CD3 (middle), and HH1 (bottom). We observe that 67% of disk failures are associated with `SMART` errors in HD1. The vast majority (65%) of these errors are pre-fail, while just 2% are old-age errors. The remaining 33% correspond to controller and connectivity issues. CD3 also exhibits a substantially smaller percentage (42%) of controller and connectivity errors than `SMART` errors (58%). In contrast, HH1 experiences a much higher fraction of controller and connectivity errors (66%). Given that HH1 runs its servers cooler than HD1 most of the time, its two-fold increase in

Error Name	Type
IOCTL ATA PASS THROUGH	*Controller/Connectivity*
SMART RCV DRIVE DATA	*Controller/Connectivity*
Raw_Read_Error_Rate	*Pre-fail*
Spin_Up_Time	*Pre-fail*
Start_Stop_Count	*Old age*
Reallocated_Sectors_Count	*Pre-fail*
Seek_Error_Rate	*Pre-fail*
Power_On_Hours	*Old age*
Spin_Retry_Count	*Pre-fail*
Power_Cycle_Count	*Old age*
Runtime_Bad_Block	*Old age*
End-to-End_Error	*Old age*
Airflow_Temperature	*Old age*
G-Sense_Error_Rate	*Old age*

Table 4: Controller/connectivity and SMART errors [2].

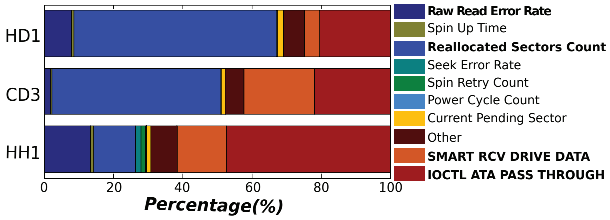


Figure 2: Failure classification in HD1 (top), CD3 (middle), and HH1 (bottom). Controller/connectivity failures in HH1 are double compared to HD1.

controller and connectivity errors (66% vs 33% in HD1) seems to result from its higher relative humidity. To understand the reason for this effect, consider the physical design of the disks. The mechanical parts are sealed, but the disk controller and the disk adaptor are directly exposed to the ambient, allowing possible condensation and corrosion agents to damage them.

The key observation here is:

5. High relative humidity seems to increase the incidence of disk controller and connectivity errors, as the controller board and the disk adaptor are exposed to condensation and corrosion effects.

Temporal disk failure clustering. Several of the above observations point to high relative humidity as an important contributor to disk failures. Next, we present even more striking evidence of this effect by considering the temporal clustering of failures of disks of the same characteristics and age.

Figure 3 presents the number of daily disk failures at HD1 (in red) and HH1 (in black) for the same two-year span. We normalize the failure numbers to the size of HD1's disk population to account for the large difference in population sizes. The green-to-red gradient band across the graph shows the temperature and humidity within HH1. The figure shows significant temporal clustering in the summer of 2013, when relative humidity at HH1 was frequently very high. Technicians found corrosion on the failed disks. The vast majority of the clustered HH1 failures were from disks of the same popular model; the disks had been installed within a few months of each other, 1 year earlier. Moreover, the figure shows increased numbers of daily failures in HH1 after the summer of 2013, compared to before it.

In more detail, we find that the HD1 failures were roughly evenly distributed with occasional spikes. The exact cause of the spikes is unclear. In contrast, HH1 shows a slightly lower failure rate in the first 12 months of its life, followed by a 3-month period of approximately 4x–5x higher daily failure rate. These months correspond to the summer, when the outside air temperature at HH1's location is often high. When the outside air is hot but dry, HH1 adds moisture to the incoming air to lower its temperature via evaporation. When the outside air is hot and humid, this humidity enters the datacenter directly, increasing relative humidity, as the datacenter's ability to recirculate heat is limited by the already high air temperature. As a result, relative humidity may be high frequently during hot periods.

We make three observations from these HH1 results:

6. High relative humidity may cause significant temporal clustering of disk failures, with potential consequences in terms of the needed frequency of manual maintenance/replacement and automatic data durability repairs during these periods.
7. The temporal clustering occurred in the *second* summer of the disks' lifetimes, suggesting that they did not fail simply because they were first exposed to high relative humidity. Rather, this temporal behavior suggests a disk lifetime model where high relative humidity excursions consume lifetime at a rate corresponding to their duration and magnitude. The amount of lifetime consumption during the summer of 2012 was not enough to cause an increased rate of failures; the additional consumption during the summer of 2013 was. This makes intuitive sense: it takes time (at a corrosion rate we model) for relative humidity (and temperature) to produce enough corrosion to cause hardware misbehavior. Our relative humidity modeling in Section 6 embodies the notion of lifetime consumption and matches our data well. A similar lifetime model has been proposed for high disk temperature [31].
8. The increased daily failures after the second summer provide extra evidence for the lifetime consumption model and the long-term effect of high relative humidity.

Correlating disk failures and environmentals. So far, our observations have listed several indications that relative humidity is a key disk reliability factor. However, one of our key goals is to determine the relative impact of

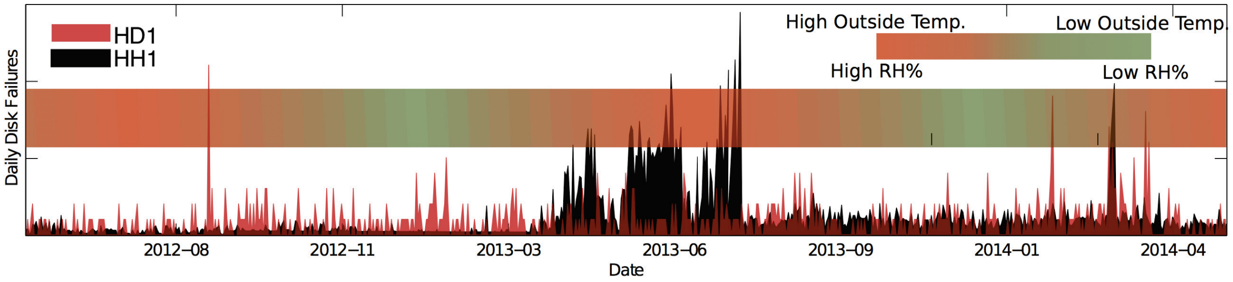


Figure 3: Two-year comparison between HD1 and HH1 daily failures. The horizontal band shows the outside temperature and relative humidity at HH1. HH1 experienced temporally clustered failures during summer'13, when the relative humidities were high. We normalized the data to the HD1 failures due to the different numbers of disks in these datacenters.

the four related environmental conditions: absolute temperature, relative humidity, temperature variation, and relative humidity variation.

In Table 5, we present both linear and exponential fits of the monthly failure rates for HH1, as a function of our four environmental condition metrics. For example, when seeking a fit for relative humidity, we use x = the average relative humidity experienced by the failed disks during a month, and y = the failure rate for that month. Besides the parameter fits, we also present R^2 as an estimate of the fit error. The exponential fit corresponds to a model similar to the Arrhenius equation (Equation 1). We use the disks' temperature and relative humidity Coefficient of Variation (CoV) to represent temporal temperature and relative humidity variations. (The advantage of the CoV over the variance or the standard deviation is that it is normalized by the average.) El-Sayed *et al.* [10] studied the same types of fits for their data, and also used CoVs to represent variations. Note that we split HH1's disk population into four groups (P1–P4) roughly corresponding to the spatial location of the disks in the datacenter; this accounts for potential environmental differences across cold aisles.

We can see that the relative humidity consistently exhibits positive correlations with failure rates, by considering the a parameter in the linear fit and the b parameter in the exponential fit. This positive correlation means that the failure rate increases when the relative humidity increases. In contrast, the other environmental conditions either show some or almost all negative parameters, suggesting much weaker correlations.

Interestingly, the temperature CoVs and the relative humidity CoVs suggest mostly *negative* correlations with the failure rate. This is antithetical to the physical stresses that these variations would be expected to produce. To explain this effect most clearly, we plot two correlation matrices in Figure 4 for the most popular disk model in HH1. The figure illustrates the correlations between all environmental condition metrics and the monthly failure rate. To derive these correlations, we use (1) the Pearson product-moment correlation coefficient, as a measure of linear dependence; and (2) Spearman's rank correlation coefficient, as a measure of nonlinear dependence. The color scheme indicates the correlation between each pair of metrics. The diagonal elements have a correlation equal to one as they represent dependence between the same metric. As before, a positive correlation indicates an analogous relationship, whereas a negative correlation indicates an inverse relationship. These matrices show that the CoVs are *strongly negatively* correlated (correlations close to -1) with average relative humidity. This implies that the periods with higher temperature and relative humidity variations are also the periods with low relative humidity, providing extended lifetime.

For completeness, we also considered the impact of average disk utilization on lifetime, but found no correlation. This is consistent with prior work, e.g. [25].

From the figures and table above, we observe that:

9. Average relative humidity seems to have the strongest impact on disk lifetime of all the environmental condition metrics we consider.
10. Average temperature seems to have a substantially lower (but still non-negligible) impact than average relative humidity on disk lifetime.
11. Our dataset includes no evidence that temperature variation or relative humidity variation has an effect on disk reliability in free-cooled datacenters.

5.4 Trading off reliability, energy, and cost

The previous section indicates that higher relative humidity appears to produce shorter lifetimes and, consequently, higher equipment costs (maintenance/repair costs tend to be small compared to the other TCO factors, so we do not consider them. However, the set of tradeoffs is broader. Higher humidity results from free cooling in certain geographies, but this type of cooling

Popul.	%	**Linear Fit** $a \cdot x + b$											
		Temperature			**RH**			**CoV - Temperature**			**CoV - RH**		
		a	b	R^2	a	b	R^2	a	b	R^2	a	b	R^2
P1	30.1	$5.17 \cdot 10^{-5}$	$-2.43 \cdot 10^{-3}$	0.81	$1.20 \cdot 10^{-4}$	$-4.88 \cdot 10^{-3}$	0.83	$-7.90 \cdot 10^{-3}$	$1.14 \cdot 10^{-3}$	0.83	$-6.56 \cdot 10^{-3}$	$3.29 \cdot 10^{-3}$	0.84
P2	25.6	$-1.91 \cdot 10^{-5}$	$2.46 \cdot 10^{-3}$	0.83	$1.03 \cdot 10^{-4}$	$-1.73 \cdot 10^{-3}$	0.84	$-9.06 \cdot 10^{-3}$	$1.28 \cdot 10^{-3}$	0.84	$-3.71 \cdot 10^{-3}$	$1.98 \cdot 10^{-3}$	0.83
P3	23.3	$1.41 \cdot 10^{-3}$	$-1.04 \cdot 10^{-1}$	0.75	$2.11 \cdot 10^{-4}$	$-5.59 \cdot 10^{-3}$	0.71	$-4.91 \cdot 10^{-2}$	$7.26 \cdot 10^{-2}$	0.77	$-4.46 \cdot 10^{-2}$	$2.42 \cdot 10^{-2}$	0.78
P4	19.6	$1.73 \cdot 10^{-3}$	$-1.07 \cdot 10^{-1}$	0.36	$4.45 \cdot 10^{-4}$	$-16.4 \cdot 10^{-3}$	0.44	$-1.36 \cdot 10^{-1}$	$1.33 \cdot 10^{-2}$	0.47	$-8.02 \cdot 10^{-2}$	$4.13 \cdot 10^{-2}$	0.55

Popul.	%	**Exponential Fit** $a \cdot e^{b \cdot x}$											
		Temperature			**RH**			**CoV - Temperature**			**CoV - RH**		
		a	b	R^2	a	b	R^2	a	b	R^2	a	b	R^2
P1	30.1	$2.38 \cdot 10^{-3}$	$-6.45 \cdot 10^{-3}$	0.84	$2.67 \cdot 10^{-4}$	$5.11 \cdot 10^{-2}$	0.89	$2.06 \cdot 10^{-3}$	$-1.74 \cdot 10^{0}$	0.72	$1.74 \cdot 10^{-2}$	$-9.01 \cdot 10^{0}$	0.81
P2	25.6	$3.08 \cdot 10^{-3}$	$-1.64 \cdot 10^{-2}$	0.85	$5.37 \cdot 10^{-4}$	$5.38 \cdot 10^{-2}$	0.88	$1.84 \cdot 10^{-3}$	$-1.09 \cdot 10^{1}$	0.79	$1.37 \cdot 10^{-2}$	$-1.63 \cdot 10^{1}$	0.81
P3	23.3	$2.57 \cdot 10^{-3}$	$6.35 \cdot 10^{-3}$	0.76	$5.31 \cdot 10^{-5}$	$9.93 \cdot 10^{-2}$	0.69	$5.13 \cdot 10^{-3}$	$-8.45 \cdot 10^{0}$	0.58	$1.66 \cdot 10^{-3}$	$3.57 \cdot 10^{0}$	0.57
P4	19.6	$3.62 \cdot 10^{-4}$	$3.36 \cdot 10^{-2}$	0.43	$1.31 \cdot 10^{-5}$	$11.54 \cdot 10^{-2}$	0.59	$6.10 \cdot 10^{-3}$	$-3.91 \cdot 10^{0}$	0.23	$7.17 \cdot 10^{-3}$	$-1.39 \cdot 10^{0}$	0.21

Table 5: Linear ($a \cdot x + b$) and nonlinear ($a \cdot e^{b \cdot x}$) fits for the monthly failure rates of four disk populations in HH1, as a function of the absolute temperature, relative humidity, temperature variation, and relative humidity variation.

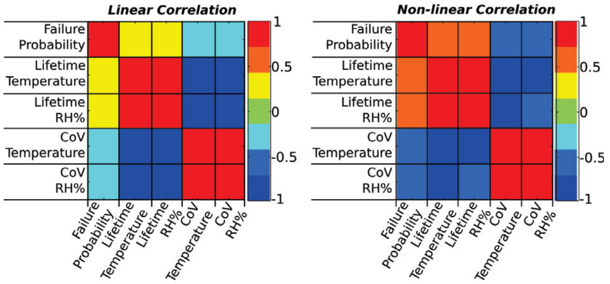


Figure 4: Linear and non-linear correlation between the monthly failure rate and the environmental conditions we consider, for the most popular disk model in HH1. red = high correlation, dark blue = low correlation.

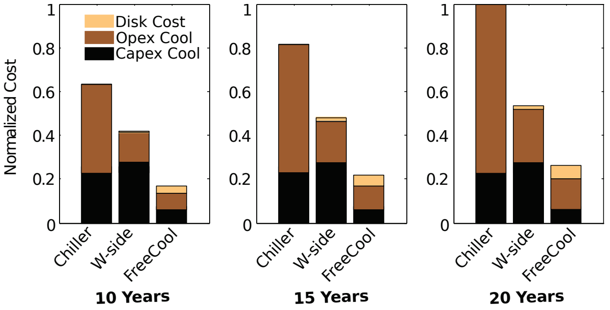


Figure 5: 10, 15, and 20-year cost comparison for chiller-based, water-side, and free-cooled datacenters including the cost replacing disks.

also improves energy efficiency and lowers both capital and operating costs.

Putting all these effects together, in Figure 5 we compare the cooling-related (capital + operational + disk replacement) costs for a chiller-based (CD1), a water-side economized (HD2), and a free-cooled datacenter (HH4). For the disk replacement costs, we consider only the cost above that of a 1.5% AFR (the AFR of CD1), so the chiller-based bars do not include these costs. We use HD2 and HH4 datacenters for this figure because they exhibit the highest disk AFRs in their categories; they represent the worst-case disk replacement costs for their respective datacenter classes. We use the PUEs and CAPEX estimates from Table 1, and estimate the cooling energy cost assuming $0.07/kWh for the electricity price (the average industrial electricity price in the United States). We also assume that the datacenter operator brunts the cost of replacing each failed disk, which we assume to be $100, by having to buy the extra disk (as opposed to simply paying a slightly more expensive warranty). We perform the cost comparison for 10, 15, and 20 years, as the datacenter lifetime.

The figure shows that, for a lifetime of 10 years, the cooling cost of the chiller-based datacenter is roughly balanced between capital and operating expenses. For a lifetime of 15 years, the operational cooling cost becomes the larger fraction, whereas for 20 years it becomes roughly 75% of the total cooling-related costs. In comparison, water-side economized datacenters have slightly higher capital cooling costs than chiller-based ones. However, their operational cooling costs are substantially lower, because of the periods when the chillers can be bypassed (and turned off). Though these datacenters may lead to slightly higher AFRs, the savings from lower operational cooling costs more than compensate for the slight disk replacement costs. In fact, the fully burdened cost of replacing a disk would have to be several fold higher than $100 (which is unlikely) for replacement costs to dominate. Finally, the free-cooled datacenter exhibits lower capital cooling costs (by 3.6x) and operational cooling costs (by 8.3x) than the chiller-based datacenter. Because of these datacenters' sometimes higher AFRs, their disk replacement costs may be non-trivial, but the overall cost tradeoff is still clearly in their favor. For 20 years, the free-cooled datacenter exhibits overall cooling-related costs that are roughly 74% and 45% lower than the chiller-based and water-side economized datacenters, respectively.

Based on this figure, we observe that:

12. Although operating at higher humidity may entail substantially higher component AFRs in certain geographies, the cost of this increase is small compared to the savings from reducing energy consumption and infrastructure costs via free cooling, especially for longer lifetimes.

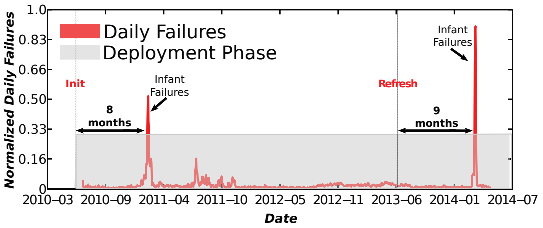


Figure 6: Normalized daily failures in CD1 over 4 years. Infant mortality seems clear.

6 Modeling Lifetime in Modern DCs

Models of hardware component lifetime have been used extensively in industry and academia to understand and predict failures, e.g. [9, 29, 30]. In the context of a datacenter, which hosts hundreds of thousands of components, modeling lifetimes is important to select the conditions under which datacenters should operate and prepare for the unavoidable failures. Because disk drives fail much more frequently than other components, they typically receive the most modeling attention. The two most commonly used disk lifetime models are the *bathtub model* and *the failure acceleration model based on the Arrhenius equation* (Equation 1). The latter acceleration model does not consider the impact of relative humidity, which has only become a significant factor after the advent of free cooling (Section 5.3).

Bathtub model. This model abstracts the lifetime of large component populations as three consecutive phases: (1) in the first phase, a significant number of components fail prematurely (*infant mortality*); (2) the second phase is a stable period of lower failure rate; and (3) in the third phase, component failure increase again due to wear-out effects. We clearly observe infant mortality in some datacenters but not others, depending on disk model/vendor. For example, Figure 3 does not show any obvious infant mortality in HH1. The same is the case of CD3. (Prior works have also observed the absence of infant mortality in certain cases, e.g. [28].) In contrast, CD1 uses disks from a different vendor and does show clear infant mortality. Figure 6 shows the normalized number of daily failures over a period of 4 years in CD1. This length of time implies 1 server refresh (which happened after 3 years of operation). The figure clearly shows 2 spikes of temporally clustered failures: one group 8 months after the first server deployment, and another 9 months after the first server refresh. A full deployment/refresh may take roughly 3 months, so the first group of disks may have failed 5-8 months after deployment, and the second 6-9 months after deployment.

A new disk lifetime model for free-cooled datacenters. Section 5.3 uses multiple sources of evidence to argue that relative humidity has a significant impact on hardware reliability in humid free-cooled datacenters. The section also classifies the disk failures into two main groups: SMART and controller/connectivity. Thus, we extend the acceleration model above to include relative humidity, and recognize that there are two main disk lifetime processes in free-cooled datacenters: (1) one that affects the disk mechanics and SMART-related errors; and (2) another that affects its controller/connector.

We model process #1 as the Arrhenius acceleration factor, i.e. $AF_1 = AF_T$ (we define AF_T in Equation 1), as has been done in the past [10, 27]. For process #2, we model the corrosion rate due to high relative humidity and temperature, as both of them are known to affect corrosion [33]. Prior works have devised models allowing for more than one accelerating variable [12]. A general such model extends the Arrhenius failure rate to account for relative humidity, and compute a corrosion rate CR:

$$CR(\overline{T},\overline{RH}) = const \cdot e^{(\frac{-E_a}{k\cdot\overline{T}})} \cdot e^{(b\cdot\overline{RH})+(\frac{c\cdot\overline{RH}}{k\cdot\overline{T}})} \qquad (2)$$

where $\overline{T}$ is the average temperature, $\overline{RH}$ is the average relative humidity, E_a is the temperature activation energy, k is Boltzmann's constant, and $const$, b, and c are other constants. Peck empirically found an accurate model that assumes $c = 0$ [24], and we make the same assumption. Intuitively, Equation 2 exponentially relates the corrosion rate with both temperature and relative humidity.

One can now compute the acceleration factor AF_2 by dividing the corrosion rate at the elevated temperature and relative humidity $CR(\overline{T_e},\overline{RH_e})$ by the same rate at the baseline temperature and relative humidity $CR(\overline{T_b},\overline{RH_b})$. Essentially, AF_2 calculates how much faster disks will fail due to the combined effects of these environmentals. This division produces:

$$AF_2 = AF_T \cdot AF_{RH} \qquad (3)$$

where $AF_{RH} = e^{b\cdot(\overline{RH_e}-\overline{RH_b})}$ and AF_T is from Equation 1.

Now, we can compute the compound average failure rate $\overline{FR}$ as $AF_1 \cdot \overline{FR_{1b}} + AF_2 \cdot \overline{FR_{2b}}$, where $\overline{FR_{1b}}$ is the average mechanical failure rate at the baseline temperature, and $\overline{FR_{2b}}$ is the average controller/connector failure rate at the baseline relative humidity and temperature. The rationale for this formulation is that the two failure processes proceed in parallel, and a disk's controller/connector would not fail at exactly the same time as its other components; AF_1 estimates the extra failures due to mechanical/SMART issues, and AF_2 estimates the extra failures due to controller/connectivity issues.

To account for varying temperature and relative humidity, we also weight the factors based on the duration of those temperatures and relative humidities. Other works have used weighted acceleration factors, e.g. [31]. For simplicity in the monitoring of these environmentals, we can use the average temperature and average relative

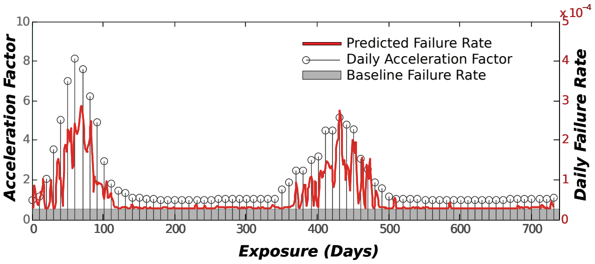


Figure 7: AFR prediction using our new disk lifetime model and the environmental conditions from HH1.

humidity per unit of time (e.g., hour, day) and compute the compound acceleration as a function of time t:

$$\overline{FR(t)} = AF_1(t) \cdot \overline{FR_{1b}(t)} + AF_2(t) \cdot \overline{FR_{2b}(t)} \qquad (4)$$

Model validation. We validate our disk lifetime model using real temperature, relative humidity, and daily failure rate data from HH1. (In the next section, we also validate it for CD3.) We use daily failure rates as they are finer grained than annualized failure rates and capture the underlying processes well.

Since ours is an acceleration model, we need to build it on a baseline failure rate. Unfortunately, our dataset does not include data that we could use as such a baseline (e.g., daily failure rates for a datacenter that keeps temperature and relative humidity consistently low *and* uses the same disk model as HH1). Instead of this data, we use daily failure rates derived from Equation 4 with temperature and relative humidity fixed at 20°C and 30%, respectively. For this baseline, we use E_a = 0.46 as computed by Sankar *et al.* [27], and b = 0.0455 as we derive from the copper corrosion rate measured by Rice *et al.* [26]. These values produce an AFR = 1.5%, and an average daily failure rate of 1.5%/365.

Using the actual temperatures and relative humidities of HH1, we apply our acceleration model to the baseline average daily failure rate. We train the model with data for populations P2 and P3 (for these data, b = 0.0652), and validate the results with data from P1 and P4. We do not include any infant mortality effects, because HH1 does not exhibit them. If desired, infant mortality can be modeled using a Weibull distribution, as other authors have done, e.g. [28].

Figure 7 shows the predicted acceleration factors, predicted daily failure rates, and the baseline failure rate over 2 years, starting in the beginning of the summer. Based on the daily baseline and acceleration factors, we compute the predicted daily failure rates and depict them with the red curve. These results show that our model is accurate: from the red curve, we compute the predicted disk AFR for P1 and P4 to be 3.04%, whereas the real AFR for these populations is 3.18%.

As one would expect from Section 5.3, the relative humidity contributes the most to the accuracy of the model. Removing the temperature components from the acceleration factors shows that it accounts for only 8.12% of the predicted AFR. Previous models (which are based solely on temperature) predict no increase in failure rates with respect to a baseline of 1.5%. Thus, models that do not account for relative humidity severely underestimate the corresponding failure rates.

7 Design Lessons

We next derive design lessons for servers and datacenters. We start by using our acceleration model to discuss different server layouts, and then discuss the implications of our findings to the placement, design, and management of datacenters.

Server design lessons. Our discussion of server designs relies on two main observations: (1) high relative humidity tends to produce more disk failures, as we have seen so far; and (2) higher temperature leads to lower relative humidity, given a constant amount of moisture in the air. These observations suggest that the layout of the disks within a server blade or enclosure may have a significant impact on its reliability in free-cooled datacenters.

To see this, consider Figure 8, where we present three possible layouts for a two-socket server blade. Our dataset has examples of all these layouts. In Figures 8(a) and (c), the disks are not exposed to the heat generated by the processors and memory DIMMs. This means that the relative humidity to which the disks will be exposed is roughly the same as that in the server's air inlet. In contrast, in Figure 8(b), the disks will be exposed to lower relative humidity, as they are placed downstream from the processors and DIMMs. The difference in relative humidity in this layout can be significant.

To demonstrate this difference, we consider the inlet air temperature and relative humidity data from CD3, in which the server blades have the layout of Figure 8(b). This datacenter exhibits average inlet air temperatures of 19.9°C, leading to a much higher air temperature of roughly 42°C at the disks. Given these temperatures and an average inlet relative humidity of 44.1%, psychometrics calculations show that the average relative humidity at the disks would be only 13%. This is one of the reasons that CD3 exhibits such a low AFR. In fact, given this adjusted temperature and relative humidity, our acceleration model produces an accurate prediction of the AFR: 1.75% versus the real AFR of 1.8%.

In contrast with CD3, HH1 uses server blades with the layout in Figure 8(a). As HH1 exhibits a wider range of temperatures and relative humidities than CD3, we compute what the maximum relative humidity at any disk across the entire range of air temperatures would be at the back to the servers. The maximum values would be

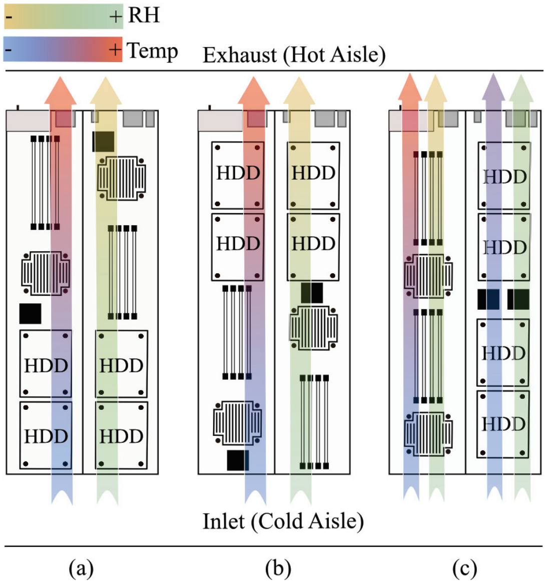


Figure 8: The server blade designs. Server designs with disks near the air exhaust lower their AFRs.

27% and 54.4°C at the disks. These values suggest that relative humidity should no longer have an impact on the AFR, while temperature should have a stronger impact. Overall, our model predicts AFR = 2.7%, instead of the actual AFR = 3.1% with disks in the front. Thus, changing the server layout in HH1 would decrease the importance of the relative humidity, increase the importance of temperature, but still produce a substantially lower AFR.

Obviously, placing disks at the back of servers could cause other components to experience higher relative humidity. Memory DIMMs would perhaps be the most relevant concern, as they also have exposed connectors. To prevent increases in AFR, DIMMs may be placed downstream from CPUs, which are the components most responsible for lowering relative humidity. Clearly, the full AFR and cost implications of the different layouts need to be understood, especially in free-cooled datacenters.

Datacenter lessons. As we argue in Section 5.4, free cooling reduces capital and operational cooling expenses significantly compared to other approaches. These savings may come at the cost of higher AFRs, depending on the outside environmental conditions at the datacenter's location. Fortunately, the server design lessons above suggest that organizations can lower the disk AFR of their hot and humid datacenters by simply using server designs in which the disks are placed in the back.

In fact, since high relative humidity seems much more harmful than high temperature, operators may consider running their datacenters somewhat hotter in the summer, instead of increasing the relative humidity to keep the temperature lower. Obviously, this has to be done carefully, since our observations and inferences apply only to the conditions that our datacenters have experienced.

Another important lesson involves data availability/reliability. Clearly, a higher disk AFR could require the datacenter software (i.e., its online services and/or management systems) to manage data redundancy more aggressively. Fortunately, this does not pose a serious problem for at least two reasons: (1) in large datacenters with hundreds of thousands of hardware components, failures are a common occurrence, so software is already capable of tolerating them via data redundancy within or across datacenters; (2) the increases in disk AFR that may result from using free cooling at challenging locations (roughly 3x in our dataset) are not large enough that they would threaten the reliability of data stored by these datacenters. In the worst case, the software would add slightly more redundancy to the data (disk space has negligible cost per bit, compared to other TCO factors). Nevertheless, if software cannot manage the impact of the higher AFRs, datacenter operators must tightly control the relative humidity and site their datacenters in locations where this is more easily accomplished.

8 Conclusions

In this paper, we studied the impact of environmental conditions on the reliability of disk drives from nine datacenters. We also explored the tradeoffs between environmental conditions, energy consumption, and datacenter costs. Based on these analyses, we proposed and validated a new disk lifetime model that is particularly useful for free-cooled datacenters. Using the model and our data, we derived server and datacenter design lessons.

Based on our experience and observations, we conclude that high relative humidity degrades reliability significantly, having a much more substantial impact than temperature or temperature variation. Thus, the design of free-cooled datacenters and their servers must consider the relative humidity to which components are exposed as a first-class issue. Organizations that operate datacenters with different cooling technologies in multiple geographies can select the conditions under which their services strike the right tradeoff between energy consumption, hardware reliability, cost, and quality of service.

Acknowledgements: We thank Darren Carroll, Eoin Doherty, Matthew Faist, Inigo Goiri, John Hardy, Xin Liu, Sunil Panghal, Ronan Murphy, Brandon Rubenstein, and Steve Solomon for providing some of the data we use in this paper. We are also indebted to Annemarie Callahan, Matthew Faist, Dave Gauthier, Dawn Larsen, Bikash Sharma, Mark Shaw, Steve Solomon, Kushagra Vaid, and Di Wang for comments that helped improve the paper significantly. Finally, we thank our shepherd, Daniel Peek, and the anonymous referees for their extensive comments and support.

References

[1] Ahmad, F., and Vijaykumar, T. Joint Optimization of Idle and Cooling Power in Data Centers while Maintaining Response Time. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (2010).

[2] Allen, B. Monitoring Hard Disks with SMART. *Linux Journal*, 117 (2004).

[3] Anubhav, K., and Yogendra, J. Use of Airside Economizer for Data Center Thermal Management. In *Proceedings of the 2nd International Conference on Thermal Issues in Emerging Technologies* (2008).

[4] ASHRAE Technical Committee TC 9.9. 2011 Thermal Guidelines for Data Processing Environments – Expanded Data Center Classes and Usage Guidance, 2011.

[5] Bairavasundaram, L. N., Goodson, G. R., Pasupathy, S., and Schindler, J. An Analysis of Latent Sector Errors in Disk Drives. *ACM SIGMETRICS Performance Evaluation Review 35*, 1 (2007).

[6] Banerjee, A., Mukherjee, T., Varsamopoulos, G., and Gupta, S. Cooling-Aware and Thermal-Aware Workload Placement for Green HPC Data Centers. In *Proceedings of the International Green Computing Conference* (2010).

[7] Barroso, L. A., Clidaras, J., and Holzle, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, second ed. 2013.

[8] Bash, C., Patel, C., and Sharma, R. Dynamic Thermal Management of Air Cooled Data Centers. In *Proceedings of the 10th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems* (2006).

[9] Cole, G. Estimating Drive Reliability in Desktop Computers and Consumer Electronics Systems. *Seagate Technology Paper*, TP-338.1 (2000).

[10] El-Sayed, N., Stefanovici, I., Amvrosiadis, G., Hwang, A., and Schroeder, B. Temperature Management in Data Centers: Why Some (Might) Like it Hot. In *Proceedings of the 12th International Conference on Measurement and Modeling of Computer Systems* (2012).

[11] Emerson Network Power. Energy Efficient Cooling Solutions for Data Centers, 2007.

[12] Escobar, L. A., and Meeker, W. Q. A Review of Accelerated Test Models. *Statistical Science 21*, 4 (2006).

[13] Facebook. Prineville Data Center – PUE/WUE, 2014.

[14] Greenberg, A., Hamilton, J., Maltz, D. A., and Patel, P. The Cost of a Cloud: Research Problems in Data Center Networks. *ACM SIGCOMM Computer Communication Review 39*, 1 (2008).

[15] Greenberg, S., Mills, E., Tschudi, B., Rumsey, P., and Myatt, B. Best Practices for Data centers: Lessons Learned from Benchmarking 22 Data Centers. In *ACEEE Summer Study on Energy Efficiency in Buildings* (2006).

[16] Kaiser, J., Bean, J., Harvey, T., Patterson, M., and Winiecki, J. Survey Results: Data Center Economizer Use. *White Paper* (2011).

[17] Kim, J., Ruggiero, M., and Atienza, D. Free Cooling-Aware Dynamic Power Management for Green Datacenters. In *Proceedings of the International Conference on High-Performance Computing and Simulation* (2012).

[18] Koomey, J. Growth in Data Center Electricity Use 2005 to 2010, 2011. Analytic Press.

[19] Liu, Z., Chen, Y., Bash, C., Wierman, A., Gmach, D., Wang, Z., Marwah, M., and Hyser, C. Renewable and Cooling Aware Workload Management for Sustainable Data Centers. In *Proceedings of the 12th International Conference on Measurement and Modeling of Computer Systems* (2012).

[20] Ma, A., Douglis, F., Lu, G., Sawyer, D., Chandra, S., and Hsu, W. RAIDShield: Characterizing, Monitoring, and Proactively Protecting Against Disk Failures. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (2015).

[21] Moore, J., Chase, J., Ranganathan, P., and Sharma, R. Making Scheduling "Cool": Temperature-Aware Workload Placement in Data Centers. In *Proceedings of the 2005 USENIX Annual Technical Conference* (2005).

[22] Patel, C. D., Sharma, R., Bash, C. E., and Beitelmal, A. Thermal Considerations in Cooling Large Scale High Compute Density Data Centers. In *Proceedings of the 8th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems* (2002).

[23] Patterson, M. K. The Effect of Data Center Temperature on Energy Efficiency. In *Proceedings of the 11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems* (2008).

[24] Peck, D. S. Comprehensive Model for Humidity Testing Correlation. In *Proceedings of the 24th Annual Reliability Physics Symposium* (1986).

[25] Pinheiro, E., Weber, W.-D., and Barroso, L. A. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (2007).

[26] Rice, D., Peterson, P., Rigby, E. B., Phipps, P., Cappell, R., and Tremoureux, R. Atmospheric Corrosion of Copper and Silver. *Journal of the Electrochemical Society 128*, 2 (1981).

[27] Sankar, S., Shaw, M., Vaid, K., and Gurumurthi, S. Datacenter Scale Evaluation of the Impact of Temperature on Hard Disk Drive Failures. *ACM Transactions on Storage 9*, 2 (2013).

[28] Schroeder, B., and Gibson, G. A. Understanding Disk Failure Rates: What Does an MTTF of 1,000,000 Hours Mean to You? *ACM Transactions on Storage 3*, 3 (2007).

[29] Shah, S., and Elerath, J. G. Reliability Analysis of Disk Drive Failure Mechanisms. In *Proceedings of the Annual Symposium on Reliability and Maintainability* (2005).

[30] Srinivasan, J., Adve, S. V., Bose, P., and Rivers, J. A. The Case for Lifetime Reliability-Aware Microprocessors. In *Proceedings of the 41st Annual International Symposium on Computer Architecture* (2004).

[31] The Green Grid. Data Center Efficiency and IT Equipment Reliability at Wider Operating Temperature and Humidity Ranges, 2012.

[32] Tolia, N., Wang, Z., Marwah, M., Bash, C., Ranganathan, P., and Zhu, X. Delivering Energy Proportionality with Non-Energy-Proportional Systems – Optimizing the Ensemble. In *Proceedings of the HotPower Workshop* (2008).

[33] Tullmin, M., and Roberge, P. R. Corrosion of Metallic Materials. *IEEE Transactions on Reliability 44*, 2 (1995).

[34] Uptime Institute. 2014 Data Center Industry Survey Results, 2014.

[35] Vishwanath, K. V., and Nagappan, N. Characterizing Cloud Computing Hardware Reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010).

Flash Reliability in Production: The Expected and the Unexpected

Bianca Schroeder
University of Toronto
Toronto, Canada

Raghav Lagisetty
Google Inc.
Mountain View, CA

Arif Merchant
Google Inc.
Mountain View, CA

Abstract

As solid state drives based on flash technology are becoming a staple for persistent data storage in data centers, it is important to understand their reliability characteristics. While there is a large body of work based on experiments with individual flash chips in a controlled lab environment under synthetic workloads, there is a dearth of information on their behavior in the field. This paper provides a large-scale field study covering many millions of drive days, ten different drive models, different flash technologies (MLC, eMLC, SLC) over 6 years of production use in Google's data centers. We study a wide range of reliability characteristics and come to a number of unexpected conclusions. For example, raw bit error rates (RBER) grow at a much slower rate with wear-out than the exponential rate commonly assumed and, more importantly, they are not predictive of uncorrectable errors or other error modes. The widely used metric UBER (uncorrectable bit error rate) is not a meaningful metric, since we see no correlation between the number of reads and the number of uncorrectable errors. We see no evidence that higher-end SLC drives are more reliable than MLC drives within typical drive lifetimes. Comparing with traditional hard disk drives, flash drives have a significantly lower replacement rate in the field, however, they have a higher rate of uncorrectable errors.

1 Introduction

The use of solid state drives based on NAND flash technology in data center servers is continuously growing. As more data lives on flash, data durability and availability critically depend on flash reliability. While it is widely understood that flash drives offer substantial performance improvements relative to hard disk drives, their failure characteristics are not well understood. The datasheets that manufacturers provide only contain vague guarantees, such as the number of times a flash chip can be erased before wearing out. Our current understanding is based on work that studies flash reliability in controlled lab experiments (such as accelerated life tests), using a small population of raw flash chips under synthetic workloads. There is a dearth of studies that report on the reliability of flash drives and their failure characteristics in large-scale production use in the field.

This paper provides a detailed field study of flash reliability based on data collected over 6 years of production use in Google's data centers. The data spans many millions of drive days [1], ten different drive models, different flash technologies (MLC, eMLC and SLC) and feature sizes (ranging from 24nm to 50nm). We use this data to provide a better understanding of flash reliability in production. In particular, our contributions include a detailed analysis of the following aspects of flash reliability in the field:

1. The different types of errors experienced by flash drives and their frequency in the field (Section 3).
2. Raw bit error rates (RBER), how they are affected by factors such as wear-out, age and workload, and their relationship with other types of errors (Section 4).
3. Uncorrectable errors, their frequency and how they are affected by various factors (Section 5).
4. The field characteristics of different types of hardware failure, including block failures, chip failures and the rates of repair and replacement of drives (Section 6).
5. A comparison of the reliability of different flash technologies (MLC, eMLC, SLC drives) in Sections 7, and between flash drives and hard disk drives in Section 8.

As we will see, our analysis uncovers a number of aspects of flash reliability in the field that are different from common assumptions and reports in prior work, and will hopefully motivate further work in this area.

[1] The size of their fleet and the number of devices in it is considered confidential at Google, so we can not provide precise numbers. We are making sure throughout this work that the reported numbers are statistically significant.

Model name	MLC-A	MLC-B	MLC-C	MLC-D	SLC-A	SLC-B	SLC-C	SLC-D	eMLC-A	eMLC-B
Generation	1	1	1	1	1	1	1	1	2	2
Vendor	I	II	I	I	I	I	III	I	I	IV
Flash type	MLC	MLC	MLC	MLC	SLC	SLC	SLC	SLC	eMLC	eMLC
Lithography (nm)	50	43	50	50	34	50	50	34	25	32
Capacity	480GB	480GB	480GB	480GB	480GB	480GB	480GB	960GB	2TB	2TB
PE cycle limit	3,000	3,000	3,000	3,000	100,000	100,000	100,000	100,000	10,000	10,000
Avg. PE cycles	730	949	529	544	860	504	457	185	607	377

Table 1: *Overview of drive models*

2 Background on data and systems

2.1 The flash drives

The drives in our study are custom designed high performance solid state drives, which are based on commodity flash chips, but use a custom PCIe interface, firmware and driver. We focus on two generations of drives, where all drives of the same generation use the same device driver and firmware. That means that they also use the same error correcting codes (ECC) to detect and correct corrupted bits and the same algorithms for wear-levelling. The main difference between different drive models of the same generation is the type of flash chips they comprise.

Our study focuses on the 10 drive models, whose key features are summarized in Table 1. Those models were chosen as they each span millions of drive days, comprise chips from four different flash vendors, and cover the three most common types of flash (MLC, SLC, eMLC).

2.2 The data

The data was collected over a 6-year period and contains for each drive aggregated monitoring data for each day the drive was in the field. Besides daily counts for a variety of different types of errors, the data also includes daily workload statistics, including the number of read, write, and erase operations, and the number of bad blocks developed during that day. The number of read, write, and erase operations includes user-issued operations, as well as internal operations due to garbage collection. Another log records when a chip was declared failed and when a drive was being swapped to be repaired.

3 Prevalence of different error types

We begin with some baseline statistics on the frequency of different types of errors in the field. We distinguish transparent errors, which the drive can mask from the user, and non-transparent errors, which will lead to a failed user operation. The device driver of the flash drives reports the following transparent types of errors:

Correctable error: During a read operation an error is detected and corrected by the drive internal ECC (error correcting code).
Read error: A read operation experiences a (non-ECC) error, but after retrying it succeeds.
Write error: A write operation experiences an error, but after retrying the operation succeeds.
Erase error: An erase operation on a block fails.

The devices report the following types of non-transparent errors:
Uncorrectable error: A read operation encounters more corrupted bits than the ECC can correct.
Final read error: A read operation experiences an error, and even after retries the error persists.
Final write error: A write operation experiences an error that persists even after retries.
Meta error: An error accessing drive-internal metadata.
Timeout error: An operation timed out after 3 seconds.

Uncorrectable errors include errors that were detected either during user-initiated operations or internal operations due to garbage collection, while final read errors include only errors encountered during user operations.

Note that errors vary in the severity of their possible impact. Besides the distinction between transparent and non-transparent errors, the severity of non-transparent errors varies. In particular, some of these errors (final read error, uncorrectable error, meta error) lead to data loss, unless there is redundancy at higher levels in the system, as the drive is not able to deliver data that it had previously stored.

We consider only drives that were put into production at least 4 years ago (for eMLC drives 3 years ago, as they are more recent drives), and include any errors that they experienced during their first 4 years in the field. Table 2 reports for each error type the fraction of drives for each model that experienced at least one error of that type (top half of table) and the fraction of drives days that had an error of that type (bottom half of table).

3.1 Non-transparent errors

We find that the most common non-transparent errors are final read errors, i.e. read errors that cannot be resolved even after retrying the operation. Depending on the model, between 20-63% of drives experience at least one such error and between 2-6 out of 1,000 drive days are affected. We find that the count of final read errors

Model name	MLC-A	MLC-B	MLC-C	MLC-D	SLC-A	SLC-B	SLC-C	SLC-D	eMLC-A	eMLC-B
Fraction of drives affected by different types of errors										
final read error	2.63e-01	5.64e-01	3.25e-01	3.17e-01	5.08e-01	2.66e-01	1.91e-01	6.27e-01	1.09e-01	1.27e-01
uncorrectable error	2.66e-01	5.75e-01	3.24e-01	3.24e-01	5.03e-01	2.84e-01	2.03e-01	6.34e-01	8.63e-01	9.05e-01
final write error	1.73e-02	2.11e-02	1.28e-02	1.85e-02	2.39e-02	2.33e-02	9.69e-03	5.67e-03	5.20e-02	3.16e-02
meta error	9.83e-03	7.97e-03	9.89e-03	1.93e-02	1.33e-02	3.68e-02	2.06e-02	7.04e-03	0.00e+00	0.00e+00
timeout error	5.68e-03	9.17e-03	5.70e-03	8.21e-03	1.64e-02	1.15e-02	8.47e-03	5.08e-03	0.00e+00	0.00e+00
response error	7.95e-04	3.90e-03	1.29e-03	1.88e-03	4.97e-03	2.08e-03	0.00e+00	9.78e-04	1.97e-03	8.76e-04
correctable error	9.89e-01	9.98e-01	9.96e-01	9.91e-01	9.99e-01	9.61e-01	9.72e-01	9.97e-01	9.97e-01	9.94e-01
read error	8.64e-03	1.46e-02	9.67e-03	1.12e-02	1.29e-02	1.77e-02	6.05e-03	1.02e-02	2.61e-01	2.23e-01
write error	6.37e-02	5.61e-01	6.11e-02	6.40e-02	1.30e-01	1.11e-01	4.21e-01	9.83e-02	5.46e-02	2.65e-01
erase error	1.30e-01	3.91e-01	9.70e-02	1.26e-01	6.27e-02	3.91e-01	6.84e-01	4.81e-02	1.41e-01	9.38e-02
Fraction of drive days affected by different types of errors										
final read error	1.02e-03	1.54e-03	1.78e-03	1.39e-03	1.06e-03	9.90e-04	7.99e-04	4.44e-03	1.67e-04	2.93e-04
uncorrectable error	2.14e-03	1.99e-03	2.51e-03	2.28e-03	1.35e-03	2.06e-03	2.96e-03	6.07e-03	8.35e-03	7.82e-03
final write error	2.67e-05	2.13e-05	1.70e-05	3.23e-05	2.63e-05	4.21e-05	1.21e-05	9.42e-06	1.06e-04	6.40e-05
meta error	1.32e-05	1.18e-05	1.16e-05	3.44e-05	1.28e-05	5.05e-05	3.62e-05	1.02e-05	0.00e+00	0.00e+00
timeout error	7.52e-06	9.45e-06	7.38e-06	1.31e-05	1.73e-05	1.56e-05	1.06e-05	8.88e-06	0.00e+00	0.00e+00
response error	7.43e-07	3.45e-06	2.77e-06	2.08e-06	4.45e-06	3.61e-06	0.00e+00	2.69e-06	2.05e-06	1.11e-06
correctable error	8.27e-01	7.53e-01	8.49e-01	7.33e-01	7.75e-01	6.13e-01	6.48e-01	9.00e-01	9.38e-01	9.24e-01
read error	7.94e-05	2.75e-05	3.83e-05	7.19e-05	3.07e-05	5.85e-05	1.36e-05	2.91e-05	2.81e-03	5.10e-03
write error	1.12e-04	1.40e-03	1.28e-04	1.52e-04	2.40e-04	2.93e-04	1.21e-03	4.80e-04	2.07e-04	4.78e-04
erase error	2.63e-04	5.34e-04	1.67e-04	3.79e-04	1.12e-04	1.30e-03	4.16e-03	1.88e-04	3.53e-04	4.36e-04

Table 2: *The prevalence of different types of errors. The top half of the table shows the fraction of drives affected by each type of error, and the bottom half the fraction of drive* days *affected.*

and that of uncorrectable errors is strongly correlated and conclude that these final read errors are almost exclusively due to bit corruptions beyond what the ECC can correct. For all drive models, final read errors are around two orders of magnitude more frequent (in terms of the number of drive days they affect) than any of the other non-transparent types of errors.

In contrast to read errors, write errors rarely turn into non-transparent errors. Depending on the model, 1.5-2.5% of drives and 1-4 out of 10,000 drive days experience a final write error, i.e. a failed write operation that did not succeed even after retries. The difference in the frequency of final read and final write errors is likely due to the fact that a failed write will be retried at other drive locations. So while a failed read might be caused by only a few unreliable cells on the page to be read, a final write error indicates a larger scale hardware problem.

Meta errors happen at a frequency comparable to write errors, but again at a much lower frequency than final read errors. This might not be surprising given that a drive contains much less meta-data than real data, which lowers the chance of encountering an error accessing meta data. Other non-transparent errors (timeout and response errors) are rare, typically affecting less than 1% of drives and less than 1 in 100,000 drive days.

3.2 Transparent errors

Maybe not surprisingly, we find that correctable errors are the most common type of transparent error. Virtually all drives have at least some correctable errors, and the majority of drive days (61-90%) experience correctable errors. We discuss correctable errors, including a study of raw bit error rates (RBER), in more detail in Section 4.

The next most common transparent types of error are write errors and erase errors. They typically affect 6-10% of drives, but for some models as many as 40-68% of drives. Generally less than 5 in 10,000 days experience those errors. The drives in our study view write and erase errors as an indication of a block failure, a failure type that we will study more closely in Section 6.

Errors encountered during a read operations are rarely transparent, likely because they are due to bit corruption beyond what ECC can correct, a problem that is not fixable through retries. Non-final read errors, i.e. read errors that can be recovered by retries, affect less than 2% of drives and less than 2-8 in 100,000 drive days.

In summary, besides correctable errors, which affect the majority of drive days, transparent errors are rare in comparison to all types of non-transparent errors. The most common type of non-transparent errors are uncorrectable errors, which affect 2–6 out of 1,000 drive days.

4 Raw bit error rates (RBER)

The standard metric to evaluate flash reliability is the raw bit error rate (RBER) of a drive, defined as the number of corrupted bits per number of total bits read (including correctable as well as uncorrectable corruption events). The second generation of drives (i.e. models eMLC-A and eMLC-B) produce precise counts of the number of corrupted bits and the number of bits read, allowing us to accurately determine RBER. The first generation of

Model name	MLC-A	MLC-B	MLC-C	MLC-D	SLC-A	SLC-B	SLC-C	SLC-D	eMLC-A	eMLC-B
Median RBER	2.1e-08	3.2e-08	2.2e-08	2.4e-08	5.4e-09	6.0 e-10	5.8 e-10	8.5 -09	1.0 e-05	2.9 e-06
95%ile RBER	2.2e-06	4.6e-07	1.1e-07	1.9e-06	2.8e-07	1.3e-08	3.4e-08	3.3e-08	5.1e-05	2.6e-05
99%ile RBER	5.8e-06	9.1e-07	2.3e-07	2.7e-05	6.2e-06	2.2e-08	3.5e-08	5.3e-08	1.2e-04	4.1e-05

Table 3: *Summary of raw bit error rates (RBER) for different models*

drives report accurate counts for the number of bits read, but for each page, consisting of 16 data chunks, only report the number of corrupted bits in the data chunk that had the largest number of corrupted bits. As a result, in the (unlikely) absolute worst case, where all chunks have errors and they all have the same number of errors as the worst chunk, the RBER rates could be 16X higher than the drives record. While irrelevant when comparing drives within the same generation, this subtlety must be kept in mind when comparing across generations.

4.1 A high-level view of RBER

Table 3 shows for each drive model the median RBER across all drives for that model, as well as the 95th and 99th percentile. We decided to work with medians and percentiles since we find averages to be heavily biased by a few outliers, making it hard to identify any trends.

We observe large differences in the RBER across different drive models, ranging from as little as 5.8e-10 to more than 3e-08 for drives of the first generation. The differences are even larger when considering the 95th or 99th percentile RBER, rather than the median. For example, the 99th percentiles of RBER ranges from 2.2e-08 for model SLC-B to 2.7e-05 for MLC-D. Even within drives of the same model, there are large differences: the RBER of a drive in the 99th percentile tends to be at least an order of magnitude higher than the RBER of the median drive of the same model.

The difference in RBER between models can be partially explained by differences in the underlying flash technology. RBER rates for the MLC models are orders of magnitudes higher than for the SLC models, so the higher price point for the SLC models pays off with respect to RBER. We will see in Section 5 whether these differences will translate to differences in user-visible, non-transparent errors.

The eMLC models report RBER that are several orders of magnitude larger than for the other drives. Even taking into account that the RBER for the first generation drives are a lower bound and might in the worst case be 16X higher, there is still more than an order of magnitude difference. We speculate that feature size might be a factor, as the two eMLC models have the chips with the smallest lithography of all models.

Finally, there is not one vendor that consistently outperforms the others. Within the group of SLC and eMLC drives, respectively, the same vendor is responsible for one of the worst and the best models in the group.

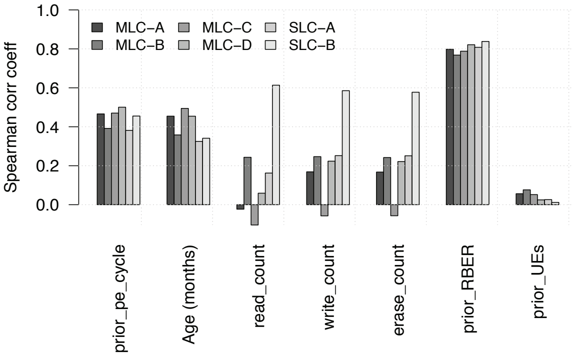


Figure 1: *The Spearman rank correlation coefficient between the RBER observed in a drive month and other factors.*

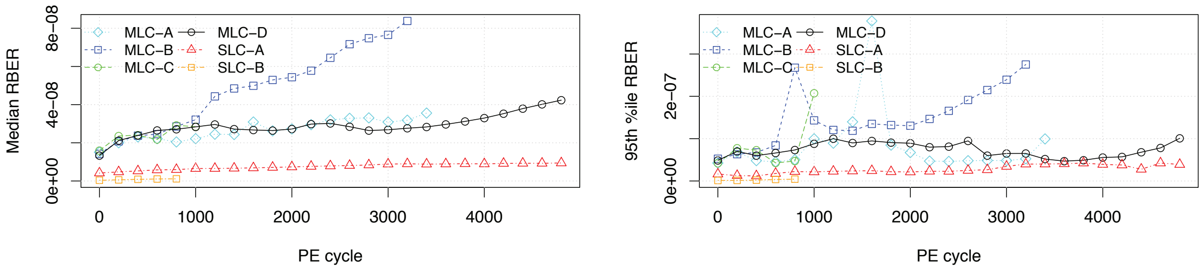
In summary, RBER varies greatly across drive models and also across drives within the same model. This motivates us to further study what factors affect RBER.

4.2 What factors impact RBER

In this section, we consider the effect of a number of factors on RBER: wear-out from program erase (PE) cycles; physical age, i.e. the number of months a device has been in the field, independently of PE cycles; workload, measured by the number of read, write, and erase operations, as an operation to a page can potentially disturb surrounding cells; and the presence of other errors.

We study the effect of each factor on RBER in two different ways. We use visual inspection by plotting the factor against RBER and we quantify the relationship using correlation coefficients. We use the Spearman rank correlation coefficient as it can also capture non-linear relationships, as long as they are monotonic (in contrast, for example, to the Pearson correlation coefficient).

Before analyzing individual factors in detail, we present a summary plot in Figure 1. The plot shows the Spearman rank correlation coefficient between the RBER observed in a given drive month, and other factors that were present, including the device age in months, the number of previous PE cycles, the number of read, write or erase operations in that month, the RBER observed in the previous month and the number of uncorrectable errors (UEs) in the previous month. Values for the Spearman correlation coefficient can range from -1



(a)

(b)

Figure 2: *The figures show the median and the 95th percentile RBER as a function of the program erase (PE) cycles.*

(strong negative correlation) to +1 (strong positive correlation). Each group of bars shows the correlation coefficients between RBER and one particular factor (see label on X-axis) and the different bars in each group correspond to the different drive models. All correlation coefficients are significant at more than 95% confidence.

We observe that all of the factors, except the prior occurrence of uncorrectable errors, show a clear correlation with RBER for at least some of the models. We also note that some of these correlations might be spurious, as some factors might be correlated with each other. We will therefore investigate each factor in more detail in the following subsections.

4.2.1 RBER and wear-out

As the endurance of flash cells is limited, RBER rates are expected to grow with the number of program erase (PE) cycles, with rates that have previously been reported as exponential [5, 8, 18, 22]. The high correlation coefficients between RBER and PE cycles in Figure 1 confirm that there is a correlation.

To study the effect of PE cycles on RBER in more detail, the two graphs in Figure 2 plot the median and the 95th percentile RBER against the number of PE cycles. We obtain these graphs by dividing all drive days in our data into different bins, based on their PE cycle count, and then determine the median and 95th percentile RBER across all days in a bin.

We observe that, as expected, RBER grows with the number of PE cycles, both in terms of median and 95th percentile RBER. However, the growth rate is slower than the commonly assumed exponential growth, and more closely resembles a linear increase. We verified this observation through curve fitting: we fit a linear model and an exponential model to the data and find that the linear model has a better fit than the exponential model.

The second interesting observation is that the RBER rates under wear-out vary greatly across drive models, even for models that have very similar RBER rates for low PE cycles. For example, the four MLC models start out with nearly identical RBER at very low PE cycles, but by the time they reach their PE cycle limit (3,000 for all MLC models) there is a 4X difference between the model with the highest and the lowest RBER.

Finally, we find that the increase in RBER is surprisingly smooth, even when a drive goes past its expected end of life (see for example model MLC-D with a PE cycle limit of 3,000). We note that accelerated life tests for the devices showed a rapid increase in RBER at around 3X the vendor's PE cycle limit, so vendors PE cycle limits seem to be chosen very conservatively.

4.2.2 RBER and age (beyond PE cycles)

Figure 1 shows a significant correlation between age, measured by the number of months a drive has been in the field, and RBER. However, this might be a spurious correlation, since older drives are more likely to have higher PE cycles and RBER is correlated with PE cycles.

To isolate the effect of age from that of PE cycle wear-out we group all drive months into bins using deciles of the PE cycle distribution as the cut-off between bins, e.g. the first bin contains all drive months up to the first decile of the PE cycle distribution, and so on. We verify that within each bin the correlation between PE cycles and RBER is negligible (as each bin only spans a small PE cycle range). We then compute the correlation coefficient between RBER and age separately for each bin. We perform this analysis separately for each model, so that any observed correlations are not due to differences between younger and older drive models, but purely due to younger versus older drives within the same model.

We observe that even after controlling for the effect of PE cycles in the way described above, there is still a significant correlation between the number of months a device has been in the field and its RBER (correlation coefficients between 0.2 and 0.4) for all drive models.

We also visualize the effect of drive age, by separating out drive days that were observed at a young drive age (less than one year) and drive days that were observed when a drive was older (4 years or more) and then plotting each group's RBER as a function of PE cycles. The results for one drive model (MLC-D) are shown in Fig-

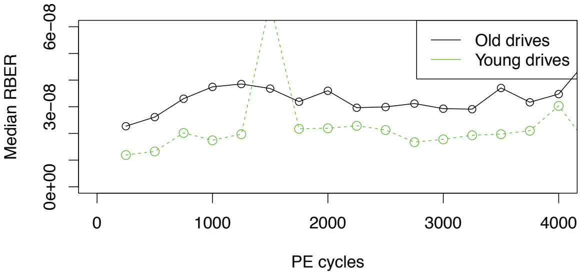


Figure 3: *RBER rates as a function of P/E cycles for young and old drives, showing that age has an effect on RBER, independently of P/E cycle induced wear-out.*

ure 3. We see a marked difference in the RBER rates between the two groups, across all PE cycles.

We conclude that age, as measured by days in the field, has a significant effect on RBER, independently of cell wear-out due to PE cycles. That means there must be other aging mechanisms at play, such as silicon aging.

4.2.3 RBER and workload

Bit errors are thought to be caused by one of four different mechanisms: retention errors, where a cell loses charge over time; read disturb errors, where a read operation disturbs the charge in a nearby cell; write disturb errors, where a write disturbs the charge in a nearby cell; or an incomplete erase errors, where an erase operation did not fully reset the cells in an erase block.

Errors that are of the latter three types (read disturb, write disturb, incomplete erase) will be correlated with workload, so understanding the correlation between RBER and workload helps us understand the prevalence of different error mechanisms. A recent field study [16] concludes that errors in the field are dominated by retention errors, while read disturb errors are negligible.

Figure 1 shows a significant correlation between the RBER in a given drive month and the number of read, write, and erase operations in the same month for some models (e.g. a correlation coefficient above 0.2 for model MLC-B and above 0.6 for model SLC-B). However, this might be a spurious correlation, as the per-month workload might be correlated with the total number of PE cycles seen so far. We use the same technique as described in Section 4.2.2 to isolate the effects of workload from that of PE cycles, by binning the drive months based on the prior PE cycles, and then determining correlation coefficients separately for each bin.

We find that the correlation between the number of read operations in a given drive month and the RBER in the same month does persist for models MLC-B and SLC-B, even when controlling for the PE cycles. We also repeat a similar analysis, where we isolate the effect of read operations from the count of concurrent write and erase operations, and find that for model SLC-B the correlation between RBER and read counts persists.

Figure 1 also showed a correlation between RBER and write and erase operations. We therefore repeat the same analysis we performed for read operations, for write and erase operations. We find that the correlation between RBER and write and erase operations is not significant, when controlling for PE cycles and read operations.

We conclude that there are drive models, where the effect of read disturb is significant enough to affect RBER. On the other hand there is no evidence for a significant impact of write disturb and incomplete erase operations on RBER.

4.2.4 RBER and lithography

Differences in feature size might partially explain the differences in RBER across models using the same technology, i.e. MLC or SLC. (Recall Table 1 for an overview of the lithography of different models in our study.) For example, the two SLC models with a 34nm lithography (models SLC-A and SLC-D) have RBER that are an order of magnitude higher than the two 50nm models (models SLC-B and SLC-C). For the MLC models, the only 43nm model (MLC-B) has a median RBER that is 50% higher than that of the other three models, which are all 50nm. Moreover, this difference in RBER increases to 4X with wear-out, as shown in Figure 2. Finally, their smaller lithography might explain the higher RBER for the eMLC drives compared to the MLC drives.

In summary, there is clear evidence that lithography affects RBER.

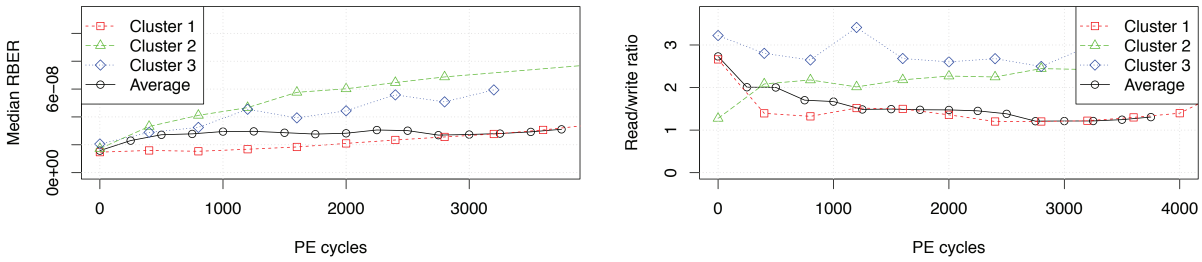
4.2.5 Presence of other errors

We investigate the relationship between RBER and other errors (such as uncorrectable errors, timeout errors, etc.), in particular whether RBER is higher in a month that also experiences other types of errors.

Figure 1 shows that while RBER experienced in the previous month is very predictive of future RBER (correlation coefficient above 0.8), there is no significant correlation between uncorrectable errors and RBER (see the right-most group of bars in Figure 1). Correlation coefficients are even lower for other error types (not shown in plot). We will further investigate the relationship between RBER and uncorrectable errors in Section 5.2.

4.2.6 Effect of other factors

We find evidence that there are factors with significant impact on RBER that our data does not directly account for. In particular, we observe that the RBER for a particular drive model varies depending on the cluster where the drive is deployed. One illustrative example is Figure 4, which shows RBER against PE cycles for drives of



(a)

(b)

Figure 4: *Figure (a) shows the median RBER rates as a function of PE cycles for model MLC-D for three different clusters. Figure (b) shows for the same model and clusters the read/write ratio of the workload.*

model MLC-D in three different clusters (dashed lines) and compares it to the RBER for this model across its entire population (solid line). We find that these differences persist even when we control for other factors, such as age or read count.

One possible explanation are differences in the type of workload in different clusters, as we observe that those clusters, whose workload has the highest read/write ratios, tend to be among the ones with the highest RBER. For example, Figure 4(b) shows the read/write ratio of model MLC-D. However, the read/write ratio does not explain differences across clusters for all models, so there might be other factors the data does not account for, such as environmental factors or other workload parameters.

4.3 RBER in accelerated life tests

Much academic work and also tests during the procurement phase in industry rely on accelerated life tests to derive projections for device reliability in the field. We are interested in how well predictions from such tests reflect field experience.

Analyzing results from tests performed during the procurement phase at Google, following common methods for test acceleration [17], we find that field RBER rates are significantly higher than the projected rates. For example, for model eMLC-A the median RBER for drives in the field (which on average reached 600 PE cycles at the end of data collection) is 1e-05, while under test the RBER rates for this PE cycle range were almost an order of magnitude lower and didn't reach comparable rates until more than 4,000 PE cycles. This indicates that it might be very difficult to accurately predict RBER in the field based on RBER estimates from lab tests.

We also observe that some types of error, seem to be difficult to produce in accelerated tests. For example, for model MLC-B, nearly 60% of drives develop uncorrectable errors in the field and nearly 80% develop bad blocks. Yet in accelerated tests none of the six devices under test developed any uncorrectable errors or bad blocks until the drives reached more than 3X of their PE cycle limit. For the eMLC models, more than 80% develop uncorrectable errors in the field, while in accelerated tests no device developed uncorrectable errors before 15,000 PE cycles.

We also looked at RBER reported in previous work, which relied on experiments in controlled environments. We find that previously reported numbers span a very large range. For example, Grupp et al. [10, 11] report RBER rates for drives that are close to reaching their PE cycle limit. For SLC and MLC devices with feature sizes similar to the ones in our work (25-50nm) the RBER in [11] ranges from 1e-08 to 1e-03, with most drive models experiencing RBER close to 1e-06. The three drive models in our study that reach their PE cycle limit experienced RBER between 3e-08 to 8e-08. Even taking into account that our numbers are lower bounds and in the absolute worst case could be 16X higher, or looking at the 95th percentile of RBER, our rates are significantly lower.

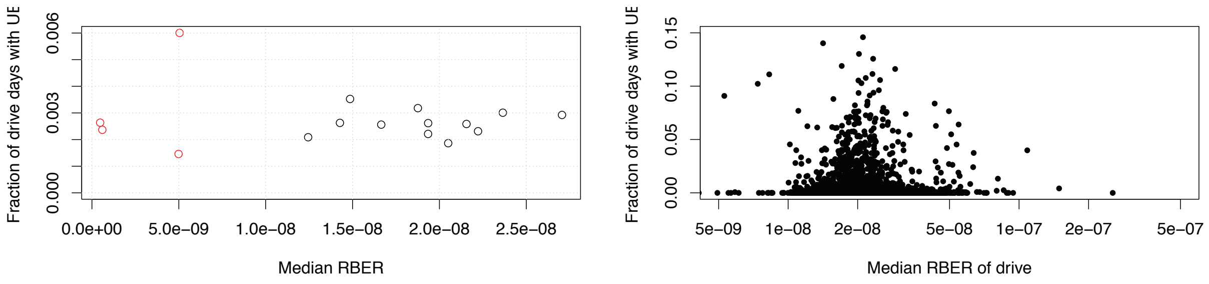
In summary, while the field RBER rates are higher than in-house projections based on accelerated life tests, they are lower than most RBER reported in other work for comparable devices based on lab tests. This suggests that predicting field RBER in accelerated life tests is not straight-forward.

5 Uncorrectable errors

Given the high prevalence of uncorrectable errors (UEs) we observed in Section 3, we study their characteristics in more detail in this section, starting with a discussion of what metric to use to measure UEs, their relationship with RBER and then moving to the impact of various factors on UEs.

5.1 Why UBER is meaningless

The standard metric used to report uncorrectable errors is UBER, i.e. the number of uncorrectable bit errors per



(a) (b)

Figure 5: *The two figures show the relationship between RBER and uncorrectable errors for different drive models (left) and for individual drives within the same model (right).*

total number of bits read. This metric makes the implicit assumption that the number of uncorrectable errors is in some way tied to the number of bits read, and hence should be normalized by this number.

This assumption makes sense for correctable errors, where we find that the number of errors observed in a given month is strongly correlated with the number of reads in the same time period (Spearman correlation coefficient larger than 0.9). The reason for this strong correlation is that one corrupted bit, as long as it is correctable by ECC, will continue to increase the error count with every read that accesses it, since the value of the cell holding the corrupted bit is not immediately corrected upon detection of the error (drives only periodically rewrite pages with corrupted bits).

The same assumption does not hold for uncorrectable errors. An uncorrectable error will remove the affected block from further usage, so once encountered it will not continue to contribute to error counts in the future. To formally validate this intuition, we used a variety of metrics to measure the relationship between the number of reads in a given drive month and the number of uncorrectable errors in the same time period, including different correlation coefficients (Pearson, Spearman, Kendall) as well as visual inspection. In addition to the number of uncorrectable errors, we also looked at the incidence of uncorrectable errors (e.g. the probability that a drive will have at least one within a certain time period) and their correlation with read operations.

We find no evidence for a correlation between the number of reads and the number of uncorrectable errors. The correlation coefficients are below 0.02 for all drive models, and graphical inspection shows no higher UE counts when there are more read operations.

As we will see in Section 5.4, also write and erase operations are uncorrelated with uncorrectable errors, so an alternative definition of UBER, which would normalize by write or erase operations instead of read operations, would not be any more meaningful either.

We therefore conclude that UBER is not a meaningful metric, except maybe in controlled environments where the number of read operations is set by the experimenter. If used as a metric in the field, UBER will artificially decrease the error rates for drives with high read count and artificially inflate the rates for drives with low read counts, as UEs occur independently of the number of reads.

5.2 Uncorrectable errors and RBER

RBER is relevant because it serves as a measure for general drive reliability, and in particular for the likelihood of experiencing UEs. Mielke et al. [18] first suggested to determine the expected rate of uncorrectable errors as a function of RBER. Since then many system designers, e.g. [2,8,15,23,24], have used similar methods to, for example, estimate the expected frequency of uncorrectable errors depending on RBER and the type of error correcting code being used.

The goal of this section is to characterize how well RBER predicts UEs. We begin with Figure 5(a), which plots for a number of first generation drive models [2] their median RBER against the fraction of their drive days with UEs. Recall that all models within the same generation use the same ECC, so differences between models are not due to differences in ECC. We see no correlation between RBER and UE incidence. We created the same plot for 95th percentile of RBER against UE probability and again see no correlation.

Next we repeat the analysis at the granularity of individual drives, i.e. we ask whether drives with higher RBER have a higher incidence of UEs. As an example, Figure 5(b) plots for each drive of model MLC-C its median RBER against the fraction of its drive days with UEs. (Results are similar for 95th percentile of RBER.) Again we see no correlation between RBER and UEs.

Finally, we perform an analysis at a finer time granularity, and study whether drive months with higher RBER are more likely to be months that experience a UE. Fig-

[2] Some of the 16 models in the figure were not included in Table 1, as they do not have enough data for some other analyses in the paper.

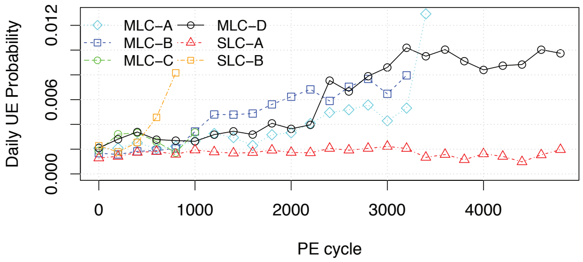


Figure 6: *The figure shows the daily probability of a drive experiencing an uncorrectable error as a function of the PE cycles the drive has experienced.*

ure 1 already indicated that the correlation coefficient between UEs and RBER is very low. We also experimented with different ways of plotting the probability of UEs as a function of RBER for visual inspection, and did not find any indication of a correlation.

In summary, we conclude that RBER is a poor predictor of UEs. This might imply that the failure mechanisms leading to RBER are different from those leading to UEs (e.g. retention errors in individual cells versus larger scale issues with the device).

5.3 Uncorrectable errors and wear-out

As wear-out is one of the main concerns with flash drives, Figure 6 shows the daily probability of developing an uncorrectable error as a function of the drive's P/E cycles. We observe that the UE probability continuously increases with age. However, as was the case for RBER, the increase is slower than commonly assumed: both visual inspection and curve fitting indicate that the UEs grow linearly with PE cycles rather than exponentially.

Also two other observations we made for RBER apply to UEs as well: First, there is no sharp increase in error probabilities after the PE cycle limit is reached, e.g. consider model MLC-D in Figure 6, whose PE cycle limit is 3,000. Second, error incidence varies across models, even within the same class. However, the differences are not as large as they were for RBER.

Finally, further supporting the observations we make in Section 5.2 we find that within a class of models (MLC versus SLC) the models with the lowest RBER rates for a given PE cycle count are not necessarily the ones with the lowest probabilities of UEs. For example, for 3,000 PE cycles model MLC-D had RBER rates 4X lower than that of MLC-B, yet its UE probability at the same PE cycles is slightly higher than that of MLC-B.

5.4 Uncorrectable errors and workload

For the same reasons that workload can affect RBER (recall Section 4.2.3) one might expect an effect on UEs. For example, since we observed read disturb errors affecting RBER, read operations might also increase the chance of uncorrectable errors.

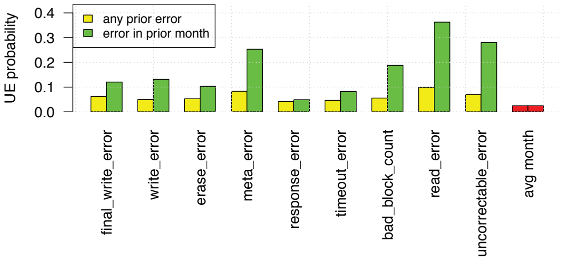


Figure 7: *The monthly probability of a UE as a function of whether there were previous errors of various types.*

We performed a detailed study of the effect of workload on UEs. However, as noted in Section 5.1, we find no correlation between UEs and the number of read operations. We repeated the same analysis for write and erase operations and again see no correlation.

Note that at first glance one might view the above observation as a contradiction to our earlier observation that uncorrectable errors are correlated with PE cycles (which one would expect to be correlated with the number of write and erase operations). However, in our analysis of the effect of PE cycles we were correlating the number of uncorrectable errors in a given month with the total number of PE cycles the drive has experienced in its life at that point (in order to measure the effect of wear-out). When studying the effect of workload, we look at whether drive months that had a higher read/write/erase count in *that particular month* also had a higher chance of uncorrectable errors in that particular month, i.e. we do not consider the cumulative count of read/write/erase operations.

We conclude that read disturb errors, write disturb errors or incomplete erase operations are not a major factor in the development of UEs.

5.5 Uncorrectable errors and lithography

Interestingly, the effect of lithography on uncorrectable errors is less clear than for RBER, where smaller lithography translated to higher RBER, as expected. Figure 6 shows, for example, that model SLC-B has a higher rate of developing uncorrectable errors than SLC-A, although SLC-B has the larger lithography (50nm compared to 34nm for model SLC-A). Also, the MLC model with the smallest feature size (model MLC-B), does not generally have higher rates of uncorrectable errors than the other models. In fact, during the first third of its life ($0-1,000$ PE cycles) and the last third ($> 2,200$ PE cycles) it has lower rates than, for example, model MLC-D. Recall, that all MLC and SLC drives use the same ECC, so these effects cannot be attributed to differences in the ECC.

Model name	MLC-A	MLC-B	MLC-C	MLC-D	SLC-A	SLC-B	SLC-C	SLC-D	eMLC-A	eMLC-B
Drives w/ bad blocks (%)	31.1	79.3	30.7	32.4	39.0	64.6	91.5	64.0	53.8	61.2
Median # bad block	2	3	2	3	2	2	4	3	2	2
Mean # bad block	772	578	555	312	584	570	451	197	1960	557
Drives w/ fact. bad blocks (%)	99.8	99.9	99.8	99.7	100	97.0	97.9	99.8	99.9	100
Median # fact. bad block	1.01e+03	7.84e+02	9.19e+02	9.77e+02	5.00e+01	3.54e+03	2.49e+03	8.20e+01	5.42e+02	1.71e+03
Mean # fact. bad block	1.02e+03	8.05e+02	9.55e+02	9.94e+02	3.74e+02	3.53e+03	2.55e+03	9.75e+01	5.66e+02	1.76e+03

Table 4: *Overview of prevalence of factory bad blocks and new bad blocks developing in the field*

Overall, we find that lithography has a smaller effect on uncorrectable errors than expected and a smaller effect than what we observed for RBER.

5.6 Other types of errors versus UEs

Next we look at whether the presence of other errors increases the likelihood of developing uncorrectable errors. Figure 7 shows the probability of seeing an uncorrectable error in a given drive month depending on whether the drive saw different types of errors at some previous point in its life (yellow) or in the previous month (green bars) and compares it to the probability of seeing an uncorrectable error in an average month (red bar).

We see that all types of errors increase the chance of uncorrectable errors. The increase is strongest when the previous error was seen recently (i.e. in the previous month, green bar, versus just at any prior time, yellow bar) and if the previous error was also an uncorrectable error. For example, the chance of experiencing an uncorrectable error in a month following another uncorrectable error is nearly 30%, compared to only a 2% chance of seeing an uncorrectable error in a random month. But also final write errors, meta errors and erase errors increase the UE probability by more than 5X.

In summary, prior errors, in particular prior uncorrectable errors, increase the chances of later uncorrectable errors by more than an order of magnitude.

6 Hardware failures

6.1 Bad blocks

Blocks are the unit at which erase operations are performed. In our study we distinguish blocks that fail in the field, versus factory bad blocks that the drive was shipped with. The drives in our study declare a block bad after a final read error, a write error, or an erase error, and consequently remap it (i.e. it is removed from future usage and any data that might still be on it and can be recovered is remapped to a different block).

The top half of Table 4 provides for each model the fraction of drives that developed bad blocks in the field, the median number of bad blocks for those drives that had bad blocks, and the average number of bad blocks among drives with bad blocks. We only include drives that were put into production at least four years ago, and consider only bad blocks that developed during the first four years in the field. The bottom half of the table provides statistics for factory bad blocks.

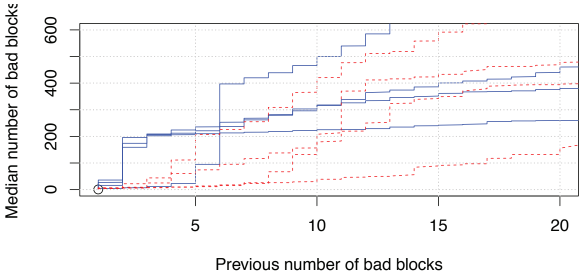


Figure 8: *The graph shows the median number of bad blocks a drive will develop, as a function of how many bad blocks it has already developed.*

6.1.1 Bad blocks developed in the field

We find that bad blocks are a frequent occurrence: Depending on the model, 30-80% of drives develop bad blocks in the field. A study of the cumulative distribution function (CDF) for the number of bad blocks per drive shows that most drives with bad blocks experience only a small number of them: the median number of bad blocks for drives with bad blocks is 2-4, depending on the model. However, if drives develop more than that they typically develop many more. Figure 8 illustrates this point. The figure shows the median number of bad blocks drives develop, as a function of how many bad blocks a drive has already experienced. The blue solid lines correspond to MLC models, while the red dashed lines correspond to the SLC models. We observe, in particular for MLC drives, a sharp increase after the second bad block is detected, when the median number of total bad blocks jumps to close to 200, i.e. 50% of those drives that develop two bad blocks will develop close to 200 or more bad blocks in total.

While we don't have access to chip-level error counts, bad block counts on the order of hundreds are likely due to chip failure, so Figure 8 indicates that after experiencing only a handful of bad blocks there is a large chance of developing a chip failure. This might imply potential for predicting chip failures, based on previous counts of

Model name	MLC-A	MLC-B	MLC-C	MLC-D	SLC-A	SLC-B	SLC-C	SLC-D	eMLC-A	eMLC-B
Drives w/ bad chips (%)	5.6	6.5	6.6	4.2	3.8	2.3	1.2	2.5	1.4	1.6
Drives w/ repair (%)	8.8	17.1	8.5	14.6	9.95	30.8	25.7	8.35	10.9	6.2
MTBRepair (days)	13,262	6,134	12,970	5,464	11,402	2,364	2,659	8,547	8,547	14,492
Drives replaced (%)	4.16	9.82	4.14	6.21	5.02	10.31	5.08	5.55	4.37	3.78

Table 5: *The fraction of drives for each model that developed bad chips, entered repairs and were replaced during the first four years in the field.*

bad blocks, and by potentially taking other factors (such as age, workload, PE cycles) into account.

Besides the frequency of bad blocks, we are also interested in how bad blocks are typically detected – in a write or erase operation, where the block failure is transparent to the user, or in a final read error, which is visible to the user and creates the potential for data loss. While we don't have records for individual block failures and how they were detected, we can turn to the observed frequencies of the different types of errors that indicate a block failure. Going back to Table 2, we observe that for all models, the incidence of erase errors and write errors is lower than that of final read errors, indicating that most bad blocks are discovered in a non-transparent way, in a read operation.

6.1.2 Factory bad blocks

While the discussion above focused on bad blocks that develop in the field, we note that nearly all drives ($>99\%$ for most models) are shipped with factory bad blocks and that the number of factory bad blocks can vary greatly between models, ranging from a median number of less than 100 for two of the SLC models, to more typical values in the range of 800 or more for the others. The distribution of factory bad blocks looks close to a normal distribution, with mean and median being close in value.

Interestingly, we find that the number of factory bad blocks is to some degree predictive of other issues the drive might develop in the field: For example, we observe that for all but one drive model the drives that have above the 95%ile of factory bad blocks have a higher fraction of developing new bad blocks in the field and final write errors, compared to an average drive of the same model. They also have a higher fraction that develops some type of read error (either final or non-final). The drives in the bottom 5%ile have a lower fraction of timeout errors than average.

We summarize our observations regarding bad blocks as follows: Bad blocks are common: 30-80% of drives develop at least one in the field. The degree of correlation between bad blocks in a drive is surprisingly strong: after only 2-4 bad blocks on a drive, there is a 50% chance that hundreds of bad blocks will follow. Nearly all drives come with factory bad blocks, and the number of factory bad blocks shows a correlation with the number of bad blocks the drive will develop in the field, as well as a few other errors that occur in the field.

6.2 Bad chips

The drives in our study consider a chip failed if more than 5% of its blocks have failed, or after the number of errors it has experienced within a recent time window exceed a certain threshold. Some commodity flash drives contain spare chips, so that the drive can tolerate a bad chip by remapping it to a spare chip. The drives in our study support a similar feature. Instead of working with spare chips, a bad chip is removed from further usage and the drive continues to operate with reduced capacity. The first row in Table 5 reports the prevalence of bad chips.

We observe that around 2-7% of drives develop bad chips during the first four years of their life. These are drives that, without mechanisms for mapping out bad chips, would require repairs or be returned to the vendor.

We also looked at the symptoms that led to the chip being marked as failed: across all models, around two thirds of bad chips are declared bad after reaching the 5% threshold on bad blocks, the other third after exceeding the threshold on the number of days with errors. We note that the vendors of all flash chips in these drives guarantee that no more than 2% of blocks on a chip will go bad while the drive is within its PE cycle limit. Therefore, the two thirds of bad chips that saw more than 5% of their blocks fail are chips that violate vendor specs.

6.3 Drive repair and replacement

A drive is being swapped and enters repairs if it develops issues that require manual intervention by a technician. The second row in Table 5 shows the fraction of drives for each model that enter repairs at some point during the first four years of their lives.

We observe significant differences in the repair rates between different models. While for most drive models 6-9% of their population at some point required repairs, there are some drive models, e.g. SLC-B and SLC-C, that enter repairs at significantly higher rates of 30% and 26%, respectively. Looking at the time between repairs (i.e. dividing the total number of drive days by the total number of repair events, see row 3 in Table 5) we see a range of a couple of thousand days between repairs for the worst models to nearly 15,000 days between repairs for the best models. We also looked at how often in their life drives entered repairs: The vast majority (96%) of drives that go to repairs, go there only once in their life.

We also check whether a drive returns to the fleet after visiting repairs or not, the latter indicating that it was permanently replaced. The fourth row in Table 5 shows that most models see around 5% of their drives permanently removed from the field within 4 years after being deployed, while the worst models (MLC-B and SLC-B) see around 10% of their drives replaced. For most models less than half as many drives are being replaced as being sent to repairs, implying that at least half of all repairs are successful.

7 Comparison of MLC, eMLC, and SLC drives

eMLC and SLC drives target the enterprise market and command a higher price point. Besides offering a higher write endurance, there is also the perception that the enterprise drives are higher-end drives, which are overall more reliable and robust. This section evaluates the accuracy of this perception.

Revisiting Table 3, we see that this perception is correct when it comes to SLC drives and their RBER, as they are orders of magnitude lower than for MLC and eMLC drives. However, Tables 2 and 5 show that SLC drives do not perform better for those measures of reliability that matter most in practice: SLC drives don't have lower repair or replacement rates, and don't typically have lower rates of non-transparent errors.

The eMLC drives exhibit higher RBER than the MLC drives, even when taking into account that the RBER for MLC drives are lower bounds and could be up to 16X higher in the worst case. However, these differences might be due to their smaller lithography, rather than other differences in technology.

Based on our observations above, we conclude that SLC drives are not generally more reliable than MLC drives.

8 Comparison with hard disk drives

An obvious question is how flash reliability compares to that of hard disk drives (HDDs), their main competitor. We find that when it comes to replacement rates, flash drives win. The *annual* replacement rates of hard disk drives have previously been reported to be 2-9% [19,20], which is high compared to the 4-10% of flash drives we see being replaced in a *4 year* period. However, flash drives are less attractive when it comes to their error rates. More than 20% of flash drives develop uncorrectable errors in a four year period, 30-80% develop bad blocks and 2-7% of them develop bad chips. In comparison, previous work [1] on HDDs reports that only 3.5% of disks in a large population developed bad sectors in a 32 months period – a low number when taking into account that the number of sectors on a hard disk is orders of magnitudes larger than the number of either blocks or chips on a solid state drive, and that sectors are smaller than blocks, so a failure is less severe.

In summary, we find that the flash drives in our study experience significantly lower replacement rates (within their rated lifetime) than hard disk drives. On the downside, they experience significantly higher rates of uncorrectable errors than hard disk drives.

9 Related work

There is a large body of work on flash chip reliability based on controlled lab experiments with a small number of chips, focused on identifying error patterns and sources. For example, some early work [3, 4, 9, 12–14, 17, 21] investigates the effects of retention, program and read disturbance in flash chips, some newer work [5–8] studies error patterns for more recent MLC chips. We are interested in behaviour of flash drives in the field, and note that our observations sometimes differ from those previously published studies. For example, we find that RBER is not a good indicator for the likelihood of uncorrectable errors and that RBER grows linearly rather than exponentially with PE cycles.

There is only one, very recently published study on flash errors in the field, based on data collected at Facebook [16]. Our study and [16] complement each other well, as they have very little overlap. The data in the Facebook study consists of a single snapshot in time for a fleet consisting of very young (in terms of the usage they have seen in comparison to their PE cycle limit) MLC drives and has information on uncorrectable errors only, while our study is based on per-drive time series data spanning drives' entire lifecycle and includes detailed information on different types of errors, including correctable errors, and different types of hardware failures, as well as drives from different technologies (MLC, eMLC, SLC). As a result our study spans a broader range of error and failure modes, including wear-out effects across a drive's entire life. On the other hand, the Facebook study includes the role of some factors (temperature, bus power consumption, DRAM buffer usage) that our data does not account for.

Our studies overlap in only two smaller points and in both of them we reach slightly different conclusions: (1) The Facebook paper presents rates of uncorrectable errors and studies them as a function of usage. They observe significant infant mortality (which they refer to as early detection and early failure), while we don't. Besides differences in burn-in testing at the two companies, which might affect infant mortality, the differences might also be due to the fact that the Facebook study presents more of a close-up view of a drive's early life (with no datapoints past a couple of hundred PE cycles for drives whose PE cycle limits are in the tens of thousands) while our view is more macroscopic spanning the entire lifecycle of a drive. (2) The Facebook study concludes that read disturb errors are not a significant factor in the field. Our view of read disturb errors is more differentiated, showing that while read disturb does not create uncorrectable errors, read disturb errors happen at a rate that is significant enough to affect RBER in the field.

10 Summary

This paper provides a number of interesting insights into flash reliability in the field. Some of these support common assumptions and expectations, while many were unexpected. The summary below focuses on the more surprising results and implications from our work:

• Between 20–63% of drives experience at least one uncorrectable error during their first four years in the field, making uncorrectable errors the most common non-transparent error in these drives. Between 2–6 out of 1,000 drive days are affected by them.

• The majority of drive days experience at least one correctable error, however other types of transparent errors, i.e. errors which the drive can mask from the user, are rare compared to non-transparent errors.

• We find that RBER (raw bit error rate), the standard metric for drive reliability, is not a good predictor of those failure modes that are the major concern in practice. In particular, higher RBER does not translate to a higher incidence of uncorrectable errors.

• We find that UBER (uncorrectable bit error rate), the standard metric to measure uncorrectable errors, is not very meaningful. We see no correlation between UEs and number of reads, so normalizing uncorrectable errors by the number of bits read will artificially inflate the reported error rate for drives with low read count.

• Both RBER and the number of uncorrectable errors grow with PE cycles, however the rate of growth is slower than commonly expected, following a linear rather than exponential rate, and there are no sudden spikes once a drive exceeds the vendor's PE cycle limit, within the PE cycle ranges we observe in the field.

• While wear-out from usage is often the focus of attention, we note that independently of usage the age of a drive, i.e. the time spent in the field, affects reliability.

• SLC drives, which are targeted at the enterprise market and considered to be higher end, are not more reliable than the lower end MLC drives.

• We observe that chips with smaller feature size tend to experience higher RBER, but are not necessarily the ones with the highest incidence of non-transparent errors, such as uncorrectable errors.

• While flash drives offer lower field replacement rates than hard disk drives, they have a significantly higher rate of problems that can impact the user, such as uncorrectable errors.

• Previous errors of various types are predictive of later uncorrectable errors. (In fact, we have work in progress showing that standard machine learning techniques can predict uncorrectable errors based on age and prior errors with an interesting accuracy.)

• Bad blocks and bad chips occur at a signicant rate: depending on the model, 30-80% of drives develop at least one bad block and and 2-7% develop at least one bad chip during the first four years in the field. The latter emphasizes the importance of mechanisms for mapping out bad chips, as otherwise drives with a bad chips will require repairs or be returned to the vendor.

• Drives tend to either have less than a handful of bad blocks, or a large number of them, suggesting that impending chip failure could be predicted based on prior number of bad blocks (and maybe other factors). Also, a drive with a large number of factory bad blocks has a higher chance of developing more bad blocks in the field, as well as certain types of errors.

11 Acknowledgements

We thank the anonymous reviewers and our shepherd Yiying Zhang for their valuable feedback. We would also like to thank the Platforms Storage Team at Google, as well as Nikola Janevski and Wan Chen for help with the data collection, and Chris Sabol, Tomasz Jeznach, and Luiz Barroso for feedback on earlier drafts of the paper. Finally, the first author would like to thank the Storage Analytics team at Google for hosting her in the summer of 2015 and for all their support.

References

[1] 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* (New York, NY, USA, 2007), SIGMETRICS '07, ACM, pp. 289–300.

[2] Balakrishnan, M., Kadav, A., Prabhakaran, V., and Malkhi, D. Differential RAID: Rethinking RAID for SSD reliability. *Trans. Storage 6*, 2 (July 2010), 4:1–4:22.

[3] Belgal, H. P., Righos, N., Kalastirsky, I., Peterson, J. J., Shiner, R., and Mielke, N. A new reliability model for post-cycling charge retention of flash memories. In *Reliability Physics Symposium Proceedings, 2002. 40th Annual* (2002), IEEE, pp. 7–20.

[4] Brand, A., Wu, K., Pan, S., and Chin, D. Novel read disturb failure mechanism induced by flash cycling. In *Reliability*

Physics Symposium, 1993. 31st Annual Proceedings., International (1993), IEEE, pp. 127–132.

[5] Cai, Y., Haratsch, E. F., Mutlu, O., and Mai, K. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012* (2012), IEEE, pp. 521–526.

[6] Cai, Y., Haratsch, E. F., Mutlu, O., and Mai, K. Threshold voltage distribution in MLC NAND flash memory: Characterization, analysis, and modeling. In *Proceedings of the Conference on Design, Automation and Test in Europe* (2013), EDA Consortium, pp. 1285–1290.

[7] Cai, Y., Mutlu, O., Haratsch, E. F., and Mai, K. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on* (2013), IEEE, pp. 123–130.

[8] Cai, Y., Yalcin, G., Mutlu, O., Haratsch, E. F., Cristal, A., Unsal, O. S., and Mai, K. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *IEEE 30th International Conference on Computer Design (ICCD 2012)* (2012), IEEE, pp. 94–101.

[9] Degraeve, R., Schuler, F., Kaczer, B., Lorenzini, M., Wellekens, D., Hendrickx, P., Van Duuren, M., Dormans, G., Van Houdt, J., Haspeslagh, L., et al. Analytical percolation model for predicting anomalous charge loss in flash memories. *Electron Devices, IEEE Transactions on 51*, 9 (2004), 1392–1400.

[10] 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 *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO 42, ACM, pp. 24–33.

[11] Grupp, L. M., Davis, J. D., and Swanson, S. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 2–2.

[12] Hur, S., Lee, J., Park, M., Choi, J., Park, K., Kim, K., and Kim, K. Effective program inhibition beyond 90nm NAND flash memories. *Proc. NVSM* (2004), 44–45.

[13] Joo, S. J., Yang, H. J., Noh, K. H., Lee, H. G., Woo, W. S., Lee, J. Y., Lee, M. K., Choi, W. Y., Hwang, K. P., Kim, H. S., et al. Abnormal disturbance mechanism of sub-100 nm NAND flash memory. *Japanese journal of applied physics 45*, 8R (2006), 6210.

[14] Lee, J.-D., Lee, C.-K., Lee, M.-W., Kim, H.-S., Park, K.-C., and Lee, W.-S. A new programming disturbance phenomenon in NAND flash memory by source/drain hot-electrons generated by GIDL current. In *Non-Volatile Semiconductor Memory Workshop, 2006. IEEE NVSMW 2006. 21st* (2006), IEEE, pp. 31–33.

[15] Liu, R., Yang, C., and Wu, W. Optimizing NAND flash-based SSDs via retention relaxation. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012* (2012), p. 11.

[16] Meza, J., Wu, Q., Kumar, S., and Mutlu, O. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2015), SIGMETRICS '15, ACM, pp. 177–190.

[17] Mielke, N., Belgal, H. P., Fazio, A., Meng, Q., and Righos, N. Recovery effects in the distributed cycling of flash memories. In *Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International* (2006), IEEE, pp. 29–35.

[18] Mielke, N., Marquart, T., Wu, N., Kessenich, J., Belgal, H., Schares, E., Trivedi, F., Goodness, E., and Nevill, L. Bit error rate in NAND flash memories. In *2008 IEEE International Reliability Physics Symposium* (2008).

[19] Pinheiro, E., Weber, W.-D., and Barroso, L. A. Failure trends in a large disk drive population. In *FAST* (2007), vol. 7, pp. 17–23.

[20] 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 *FAST* (2007), vol. 7, pp. 1–16.

[21] Suh, K.-D., Suh, B.-H., Lim, Y.-H., Kim, J.-K., Choi, Y.-J., Koh, Y.-N., Lee, S.-S., Suk-Chon, S.-C., Choi, B.-S., Yum, J.-S., et al. A 3.3 v 32 mb NAND flash memory with incremental step pulse programming scheme. *Solid-State Circuits, IEEE Journal of 30*, 11 (1995), 1149–1156.

[22] Sun, H., Grayson, P., and Wood, B. Quantifying reliability of solid-state storage from multiple aspects. In *SNAPI* (2011).

[23] Wu, G., He, X., Xie, N., and Zhang, T. Exploiting workload dynamics to improve ssd read latency via differentiated error correction codes. *ACM Trans. Des. Autom. Electron. Syst. 18*, 4 (Oct. 2013), 55:1–55:22.

[24] Zambelli, C., Indaco, M., Fabiano, M., Di Carlo, S., Prinetto, P., Olivo, P., and Bertozzi, D. A cross-layer approach for new reliability-performance trade-offs in MLC NAND flash memories. In *Proceedings of the Conference on Design, Automation and Test in Europe* (San Jose, CA, USA, 2012), DATE '12, EDA Consortium, pp. 881–886.

Opening the Chrysalis: On the Real Repair Performance of MSR Codes

Lluis Pamies-Juarez*, Filip Blagojević*, Robert Mateescu*,
Cyril Guyot*, Eyal En Gad**, and Zvonimir Bandic*

*WD Research, **University of Southern California
{*lluis.pamies-juarez, filip.blagojevic, robert.mateescu, cyril.guyot, zvonimir.bandic*}*@hgst.com,*
engad@usc.edu

Abstract

Large distributed storage systems use erasure codes to reliably store data. Compared to replication, erasure codes are capable of reducing storage overhead. However, repairing lost data in an erasure coded system requires reading from many storage devices and transferring over the network large amounts of data. Theoretically, Minimum Storage Regenerating (MSR) codes can significantly reduce this repair burden. Although several explicit MSR code constructions exist, they have not been implemented in real-world distributed storage systems. We close this gap by providing a performance analysis of Butterfly codes, systematic MSR codes with optimal repair I/O. Due to the complexity of modern distributed systems, a straightforward approach does not exist when it comes to implementing MSR codes. Instead, we show that achieving good performance requires to vertically integrate the code with multiple system layers. The encoding approach, the type of inter-node communication, the interaction between different distributed system layers, and even the programming language have a significant impact on the code repair performance. We show that with new distributed system features, and careful implementation, we can achieve the theoretically expected repair performance of MSR codes.

1 Introduction

Erasure codes are becoming the redundancy mechanism of choice in large scale distributed storage systems. Compared to replication, erasure codes allow reduction in storage overhead, but at a higher repair cost expressed through excessive read operations and expensive computation. Increased repair costs negatively affect the Mean Time To Data Loss (MTTDL) and data durability.

New coding techniques developed in recent years improve the repair performance of classical erasure codes (e.g. Reed-Solomon codes) by reducing excessive network traffic and storage I/O. Regenerating Codes (RGC) [13] and Locally Repairable Codes (LRC) [19] are the main representatives of these advanced coding techniques. RGCs achieve an optimal trade-off between the storage overhead and the amount of data transferred (*repair traffic*) during the repair process. LRCs offer an optimal trade-off between storage overhead, fault tolerance and the number of nodes involved in repairs. In both cases, the repairs can be performed with a fraction of the read operations required by classical codes.

Several explicit LRC code constructions have been demonstrated in real world production systems [20, 35, 28]. LRCs are capable of reducing the network and storage traffic during the repair process, but the improved performance comes at the expense of requiring extra storage overhead. In contrast, for a fault tolerance equivalent to that of a Reed-Solomon code, RGCs can significantly reduce repair traffic [28] without increasing storage overhead. This specifically happens for a subset of RGCs operating at the Minimum Storage Regenerating tradeoff point, i.e. MSR codes. At this tradeoff point the storage overhead is minimized over repair traffic. Unfortunately, there has been little interest in using RGCs in real--world scenarios. RGC constructions of interest, those with the storage overhead below 2×, require either encoding/decoding operations over an exponentially growing finite field [8], or an exponential increase in number of sub-elements per storage disk [14, 31]. Consequently, implementation of RGCs in production systems requires dealing with complex and bug-prone algorithms. In this study we focus on managing the drawbacks of RGC-MSR codes. We present the first MSR implementation with low-storage overhead (under 2×), and we explore the design space of distributed storage systems and characterize the most important design decisions affecting the implementation and performance of MSRs.

Practical usage of MSR codes equals the importance of a code design. For example, fine-grain read accesses introduced by MSR codes may affect performance negatively and reduce potential code benefits. Therefore, understanding the advantages of MSR codes requires characterizing not only the theoretical performance, but also observing the effect the code has on real-world distributed systems. Because of the complex and multi--layer design of distributed storage systems, it is also important to capture the interaction of MSR codes with various system layers, and pinpoint the system features that improve/degrade coding performance.

We implement an MSR code in two mainstream distributed storage systems: HDFS and Ceph. These are the two most widely used systems in industry and academia, and are also based on two significantly different distributed storage models. Ceph does *online* encoding while data is being introduced to the system. HDFS performs encoding as a *batch job*. Moreover, Ceph applies erasure codes in a per-object basis whereas HDFS does it on groups of objects of the same size. And finally, Ceph has an open interface to incorporate new code implementations in a pluggable way, while HDFS has a monolithic approach where codes are embedded within the system.

The differences between HDFS and Ceph allow us to cover an entire range of system design decisions that one needs to make while designing distributed storage systems. The design observations presented in this study are intended for designing future systems that are built to allow effortless integration of MSR codes. To summarize, this paper makes the following contributions: (i) We design a recursive Butterfly code construction —a two-parity MSR code— and implement it in two real-world distributed storage systems: HDFS and Ceph. Compared to other similar codes, Butterfly only requires XOR operations for encoding/decoding, allowing for more efficient computation. To the best of our knowledge, these are the first implementations of a low overhead MSR code in real-world storage systems. (ii) We compare two major approaches when using erasure codes in distributed storage systems: *online* and *batch*-based encoding and point the major tradeoffs between the two approaches. (iii) We examine the performance of Butterfly code and draw a comparison between the theoretical results of MSR codes and the performance achievable in real systems. We further use our observations to suggest appropriate distributed system design that allows best usage of MSR codes. Our contributions in this area include *communication vectorization* and a plug-in interface design for pluggable MSR encoders/decoders.

2 Background

In this section we introduce erasure coding in large-scale distributed storage systems. In addition we provide a short overview of HDFS and Ceph distributed filesystems and their use of erasure codes.

2.1 Coding for Distributed Storage

Erasure codes allow reducing the storage footprint of distributed storage systems while providing equivalent or even higher fault tolerance guarantees than replication. Traditionally, the most common type of codes used in distributed systems were Reed-Solomon (RS) codes. RS are well-known maximum distance separable (MDS) codes used in multiple industrial contexts such as optical storage devices or data transmission. In a nutshell, Reed-Solomon codes split each data object into k chunks and generate r linear combinations of these k chunks. Then, the $n = k + r$ total chunks are stored into n storage devices. Finally, the original object can be retrieved as long as k out of the n chunks are available.

In distributed storage systems, achieving long MTTDL and high data durability requires efficient data repair mechanisms. The main drawback of traditional erasure codes is that they have a costly repair mechanism that compromises durability. Upon a single chunk failure, the system needs to read k out of n chunks in order to regenerate the missing part. The repair process entails a k to 1 ratio between the amount of data read (and transferred) and the amount of data regenerated. Regenerating Codes (RGC) and Locally Repairable Codes (LRC) are two family of erasure codes that can reduce the data and storage traffic during the regeneration. LRCs reduce the number of storage devices accessed during the regeneration of a missing chunk. However, this reduction results in losing the MDS property, and hence, relaxing the fault tolerance guarantees of the code. On the other hand, RGCs aim at reducing the amount of data transferred from each of the surviving devices, at the expense of increased number of devices contacted during repair. Additionally, when RGCs minimize the repair traffic without any additional storage overhead, we say that the code is a Minimum Storage Regenerating (MSR) code.

LRCs have been demonstrated and implemented in production environments [20, 35, 28]. However, the use of LRCs in these systems reduces the fault tolerance guarantees of equivalent traditional erasure codes, and cannot achieve the minimum theoretical repair traffic described by RGCs. Therefore, RGCs seem to be a better option when searching for the best tradeoff between storage overhead and repair performance in distributed storage systems. Several MSR codes constructions exist for rates smaller that 1/2 (i.e. $r \geq k$) [26, 23, 29, 30], however, designing codes for higher rates (more storage efficient regime) is far more complex. Although it has been shown that codes for arbitrary (n,k) values can be asymptotically achieved [9, 30], explicit finite code constructions require either storing an exponentially growing number of elements per storage device [7, 31, 24, 14], or increasing the finite field size [27]. To the best of our knowledge, Butterfly codes [14] are the only codes that allow a two-parity erasure code ($n-k=2$) over a small field (i.e. GF(2)), and hence, they incur low computational overhead. The relatively simple design and low computational overhead make Butterfly codes a good candidate for exploring the challenges of implementing a MSR code in real distributed storage systems.

2.2 Hadoop Filesystem

Hadoop is a scalable runtime designed for managing large-scale computation/storage systems. Hadoop supports a map-reduce computational model and therefore is very suitable for algorithms that target processing of large amounts of data. The Hadoop filesystem (HDFS) is its default storage backend, and was initially developed as an open source version of the Google filesystem [17] (GFS), containing many of the features initially designed for GFS. HDFS is currently one of the most widely used distributed storage systems in industrial and academic deployments.

In HDFS there are two types of physical nodes: the Namenode server and multiple Datanode servers. The Namenode server contains various metadata as well as the location information about all data blocks residing in HDFS, whereas Datanode servers contain the actual blocks of data. The "centralized metadata server" architecture lowers the system design complexity, but poses certain drawbacks: (i) Limited metadata storage This problem has been addressed by recent projects (such as Hadoop Federation [4]) that allow multiple namespaces per HDFS cluster. (ii) Single point of failure - In case of the Namenode failure, the entire system is unaccessible until the Namenode is repaired. Recent versions of HDFS address this problem by introducing multiple redundant Namenodes, allowing fast failover in case the Namenode fails.

Starting with the publicly available Facebook's implementation of a Reed-Solomon code for HDFS [2], Hadoop allows migration of replicated data into more storage efficient encoded format. The erasure code is usually used to reduce the replication factor once the data access frequency reduces. Hence, the encoding process in HDFS is not a real-time task, instead it is performed in the background, as a batch job. While the batch-based approach provides low write latency, it also requires additional storage where the intermediate data resides before being encoded.

2.3 Ceph's Distributed Object Store

Ceph [32] is an open source distributed storage system with a decentralized design and no single point of failure. Like HDFS, Ceph is self-healing and a self-managing system that can guarantee high-availability and consistency with little human intervention. RADOS [34] (Reliable, Autonomic Distributed Object Store) is Ceph's core component. It is formed by a set of daemons and libraries that allow users accessing an object-based storage system with partial and complete read/writes, and snapshot capabilities. RADOS has two kinds of daemons: monitors (MONs), that maintain consistent metadata, and object storage devices (OSDs). A larger cluster of OSDs is responsible to store all data objects and redundant replicas. Usually a single OSD is used to manage a single HDD, and typically multiple OSDs are collocated in a single server.

RADOS storage is logically divided into object containers named *pools*. Each pool has independent access control and redundancy policies, providing isolated namespaces for users and applications. Internally, and transparent to the user/application, pools are divided into subsets of OSDs named *placement groups*. The OSDs in a placement group run a distributed leader-election to elect a *Primary* OSD. When an object is stored into a pool, it is assigned to one placement group and uploaded to its Primary OSD. The Primary OSD is responsible to redundantly store the object within the placement group. In a replicated pool this means forwarding the object to all the other OSDs in the group. In an erasure encoded pool, the Primary splits and encodes the object, uploading the corresponding chunks to the other OSDs in the group. Hence, the encoding process in Ceph is performed as real-time job, i.e. the data is encoded while being introduced into the system. The placement group size directly depends on the number of replicas or the length of the code used. OSDs belong to multiple placement groups, guaranteeing good load balancing without requiring large amount of computing resources. Given the cluster map, the pool policies, and a set of fault domain constraints, RADOS uses a consistent hashing algorithm [33] to assign OSDs to placement groups, and map object names to placement groups within a pool.

3 Butterfly Codes

Vector codes are a generalization of classical erasure codes where k α-dimensional data vectors are encoded into a codeword of n α-dimensional redundant vectors, for $n>k$. As it happens for classical erasure codes, we say that a vector code is systematic if the original k vectors form a subset of the n codeword vectors, that is, the codeword only adds $n-k$ redundant vectors. In this paper we refer to the codeword vectors as code columns, and to the vector components as column elements.

Butterfly Codes are an MDS vector code construction for two-parities (i.e. $n-k=2$) of an explicit Regenerating Code operating at the minimum storage regenerating (MSR) point. This means that to repair a single disk failure, Butterfly codes require to transfer $1/2$ of all the remaining data, which is optimal. Additionally, Butterfly codes are binary vector codes defined over $GF(2)$, allowing implementation of encoding and decoding operations by means of simple exclusive-or operations.

A preliminary construction of the Butterfly code was presented earlier [14], and in this section we provide a new recursive construction of the code. Compared to the original construction, the recursive approach has a simplified design that results in a simpler implementation. Furthermore, the recursive design partitions the problem in a way that allows for a better reuse of precomputed values, leading to better cache locality. Due to space limitations we omit the exhaustive cache behavior analysis.

3.1 Butterfly Encoder

Let D_k be a matrix of boolean values of size $2^{k-1} \times k$, for $k \geq 2$. D_k represents a data object to be encoded and stored in the distributed storage system. For the purpose of describing the encoding/decoding process, we represent D_k by the following components:

$$D_k = \begin{bmatrix} \mathbf{a} & A \\ \mathbf{b} & B \end{bmatrix}, \tag{1}$$

where A and B are $2^{k-2} \times k-1$ boolean matrices, and $\mathbf{a}$ and $\mathbf{b}$ are column vectors of 2^{k-2} elements.

Let D_k^j be the jth column of D_k, $j \in \{0, \ldots, k-1\}$. Therefore, the matrix D_k can be written as a vector of k columns $D_k = (D_k^{k-1}, \ldots, D_k^0)$. From a traditional erasure code perspective each of the columns is an element of GF(2^{k-1}), and after encoding we get a systematic codeword $C_k = (D_k^{k-1}, \ldots, D_k^0, H, B)$, where H and B are two vector columns representing the horizontal and butterfly parities respectively.

To describe how to generate H and B, we define two functions such that $H = \mathcal{H}(D_k)$ and $B = \mathcal{B}(D_k)$:

- if $k = 2$, then:

$$\mathcal{H}\left(\begin{bmatrix} d_0^1 & d_0^0 \\ d_1^1 & d_1^0 \end{bmatrix}\right) = \begin{bmatrix} d_0^1 \oplus d_0^0 \\ d_1^1 \oplus d_1^0 \end{bmatrix}; \tag{2}$$

$$\mathcal{B}\left(\begin{bmatrix} d_0^1 & d_0^0 \\ d_1^1 & d_1^0 \end{bmatrix}\right) = \begin{bmatrix} d_1^1 \oplus d_0^0 \\ d_0^1 \oplus d_0^0 \oplus d_1^0 \end{bmatrix}. \tag{3}$$

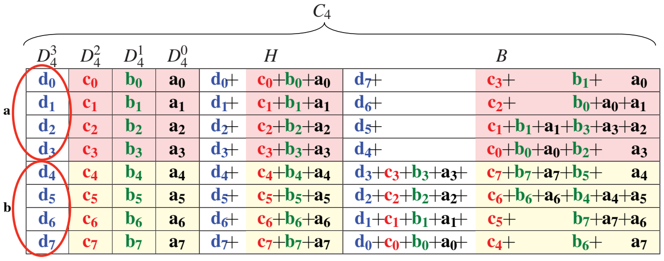
- if $k > 2$, then:

$$\mathcal{H}(D_k) = \begin{bmatrix} \mathbf{a} \oplus \mathcal{H}(A) \\ P_{k-1}\left[P_{k-1}\mathbf{b} \oplus \mathcal{H}(P_{k-1}B)\right] \end{bmatrix}; \tag{4}$$

$$\mathcal{B}(D_k) = \begin{bmatrix} P_{k-1}\mathbf{b} \oplus \mathcal{B}(A) \\ P_{k-1}\left[\mathbf{a} \oplus \mathcal{H}(A) \oplus \mathcal{B}(P_{k-1}B)\right] \end{bmatrix}, \tag{5}$$

where P_k represents a $k \times k$ permutation matrix where the counter-diagonal elements are one and all other elements are zero. Notice that left-multiplication of a vector or a matrix by P_k flips the matrix vertically.

It is interesting to note that the double vertical flip in (4) is intentionally used to simultaneously compute $\mathcal{H}$ and $\mathcal{B}$ over the same data D_k. Because of the double vertical flip, the recursion can be simplified, and encoding of D_k can be done by encoding A and $P_{k-1}B$. In Figure 1 we show an example of the recursive encoding for $k = 4$.



	C_4							
	D_4^3	D_4^2	D_4^1	D_4^0	H		B	
a	d_0	c_0	b_0	a_0	d_0+	$c_0+b_0+a_0$	d_7+	c_3+ b_1+ a_0
	d_1	c_1	b_1	a_1	d_1+	$c_1+b_1+a_1$	d_6+	c_2+ $b_0+a_0+a_1$
	d_2	c_2	b_2	a_2	d_2+	$c_2+b_2+a_2$	d_5+	$c_1+b_1+a_1+b_3+a_3+a_2$
	d_3	c_3	b_3	a_3	d_3+	$c_3+b_3+a_3$	d_4+	$c_0+b_0+a_0+b_2+$ a_3
b	d_4	c_4	b_4	a_4	d_4+	$c_4+b_4+a_4$	$d_3+c_3+b_3+a_3+$	$c_7+b_7+a_7+b_5+$ a_4
	d_5	c_5	b_5	a_5	d_5+	$c_5+b_5+a_5$	$d_2+c_2+b_2+a_2+$	$c_6+b_6+a_6+b_4+a_4+a_5$
	d_6	c_6	b_6	a_6	d_6+	$c_6+b_6+a_6$	$d_1+c_1+b_1+a_1+$	c_5+ $b_7+a_7+a_6$
	d_7	c_7	b_7	a_7	d_7+	$c_7+b_7+a_7$	$d_0+c_0+b_0+a_0+$	c_4+ b_6+ a_7

Figure 1: Butterfly codeword for $k = 4$, C_4. One can observe how C_4 can be computed by recursively encoding submatrix A (red highlight) and B (yellow highlight) from (1) and adding the extra non-highlighted elements.

3.2 Butterfly Decoder

In this section we show that Butterfly code can decode the original data matrix when any two of the codeword columns are missing, and hence it is an MDS code.

Theorem 1 (MDS)**.** *The Butterfly code can recover from the loss of any two columns (i.e. two erasures).*

Proof. The proof is by induction over the number of columns, k. In the *base case*, $k = 2$, one can carefully verify from (2) and (3) that the code can recover from the loss of any two columns. The *inductive step* proceed as follows. Let's assume that the Butterfly construction gives an MDS code for $k-1$ columns, for $k > 2$. We will prove that the construction for k columns is also MDS. We distinguish the following cases:

(1) The two parity nodes are lost. In this case we encode them again through $\mathcal{H}$ and $\mathcal{B}$ functions.

(2) One of the parities is lost, along with one data column. In this case we can use the remaining parity node to decode the lost data column, and then re-encode the missing parity node.

(3) Two data columns are lost, neither of which is the leftmost column. In this case we can generate from the parity columns the vectors $\mathcal{H}(A)$, $\mathcal{B}(A)$, by XOR-ing $\mathbf{a}$ and $P_{k-1}\mathbf{b}$. By using the inductive hypothesis, we can recover the top half of the missing columns (which is part of the A matrix). Similarly, we can generate by simple XOR the values $\mathcal{H}(P_{k-1}B)$ and $\mathcal{B}(P_{k-1}B)$. By the induction hypothesis we can recover the bottom half of the missing columns (which is part of the B matrix).

(4) The leftmost column along with another data column D_k^j, $j \neq k-1$, are lost. From the bottom half of the butterfly parity $\mathcal{B}(D_k)$ we can obtain $\mathcal{B}(P_{k-1}B)$, and then decode the bottom half of D_k^j. From the bottom half of the horizontal parity $\mathcal{H}(D_k)$ we can now decode $\mathbf{b}$. Following the decoding chain, from the top half of the butterfly parity $\mathcal{B}(D_k)$ we can obtain $\mathcal{B}(A)$, and then decode the top half of D_k^j. Finally, from the top half of the horizontal parity $\mathcal{H}(D_k)$ we can obtain $\mathbf{a}$. □

Theorem 2 (optimal regeneration)**.** *In the case of one failure, the lost column can be regenerated by communicating an amount of data equal to $1/2$ of the remaining data (i.e., $1/2$ of $k+1$ columns). If the lost column is not the butterfly parity, the amount of communicated data is exactly equal to the amount read form surviving disks (i.e. optimal I/O access).*

Due to space constrains we do not provide a proof here. Instead, in the next section we provide the details required to regenerate any of the codeword columns.

3.3 Single Column Regeneration

The recovery of a single column falls under four cases. Note that in all of them, the amount of data that is transferred is optimal, and equal to half of the remaining data. Moreover, the amount of data that is accessed (read) is also optimal (and equal to the data that is transferred), except in case (4) when we recover the butterfly parity. Case (1) is the most common. The data to be transferred is selected by algebraic expressions, but there are more intuitive ways to understand the process. The indeces correspond to locations in the butterfly parity that do not require the additional elements; similarly, they correspond to inflexion points in the butterfly lines (v-points); also they correspond to 0 value for bit $j-1$ of the binary representation of numbers ordered by the reflected Gray code. Finally, the recovery in case (4) is based on a self-duality of the butterfly encoding.

(1) One column from $\{D_k^1, \ldots, D_k^{k-1}\}$ is lost Let D_k^j be the lost columnn. Every remaining column (systematic data and parities), will access and transfer the elements in position i for which $\left\lfloor \frac{i}{2^{j-1}} \right\rfloor \equiv 0 \mod 4$, or $\left\lfloor \frac{i}{2^{j-1}} \right\rfloor \equiv 3 \mod 4$. Let D_{k-1} be the matrix of size $(k-1) \times 2^{k-2}$ formed from the transmitted systematic data, and H_{k-1}, B_{k-1} the columns of size 2^{k-2} formed from the transmitted parity information. Let $h = \mathcal{H}(D_{k-1}) \oplus H_{k-1}$ and $b = \mathcal{B}(D_{k-1}) \oplus B_{k-1}$ (i.e., we use butterfly encoding on the matrix D_{k-1}). The data lost from D_j is now contained by h and b. More precisely, for $i \in \{0, \ldots, 2^{k-2}\}$, let $p = \left\lfloor \frac{i+2^{j-1}}{2^j} \right\rfloor 2^j + i$, and let $r = p \mod 2^j$. Then $D_k^j(p) \leftarrow h(i)$, and $D_k^j(p-2r+2^j-1) \leftarrow b(i)$.

(2) Column D_k^0 is lost In this case, the columns $D_k^1, \ldots, D_k^{k-1}, H$ will access and transfer the elements with even index, and the column B will access and transfer the elements with odd index. Similar to case (1), the vectors h and b are obtained by applying butterfly encoding, and they provide the even, repectively odd, index elements of the lost column.

(3) First parity column H is lost All the remaining columns access and transfer their lower half, namely all the elements with index $i \in \{2^{k-2}, \ldots, 2^{k-1}-1\}$. The horizontal parity over the systematic transmitted data provides the lower half of H, and the butterfly parity over $D_{k-1}^0, \ldots, D_{k-1}^{k-2}$ XOR-ed with data from B will provide the top half of H.

(4) Second parity column B is lost In this case D_k^{k-1} will access and transfer its top half, while H will do the same with its bottom half. The rest of the columns $D_k^0, \ldots, D_k^{k-2}$ will access all of their data, but they will perform XOR operations and only transfer an amount of data equal to half of their size. Each D_k^j for $j \neq k-1$ will compute and transfer values equal to their contributions in the bottom half of B. Therefore a simple XOR operation between the data transferred from the systematic columns will recover the bottom half of B. Interestingly, computing a butterfly parity over the data transferred from D_k^j, where $j \neq k-1$, and XOR-ing it correspondingly with the bottom half of H will recover the top half of B.

4 Butterfly Codes in HDFS

To avoid recursion in Java, and possible performance drawbacks due to non-explicit memory management, in HDFS we implement an iterative version of Butterfly [14]. Our implementation of Butterfly code in HDFS is based on publicly available Facebook's Hadoop [2] version. In this section we provide implementation and optimization details of our Butterfly implementation.

4.1 Erasure Coding in HDFS

We use the Facebook HDFS implementation as a starting point for the Butterfly implementation. Facebook version of Hadoop contains two daemons, RaidNode and BlockFixer, that respectively create parity files and fix corrupted data. Once inserted into the HDFS, all files are initially replicated according to the configured replication policy. The RaidNode schedules map-reduce jobs for erasure encoding the data. The encoding map-reduce jobs take groups of k newly inserted chunks, and generate $n-k$ parity chunks, as presented in Figure 2. The parity chunks are then stored back in HDFS, and the replicas can be garbage collected. Lost or corrupted data is detected and scheduled for repair by the BlockFixer daemon. The repair is performed using map-reduce decode tasks. Upon decoding completion, the reconstructed symbol is stored back to HDFS.

4.2 Butterfly Implementation in HDFS

The encoding and repair process in HDFS-Butterfly follows a 4-step protocol: (i) in the first step the encoding/decoding task determines the location of the data blocks that are part of the k symbol message; (ii) the sec-

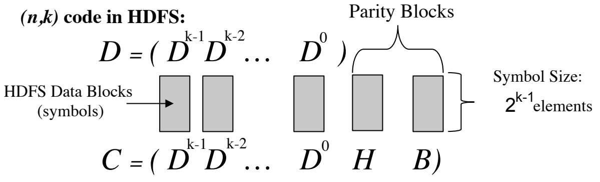

(n,k) code in HDFS:
Parity Blocks
D = (D^k-1 D^k-2 ... D^0)
HDFS Data Blocks (symbols)
Symbol Size: 2^k-1 elements
C = (D^k-1 D^k-2 ... D^0 H B)

Figure 2: Erasure Coding (k,n) in HDFS: (i) each symbol is part of a separate HDFS block, (ii) k symbols represent a message, n symbols represent a codeword (initial message blocks + parity blocks), (ii) In Butterfly, each symbol is divided in $\alpha = 2^{k-1}$ elements.

ond step assumes fetching the data to the node where the task is running; (iii) in the third step the encoding/decoding computation is performed, and finally (iv) the newly created data is committed back to HDFS.

The first step, locating the data necessary for encoding/decoding process, is identical to the initial Facebook implementation: position of the symbol being built is used to calculate the HDFS file offset of the entire *k-symbol message/codeword* (data necessary for building the symbol). Calculating the offsets is possible because the size of the data being repaired equals the size of an entire HDFS block. In case the symbol being repaired is smaller than the HDFS block size, we rebuild all the symbols contained in that HDFS block. Therefore, we always fetch k consecutive HDFS blocks – *k-symbol message*. The location of the parity symbols/blocks during the decoding is determined using similar approach.

The second step, data fetching, is performed asynchronously and in parallel, from multiple datanodes. The size of the fetched data is directly related to the Butterfly message size, i.e. set of butterfly symbols spread across different datanodes. We allow the size of a Butterfly symbol to be a configurable parameter. We set the symbol element size to $\ell = symbol\,size/2^{k-1}$. The advantage of tunable symbol size is twofold: (i) improved data locality: size of the data chunks used in computation can be tuned to fit in cache; (ii) computation - communication overlap: "rightszing" the data chunk allows communication to be completely overlapped by computation.

The third step implements Butterfly encoding and decoding algorithms. While Section 3.2 presents formal definition of Butterfly, in Figure 3 we describe an example of Butterfly encoding/decoding schemes in HDFS. Figure 3 is intended to clarify the encoding/decoding process in HDFS-Butterfly through a simple example, and encoding/decoding of specific components might be somewhat different. Our encoding/decoding implementation is completely written in Java. While moving computation to a JNI module would significantly increase the level of applicable optimizations (including vectorization), these benefits would be shadowed by the cost of data movements between Java and JNI modules.

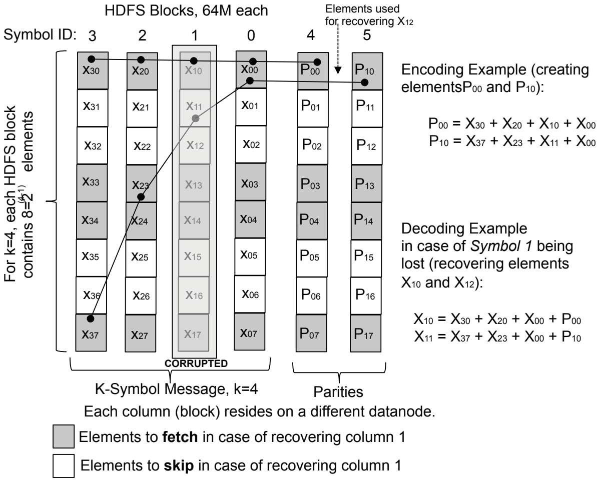

HDFS Blocks, 64M each
Elements used for recovering X12
Symbol ID: 3 2 1 0 4 5
For k=4, each HDFS block contains 8=2^(4-1) elements
X30 X31 X32 X33 X34 X35 X36 X37
X20 X21 X22 X23 X24 X25 X26 X27
X10 X11 X12 X13 X14 X15 X16 X17
X00 X01 X02 X03 X04 X05 X06 X07
P00 P01 P02 P03 P04 P05 P06 P07
P10 P11 P12 P13 P14 P15 P16 P17
CORRUPTED
Encoding Example (creating elementsP00 and P10):
P00 = X30 + X20 + X10 + X00
P10 = X37 + X23 + X11 + X00
Decoding Example in case of Symbol 1 being lost (recovering elements X10 and X12):
X10 = X30 + X20 + X00 + P00
X11 = X37 + X23 + X00 + P10
K-Symbol Message, k=4
Parities
Each column (block) resides on a different datanode.
Elements to fetch in case of recovering column 1
Elements to skip in case of recovering column 1

Figure 3: Regenerating column 1 for HDFS-Butterfly with $k = 4$. The line represents the elements that are xored to regenerate the first two symbols. The dark gray elements are the total required ones. White elements are skipped during communication.

Our Java computation is aggressively optimized through manual loop reordering and unrolling but we saw little benefits from these optimizations. By instrumenting the computational code, we found that the time spent in memory management significantly outweighs the benefits computational optimizations. We also parallelize computational loops in the OpenMP fashion. The degree of parallelization is a configurable parameter.

During the fourth step, the encoding/decoding task detects physical location of all the symbols (HDFS blocks) contained in the *k-symbol message*. The newly created symbol is placed on a physical node that does not contain any of the other message symbols, i.e. we avoid collocating two symbols from the same message on the same physical node. In this way we increase the system reliability in case of a singe node failure.

4.3 Communication Protocol

HDFS is a streaming filesystem and the client is designed to receive large amounts of contiguous data. If the data stream is broken, client assumes communication error and starts an expensive process of re-establishing connection with the datanode. However, the Butterfly repair process does not read remote data in a sequential manner. As explained in Figure 3, not all of the vector-symbol's elements are used during the decoding process (in Figure 3 only gray elements are used for reconstructing *Symbol 1*). The elements used for decoding can change, depending on the *symbol ID* being repaired. In our initial implementation, we allowed the datanode to *skip* reading unnecessary symbol components and send back only useful data (method `sendChunks()` in the HDFS datanode implementation). Surprisingly, this approach resulted in very low performance due to the interruptions in client–datanode data stream.

To avoid the HDFS overhead associated with streaming non-contiguous data, we implement *vector communication* between the client and the datanode: datanode packs all non-contiguous data chunks into a single contiguous buffer (gray components in Figure 3) and streams the entire buffer to the client side. On the client side, the received data is extracted and properly aligned following the requirements of the decoding process. Vectorization of communication introduced multi-fold performance improvement in HDFS.

To implement vectorized communication, we introduce *symbol ID* parameter to client→datanode request. *symbol ID* represents the position of the requested symbol in the butterfly message. In HDFS, the client discovers data blocks (symbols) needed for the decoding process. Therefore, the client passes *symbol ID* to the datanode, and the datanode uses this information to read from a local storage and send back only useful vector components.

4.4 Memory Management

Butterfly decoding process requires an amount of DRAM capable of storing an entire k symbol message. When the symbol size equals the size of the HDFS block (64 MB), the amount of DRAM equals $(k+2)$x64M. In addition, unpacking the data upon completing vector communication requires additional buffer space. The RaidNode daemon assigns recovery of multiple corrupted symbols to each map-reduce task for sequential processing. Tasks are required to allocate large amounts of memory when starting symbol recovery, and free the memory (garbage collect) upon decoding completion. Frequent and not properly scheduled garbage collection in JVM brings significant performance degradation.

To reduce the garbage collection overhead, we implemented a memory pool that is reused across multiple symbol decoders. The memory pool is allocated during the map-reduce task setup and reused later by the computation and communication threads. Moving the memory management from JVM to the application level increases implementation complexity, but at the same time we measured overall performance benefits of up to 15%.

5 Butterfly Codes in Ceph

Starting from version 0.80 Firefly, Ceph supports erasure code data redundancy through a pluggable interface that allows the use of a variety of traditional erasure codes and locally repairable codes (LRC). Unlike HDFS, Ceph allows encoding objects *on-line* as they are inserted in the system. The incoming data stream is partitioned into small chunks, or *stripes*, typically around 4096 bytes. Each of these small chunks is again split into k parts and encoded using the erasure code of choice. The described stripe-based approach allows the encoding process to be performed simultaneously while the data is being streamed in the system. Acknowledgment confirming the completed write operation is sent to the user immediately upon the data uploading and encoding are completed. Figure 4 describes the encoding process.

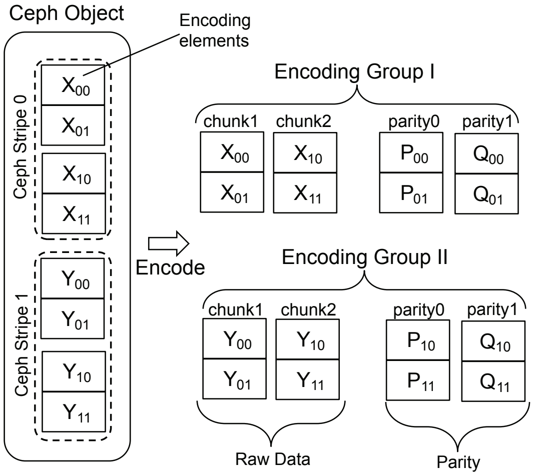


Figure 4: Ceph-Butterfly encoding example. For simplicity reasons $k = 2$. Object is divided in multiple stripes, and the Butterfly elements within each of the stripes are encoded.

The stripe size in Ceph determines the amount of memory required to buffer the data before the encoding process starts, i.e. larger stripe size requires more memory. Note that the entire encoding process takes place within a single server, therefore the memory is likely to be a scarce resource. In addition, having larger stripe sizes negatively affects the object write latency since one would get less benefits from pipelining the encoding process. Hence, from the performance point of view, it is desirable to use small stripe sizes. However, erasure code implementations benefit from operating on larger data chunks, because of being able to perform coarser computation tasks and read operations. In case of Butterfly codes, the performance is even more impacted by the stripe size, due to the large number of elements stored per column. As described in Figure 3, the Butterfly repair process requires accessing and communicating non--contiguous fragments of each code column. Small fragments incur high network and even higher HDD overhead. Due to internal HDD designs, reading random small fragments of data results in suboptimal disk performance. Because each Ceph stripe contains multiple Butterfly columns with $k \cdot 2^{k-1}$ elements per column, using large stripes is of great importance for the Butterfly repair process. In Section 6 we evaluate the effects that Ceph stripe size has on Butterfly repair performance.

5.1 Plug-In Infrastructure

To separate the erasure code logic from that of the OSD, RADOS uses an erasure code plug-in infrastructure that allows dynamical use of external erasure code libraries. The plug-in infrastructure is designed for traditional and LRC codes. Third-party developers can provide independent erasure code implementations and use the plug-in infrastructure to achieve seamless integration with RADOS. We summarize the three main plug-in functions:

– *encode()*: Given a data stripe, it returns a list of *n* encoded redundant chunks, each to be stored in a different node in the placement group. This function is not only used to generate parity chunks, but it is also responsible to stripe data across *n* nodes.

– *minimum_to_decode()*: Given a list of available chunks, it returns the IDs of the chunks required to decode the original data stripe. During the decoding process, Ceph will fetch the chunks associated with these IDs.

– *decode()*: Given a list of redundant chunks (corresponding to the IDs returned by *minimum to decode*) it decodes and returns the original data stripe.

Because it was intended for traditional and LRC codes, the Ceph erasure coding plug-in infrastructure does not differentiate between repairing missing redundant chunks, and decoding the original data. When RADOS needs to repair a missing chunk, it uses the *decode* function to decode all the *k* original message symbols, and then uses the *encode* function to only regenerate the missing ones. The described model does not allow recovering a missing chunk by only partially downloading other chunks, a feature that is indispensable for efficient MSR implementation.

Efficient integration of Butterfly code with Ceph requires re-defining the plug-in interface. The interface we propose is a generalization of the existing one, and is usable across all types of RGCs as well as existing LRCs and traditional codes. Compared to the previous interface, the new plug-in provides the following two extra functions:

– *minimum_to_repair()*: Given the ID of a missing chunk, it returns a list of IDs of the redundant blocks required to repair the missing one. Additionally, for each of the required IDs, it specifies an additional list of the subparts that need to be downloaded (a list of offsets and lengths). Ceph will download all the returns subparts from the corresponding nodes.

– *repair()*: Given the ID of a missing chunk, and the list of chunk subparts returned by *minimum to repair*, the function reconstructs the missing chunks.

In order to implement the new plug-in infrastructure, parts of the OSD implementation had to be changed. These changes in Ceph do not allow to support both systems simultaneously. For back-compatibility with legacy erasure code plug-ins, we implemented a proxy plug-in that dynamically links with existing plug-ins. In practice, if the new plug-in system does not find a requested plug-in library, the legacy proxy plug-in is loaded.

5.2 Butterfly Implementation

Matching the previous plug-in interface, Butterfly is implemented as an external C library and compiled as a new-style RADOS erasure code plug-in. The level of algorithmic and implementation optimizations included in Ceph-Butterfly is significantly higher than HDFS--Butterfly, due to HDFS's dependency on Java. Our implementation of Butterfly in Ceph follows the recursive description provided in Section 3. Compared to HDFS, the recursive approach simplifies implementation. The recursive approach also achieves better data locality, which provides better encoding throughput.

6 Results

In this section, we evaluate the repair performance of our two Butterfly implementations, HDFS-Butterfly and Ceph-Butterfly.

6.1 Experimental Setup

To evaluate our Butterfly implementations we use a cluster of 12 Dell R720 servers, each with one HDD dedicated to the OS, and seven 4TB HDDs dedicated to the distributed storage service. This makes a total cluster capacity of 336TB. The cluster is interconnected via 56 Gbps Infiniband network using IPoIB. High--performance network ensures that the communication bandwidth per server exceeds the aggregated disk bandwidth. In addition to 12 storage nodes, we use one node to act as a metadata server. In HDFS a single DataNode daemon per server manages the seven drives and an additional NameNode daemon runs on the metadata server. In Ceph each server runs one OSD daemon per drive and the MON daemon runs separately on the metadata server.

For the erasure code we consider two different configurations: $k = 5$ and $k = 7$. Since Butterfly codes add two parities, these parameters give us a storage overhead of 1.4x and 1.3x respectively, with a number of elements per code column of 16 and 64 respectively. Having two different k values allows capturing the impact that the number of code columns (and hence the IO granularity) has on the repair performance. We compare the Butterfly code performance against the default Reed Solomon code implementations in HDFS and Ceph for the same k values. For both systems we evaluate the performance to repair single node failures. Upon crashing a single data node, the surviving 11 servers are involved in recreating and storing the lost data.

Our experiments comprise 2 stages: (i) Initially we store 20,000 objects of 64MB each in the storage system. Including redundant data, that accounts for a total of 1.8TB of total stored data. Due to the data redundancy overhead, on average each node stores a total 149.33GB for $k = 5$, and 137.14GB for $k = 7$. (ii) In the second stage we power-off a single storage server and let the 11 surviving servers repair the lost data. We log CPU utilization, IO activity and network transfers of each server during the recovery process.

6.2 Repair Throughput

In Figure 5 we show aggregate repair throughput across all 12 nodes, in MB/s. Figure 5(a) represents a comparison of Butterfly and Reed-Solomon on HDFS when $k = 5$ and $k = 7$. For HDFS we allow 12 reduce tasks per node (1 task per core), i.e. each node can work on repairing 12 blocks simultaneously. In Figure 5(a), it is observable that in all cases the repair throughput has a steep fall towards the end. In addition, RS($k = 7$) experiences very low repair throughput towards the end of the repair process. If the number of blocks to be repaired is lower than the number of available tasks, the repair system does not run at the full capacity, resulting in the steep decline of the aggregated repair throughput. Repair process for RS ($k = 7$) experiences load imbalance resulting in low throughput towards the end. Providing repair load-balancing in Hadoop is out of scope of this study. We focus on the sustainable repair throughput rather than the overall running time.

In HDFS, for $k = 5$, Butterfly reaches a repair throughput between 500 and 600MB/s, which is 1.6x higher than RS repair throughput. Although this is an encouraging result, Butterfly does not reach twice the performance of RS due to several reasons. The most important being the higher storage media contention in case of Butterfly. During the repair process Butterfly accesses small non-contiguous column elements (see gray components in Figrue 3). Larger number of relatively small IOs introduces more randomness in HDD behavior which in turn reduces the overall repair performance of Butterfly. Another performance degrading source is the data manipulation required by the vector-based communications we introduced (see Section 4.3). Although vector-based communication significantly improves the overall performance, certain overhead is present due to data packing and unpacking, i.e. high overhead memory manipulation operations in Java.

It is interesting to note that for $k = 7$, the difference in repair throughput between Butterfly and RS is ∼2x. With larger values of k communication bandwidth requirements increase due to large number of blocks required during the repair process. For $k = 7$ the benefits of reducing the network contention with Butterfly significantly outweigh possible drawbacks related to HDD contention and vector communication.

In Figure 5(b),(c) we depict the repair throughput for Ceph. In the case of 4MB stripe size, Figure 5(b), each stripe forms a Butterfly data matrix of 2^{k-1} rows and k columns. Consequently, the size of each data element is of 50KB and 9KB for $k = 5$ and $k = 7$ respectively. During the repair process, when non-contiguous elements are accessed, the small element size results in an inefficient HDD utilization and additional CPU operations due to element manipulation. This in turn leads to a degraded an inconsistent repair throughput as we can observe in Figure 5(b). Increasing the stripe size to 64MB results into having element sizes of 800KB, 143KB, large enough sizes to make a better utilization of the disk, and provide better repair throughput as we depict in Figure 5(c).

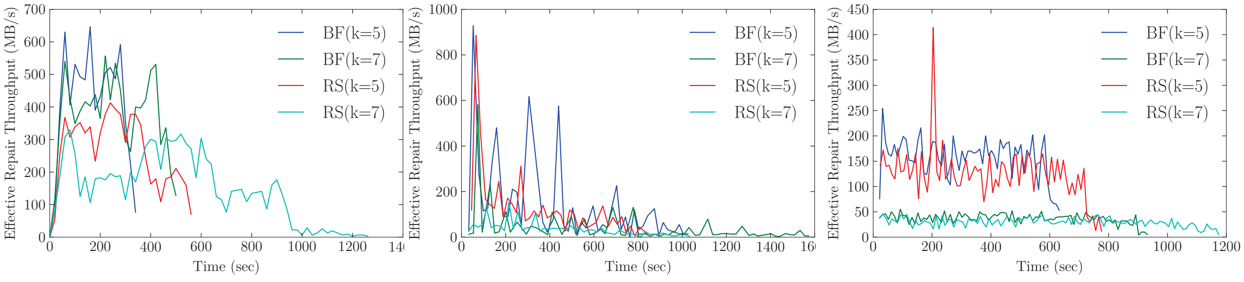
6.3 CPU Utilization

We measure CPU utilization of Butterfly/RS repair and evaluate the capability of each approach to possibly share in-node resources with other applications. CPU utilization understanding is of importance in distributed systems running in cloud-virtualized environments, or when the data repair processes share resources with other applications (e.g., map-reduce tasks). Figure 6 represents the CPU utilization of Butterfly and RS on a single node. The presented results are averaged across all nodes involved in computation.

For HDFS, in Figure 6(a) we observe that the CPU utilization for Butterfly exceeds RS CPU utilization by a factor of 3-4x, for both $k = 5$ and $k = 7$. Partially, this is due to the fact that RS spends more time waiting for network IO, because it requires higher communication costs compared to Butterfly. However, the number of total CPU cycles spent on computation in Butterfly is significantly higher than in RS. Compared to RS, Butterfly spends ∼2.1x and ∼1.7x more cycles, for $k = 5$ and $k = 7$ respectively. The observed CPU utilization is strongly tied to Java as the programming language of choice for HDFS. Butterfly implementation frequently requires non-contiguous data accesses and vector-based communication. Java does not have slice access to buffer arrays, requiring extra memory copies for packing and unpacking non-contiguous data.

Figure 6(b),(c) represents the CPU utilization for Ceph-Butterfly and RS, when the Ceph stripe size is 4MB and 64MB. For Ceph stripe size of 4MB and $k = 5$, the Butterfly repair process operates on large number of elements that are only ∼50K in size. The fine granularity computation, together with frequent and fine granularity communication, causes erratic and unpredictable CPU utilization, presented in Figure 6(b). Similar observation applies for $k = 7$. Note that the CPU utilization for RS is somewhat lower compared to Butterfly, but still unstable and with high oscillations. The RS repair process operates on somewhat coarser data chunks, but the software overhead (memory management, function calls, cache misses, etc.) is still significant.

With the Ceph stripe size of 64M, Figure 6(c), the Butterfly element size as well as the I/O size increases sig-

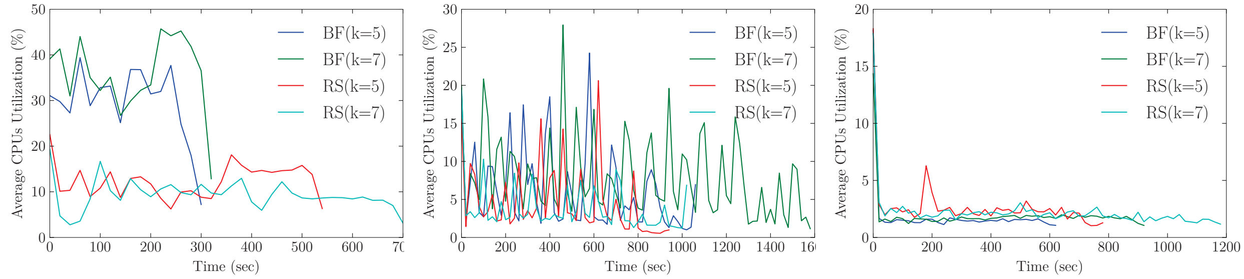


(a) HDFS.

(b) Ceph with 4MB stripes.

(c) Ceph with 64MB stripes.

Figure 5: Repair throughput aggregated across all nodes involved in the repair process. Each system configuration we run with RS and Butterfly, with $k = 5$ and $k = 7$.



(a) HDFS.

(b) Ceph with 4MB stripes.

(c) Ceph with 64MB stripes.

Figure 6: Average CPU utilization per server. Each system configuration we run with RS and Butterfly, with $k = 5$ and $k = 7$. The graphs represent the average utilization across all 12 nodes involved in the repair process.

nificantly, resulting in lower and more predictable CPU utilization. Although Butterfly algorithm requires more operations compared to RS, our cache-aware implementation, carefully optimized with vector instructions in C/C++, achieves CPU utilization comparable to that of RS. Furthermore, both Butterfly and RS achieve ∼2-3% CPU utilization across all observed k values.

The presented results show that MSR codes over GF(2) achieve low CPU usage and are a good candidate for running in multi-user environments. However, achieving efficient CPU utilization requires a programming language that allows appropriate set of optimizations, and relatively coarse data chunks. In an *on-line* encoding system, such as Ceph, the size of the stripe size is of extreme importance for achieving efficient CPU usage and good repair performance.

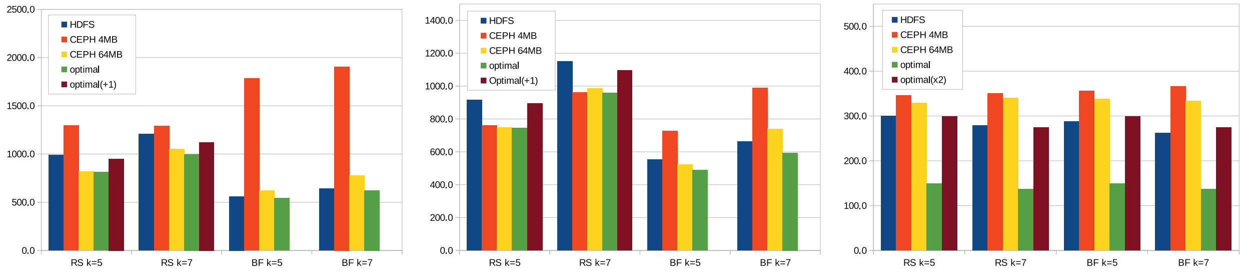
6.4 Network Traffic

In all systems used in this study we monitor network and storage traffic, and compare the observed results to the theoretical expectations. Figure 7 presents the results.

Figure 7(a) depicts the network traffic comparison. The *optimal* bars represent the lower bound on the amount of traffic. The *optimal + 1* bars represent the minimum increased by the size of a single HDFS block (we use 64MB block size). The original implementation of Reed-Solomon in Facebook - HDFS [2] unnecessarily moves an extra HDFS block to the designated repair node, causing somewhat higher network utilization. *optimal + 1* matches the amount of data pushed through the network in case of Reed-Solomon on HDFS.

We can observe in Figure 7(a) that HDFS-Butterfly implementation is very close to the theoretical minimum. The small difference between Butterfly and the *optimal* value is due to the impact of metadata size. Similarly, Reed-Solomon on HDFS is very close to *optimal + 1* with the additional metadata transfer overhead. In case of Ceph, the network traffic overhead is significantly higher. For Ceph-4MB, the large overhead comes from the very small chunks being transferred between the nodes and the per-message overhead introduced by the system. The communication overhead reduces for larger stripe sizes, i.e. Ceph-64MB. However, even with Ceph-64MB the communication overhead increases with k, again due to reduced message size and larger per-message overhead. Small message sizes in Ceph come as a consequence of the *on-line* encoding approach that significantly reduces the size of encoded messages, and hence the sizes of the symbol elements.

The results presented in Figure 7(a) reveal that if care-



(a) Overall Network Traffic in GB

(b) Overall Disk Reads in GB

(c) Overall Disk Writes in GB

Figure 7: The aggregate amount (across all 11 nodes included in the repair process) of network traffic and IOs during the repair process. We observe RS and Butterfly with $k = 5$ and $k = 7$.

fully implemented, MSR codes can reduce the repair network traffic by a factor of 2x compared to traditional erasure codes. Higher amount of network traffic in case of Ceph suggests that a specific system designs that avoid fine-grain communication is necessary.

6.5 Storage Traffic

Figures 7(b),(c) represent the observed HDD read/write traffic, as well as the theoretically optimal values predicted from the code design. We record the total amount of traffic across all HDDs in the system. HDFS-Butterfly achieves nearly optimal amount of read traffic, with the addition of metadata. HDFS-RS read traffic is also very close to *optimal + 1*, again the extra overhead comes from the HDFS metadata.

The results become somewhat more interesting with Ceph. It is noticeable that the difference between Ceph--Butterfly and *optimal* increases as we move from $k = 5$ to $k = 7$. Due to the small read I/O sizes with Ceph--Butterfly, reads suffer two drawbacks: (i) misaligned reads may cause larger data transfer (smallest HDD I/O is 4K and it needs to be 4K aligned), and (ii) read-ahead mechanism (128K by default) increases the amount of data transferred from an HDD. While read-ahead can be disabled, the entire system would suffer when reading sequential data, which is likely the most frequent scenario. The mentioned two drawbacks increase the influence on performance when the read size reduces, which is the case when we move from $k = 5$ to $k = 7$. With large enough Ceph stripes, the read I/O size increases in size, and the read overhead converges to zero. Example of large read I/Os is Ceph-RS, where the read overhead becomes negligible, Figure 7(b). In case of Butterfly, achieving large read I/Os requires impractically large Ceph stripe sizes.

For both systems and for both erasure code schemes, the overall amount of writes exceeds the *optimal* (lost data) amount by a factor of $\sim$2, as presented in Figure 7(c). Ceph allows updates of stored data, and for maintaining consistency in case of a failure, Ceph relies on journaling. In our experiments the journal for each OSD was co-located with the data, sharing the same HDD. In Ceph, all data being written have to pass through the journal and as a consequence the write traffic is doubled. Furthermore, the amount of data written in Ceph exceeds 2 x *optimal* because of data balancing. By examining the Ceph logs, we found that during the repair process many OSDs become unresponsive for certain amount of time. When that happens the recovered data is redirected to available OSDs, and load-balancing is performed when the non-responsive OSDs come back on-line. Note that the load-balancing can be performed among OSDs on the same server, therefore not affecting the network traffic significantly. Also, reads are not affected by load balancing since the data being moved around is “hot” and in large part cached in the local filesystem. Tracking down the exact reason for having OSDs temporarily unavailable is outside of scope of this study.

In case of HDFS, there is an intermediate local file where the recovered block is being written before committed back to the filesystem. This was the initial design in HDFS-RS, and our HDFS-Butterfly currently uses the same design. We will remove the extra write in the future. The intermediate file is not always entirely synced to HDD before the recovered data is further destaged to HDFS, resulting in the overall write traffic being sometimes lower than 2 x *optimal*.

7 Related Work

Traditional erasure codes, such as Reed-Solomon, have been implemented and tested in a number of large--scale distributed storage systems. Compared to replication, Reed-Solomon emerged as a good option for cost--effective data storage and good data durability. Most widely used open source distributed systems HDFS and Ceph implement Reed-Solomon variants [1, 5]. In addition, traditional erasure codes have been used in numerous other studies and production systems, including storage systems in Google and Facebook [3, 6, 15, 16, 18, 22]. Compared to the MSR code used in this study, the traditional erasure codes initiate up to 2x more network and storage traffic during the repair process.

When it comes to practical usage of advanced erasure coding schemes, previous work have mostly been focused on LRC-based implementations in distributed storage systems. Sathiamoorthy et al. [28] introduce Xorbas, an LRC-based erasure coding that they also implement in HDFS. The study presents significant performance improvements in terms of both, disk and network traffic during the repair process. Xorbas introduce 14% storage overhead compared to RS.

Huang et al. [20] implement and demonstrate the benefits of LRC-based erasure coding in Windows Azure Storage. Khan et al. [21] designed a variation of Reed-Solomon codes that allow to construct MDS array codes with symbol locality, optimized to improve degraded read performance. In case of Butterfly codes, the degraded reads performance would be equivalent to RAID 5 degraded reads, and not as efficient as the work presented in by Khan. Xia et al. [35] present an interesting approach, where they combine two different erasure coding techniques from the same family, depending on the workload. They use the fast code for high recovery performance, and compact code for low storage overhead. They focus on two erasure coding families, product codes and LRC. All of the mentioned studies focus on the LRC erasure codes that cannot achieve the minimum repair traffic described by MSRs.

Rashmi et al. [25] implemented a MSR code construction with a storage overhead above 2×. In the regime below 2×, Hu et al. [11] presented functional MSR code, capable of achieving significant performance improvement over the traditional erasure codes when it comes to repair throughput. However, upon the first data loss and repair process, functional MSRs do not hold systematic (raw) data in the system any more. Consequently, the cost of reading systematic data increases significantly as the system ages.

In recent years, several erasure code constructions have attained the theoretical minimum repair traffic [7, 10, 12, 24, 31]. Although similar in the amount of repair traffic, Butterfly codes are systematic MDS array codes with node elements over GF(2). This small field size allows relatively simple implementation and high computational performance.

8 Discussion, Future Work, Conclusions

In this study we captured the performance of MSR codes in real-world distributed systems. Our study is based on Butterfly codes, a novel MSR code which we implemented in two widely used distributed storage systems, Ceph and HDFS. Our study aims at providing answers to important questions related to MSR codes: (i) can the theoretical reduction in repair traffic translate to an actual performance improvement, and (ii) in what way the system design affects the MSR code repair performance. Our analysis shows that MSR codes are capable reducing network traffic and read I/O access during repairs. For example, Butterfly codes in HDFS achieves almost optimal network and storage traffic. However, the overall encoding/decoding performance in terms of latency and storage utilization heavily depends on the system design, as well as the ability of the system to efficiently manage local resources, such as memory allocation/deallocation/movement. Java-based HDFS experiences significant CPU overhead mainly due to non-transparent memory management in Java.

The encoding approach is one of the most important decisions the system architect faces when designing a distributed erasure coding system. The initial decision of using real-time or batch-based encoding strongly impacts the overall system design and performance. The real-time approach achieves efficient storage utilization, but suffers high storage access overhead due to excessive data fragmentation. We show that in Ceph, for stripes of 4MB the repair network overhead exceeds many times the expected one, while the storage access overhead goes up to 60% higher than optimal (depending on code parameters). The situation improves with larger stripe sizes but the communication and storage overhead remains. Batch-based data encoding (implemented in HDFS) achieves better performance, but reduces storage efficiency due to the required intermediate persistent buffer where input data is stored before being encoded.

To address the design issues, we suggest a system with on-line data encoding with large stripes, able to use local non-volatile memory (NVM) to accumulate enough data before encoding it. The non-volatile device has to be low-latency and high-endurance which are important attributes of future NVM devices, some of which have already been prototyped. Part of our on-going effort is to incorporate this non-volatile and low-latency devices into a distributed coding system.

When it comes to the features required to efficiently implement MSR codes in distributed storage systems, our results indicate that communication vectorization becomes necessary approach due to the non-contiguous data access pattern. The interface between the system and the MSR codes requires novel designs supporting the specific requirements of these codes. In case of Ceph we showed the necessity for chaining the plug-in API, and we proposed a new model that is suitable for MSR codes.

While the overall performance of MSR codes in distributed storage systems depends on many factors, we have shown that with careful design and implementation, MSR-based repairs can meet theoretical expectations and outperform traditional codes by up to a factor of 2x.

References

[1] Ceph Erasure. http://ceph.com/docs/master/rados/operations/erasure-code/. Accessed: Apr 2015.

[2] Facebook's hadoop 20. https://github.com/facebookarchive/hadoop-20.git. Accessed: Jan 2015.

[3] Google Colossus FS. http://static.googleusercontent.com/media/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf. Accessed: Apr 2015.

[4] HDFS Federation. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html. Accessed: Jan 2015.

[5] HDFS RAID. http://wiki.apache.org/hadoop/HDFS-RAID. Accessed: Apr 2015.

[6] M. Abd-El-Malek, W. V. Courtright, II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile Cluster-based Storage. In *Conference on USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[7] V. R. Cadambe, C. Huang, and J. Li. Permutation code: Optimal Exact-Repair of a Single Failed Node in MDS Code Based Distributed Storage Systems. In *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*, 2011.

[8] V. R. Cadambe, S. Jafar, H. Maleki, K. Ramchandran, C. Suh, et al. Asymptotic Interference Alignment for Optimal Repair of MDS Codes in Distributed Storage. *Information Theory, IEEE Transactions on*, 2013.

[9] V. R. Cadambe, S. A. Jafar, and H. Maleki. Distributed Data Storage with Minimum Storage Regenerating Codes-Exact and Functional Repair are Asymptotically Equally Efficient. *arXiv preprint arXiv:1004.4299*, 2010.

[10] V. R. Cadambe, S. A. Jafar, and H. Maleki. Minimum Repair Bandwidth for Exact Regeneration in Distributed Storage. In *Wireless Network Coding Conference*, 2010.

[11] H. C. Chen, Y. Hu, P. P. Lee, and Y. Tang. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. *IEEE Transactions on Computers*, 2014.

[12] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *Information Theory, IEEE Transactions on*, 2010.

[13] A. G. Dimakis, P. B. Godfrey, M. J. Wainwright, and K. Ramchandran. The Benefits of Network Coding for Peer-To-Peer Storage Systems. In *Third Workshop on Network Coding, Theory, and Applications*, 2007.

[14] E. En Gad, R. Mateescu, F. Blagojevic, C. Guyot, and Z. Bandic. Repair-Optimal MDS Array Codes Over GF (2). In *Information Theory Proceedings (ISIT), IEEE International Symposium on*, 2013.

[15] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. Diskreduce: Replication as a Prelude to Erasure Coding in Data-Intensive Scalable Computing. In *Proceedings of the International Conference for High Performance Computing Networking, Storage and Analysis (SC)*, 2011.

[16] D. Ford, F. Labelle, F. 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*, 2010.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP)*, 2003.

[18] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation (NSDI)*, 2005.

[19] C. Huang, M. Chen, and J. Li. Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems. In *Network Computing and Applications (NCA), 6th IEEE International Symposium on*, 2007.

[20] 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 *USENIX Annual Technical Conference (ATC)*, 2012.

[21] 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 *FAST*, page 20, 2012.

[22] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. *SIGPLAN Not.*, 35(11):190–201, Nov. 2000.

[23] S.-J. Lin, W.-H. Chung, Y. S. Han, and T. Y. Al-Naffouri. A Unified Form of Exact-MSR Codes via Product-Matrix Frameworks. *Information Theory, IEEE Transactions on*, 61(2):873–886, 2015.

[24] D. S. Papailiopoulos, A. G. Dimakis, and V. R. Cadambe. Repair Optimal Erasure Codes Through Hadamard Designs. *Information Theory, IEEE Transactions on*, 2013.

[25] 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 *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 81–94. USENIX Association, 2015.

[26] 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. *Information Theory, IEEE Transactions on*, 57(8):5227–5239, 2011.

[27] B. Sasidharan, G. K. Agarwal, and P. V. Kumar. A High-Rate MSR Code With Polynomial Sub-Packetization Level. *arXiv preprint arXiv:1501.06662*, 2015.

[28] 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 *Proceedings of the VLDB Endowment*, volume 6, pages 325–336. VLDB Endowment, 2013.

[29] N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchandran. Interference Alignment in Regenerating Codes for Distributed Storage: Necessity and Code Constructions. *Information Theory, IEEE Transactions on*, 58(4):2134–2158, 2012.

[30] C. Suh and K. Ramchandran. Exact-repair MDS Code Construction Using Interference Alignment. *Information Theory, IEEE Transactions on*, 57(3):1425–1442, 2011.

[31] I. Tamo, Z. Wang, and J. Bruck. Zigzag Codes: MDS Array Codes with Optimal Rebuilding. *Information Theory, IEEE Transactions on*, 59(3):1597–1616, 2013.

[32] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. 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.

[33] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122. ACM, 2006.

[34] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. Rados: A Scalable, Reliable Storage Service for Petabyte-Scale Storage Clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing'07*, pages 35–44. ACM, 2007.

[35] M. Xia, M. Saxena, M. Blaum, and D. A. Pease. A Tale of Two Erasure Codes in HDFS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 213–226. USENIX Association, 2015.

The Devil is in the Details: Implementing Flash Page Reuse with WOM Codes

Fabio Margaglia[†], Gala Yadgar[*], Eitan Yaakobi[*], Yue Li[§], Assaf Schuster[*] and André Brinkmann[†]
[†]*Johannes Gutenberg-Universität Mainz*
[*]*Computer Science Department, Technion*
[§]*California Institute of Technology*

Abstract

Flash memory is prevalent in modern servers and devices. Coupled with the scaling down of flash technology, the popularity of flash memory motivates the search for methods to increase flash reliability and lifetime. Erasures are the dominant cause of flash cell wear, but reducing them is challenging because flash is a *write-once* medium— memory cells must be erased prior to writing.

An approach that has recently received considerable attention relies on *write-once memory (WOM)* codes, designed to accommodate additional writes on write-once media. However, the techniques proposed for reusing flash pages with WOM codes are limited in their scope. Many focus on the coding theory alone, while others suggest FTL designs that are application specific, or not applicable due to their complexity, overheads, or specific constraints of MLC flash.

This work is the first that addresses all aspects of page reuse within an end-to-end implementation of a general-purpose FTL on MLC flash. We use our hardware implementation to directly measure the short and long-term effects of page reuse on SSD durability, I/O performance and energy consumption, and show that FTL design must explicitly take them into account.

1 Introduction

Flash memories have special characteristics that make them especially useful for solid-state drives (SSD). Their short read and write latencies and increasing throughput provide a great performance improvement compared to traditional hard disk based drives. However, once a flash cell is written upon, changing its value from 1 to 0, it must be erased before it can be rewritten. In addition to the latency they incur, these erasures wear the cells, degrading their reliability. Thus, flash cells have a limited lifetime, measured as the number of erasures a block can endure.

Multi-level flash cells (MLC), which support four voltage levels, increase available capacity but have especially short lifetimes, as low as several thousands of erasures. Many methods for reducing block erasures have been suggested for incorporation in the flash translation layer (FTL), the SSD management firmware. These include minimizing user and internal write traffic [14, 19, 20, 28, 37, 38, 42, 46, 55] and distributing erasure costs evenly across the drive's blocks [7, 22, 25, 27].

A promising technique for reducing block erasures is to use write-once memory (WOM) codes. WOM codes alter the logical data before it is physically written, thus allowing the reuse of cells for multiple writes. They ensure that, on every consecutive write, ones may be overwritten with zeros, but not vice versa. Reusing flash cells with this technique might make it possible to increase the amount of data written to the block before it must be erased.

Flash page reuse is appealing because it is orthogonal to other FTL optimizations. Indeed, the design of WOM codes and systems that use them has received much attention in recent years. While the coding theory community focuses on optimizing these codes to reduce their redundancy and complexity [9, 10, 13, 17, 44, 49], the storage community focuses on SSD designs that can offset these overheads and be applied to real systems [24, 36, 53].

However, the application of WOM codes to state-of-the-art flash chips is not straightforward. MLC chips impose additional constraints on modifying their voltage levels. Previous studies that examined page reuse on real hardware identified some limitations on reprogramming MLC flash, and thus resort to page reuse only on SLC flash [24], outside an SSD framework [18], or within a limited special-purpose FTL [31].

Thus, previous SSD designs that utilize WOM codes have not been implemented on real platforms, and their benefits were analyzed by simulation alone, raising the concern that they could not be achieved in real world storage systems. In particular, hardware aspects such as possible increase in cell wear and energy consumption due to the additional writes and higher resulting voltage levels have not been examined before, but may have dramatic implications on the applicability of this approach.

In this study, we present the first end-to-end evaluation and analysis of flash page reuse with WOM codes. The first part of our analysis consists of a low-level evaluation of four state-of-the-art MLC flash chips. We examine the possibility of several reprogramming schemes for MLC flash and their short and long-term effects on the chip's durability, as well as the difference in energy consumption compared to that of traditional use.

The second part of our analysis consists of a system-level FTL evaluation on the OpenSSD board [4]. Our FTL design takes into account the limitations identified in the low-level analysis and could thus be implemented and evaluated on real hardware. We measure erasures and I/O response time and compare them to those observed in previous studies.

The discrepancy between our results and previous ones emphasizes why understanding low-level constraints on page reuse is crucial for high-level designs and their objectives. We present the lessons learned from our analysis in the form of guidelines to be taken into account in future designs, implementations, and optimizations.

The rest of this paper is organized as follows. Section 2 describes the basic concepts that determine to what extent it is possible to benefit from flash page reuse. We identify the limitations on page reuse in MLC flash in Section 3, with the implications on FTL design in Section 4. We describe our experimental setup and FTL implementation in Section 5, and present our evaluation in Section 6. We survey related work in Section 7, and conclude in Section 8.

2 Preliminaries

In this section, we introduce the basic concepts that determine the potential benefit from flash page reuse: WOM codes, MLC flash, and SSD design.

2.1 Write-Once Memory Codes

Write-once memory (WOM) codes were first introduced in 1982 by Rivest and Shamir, for recording information multiple times on a write-once storage medium [40]. They give a simple WOM code example, presented in Table 1. This code enables the recording of two bits of information in three cells twice, ensuring that in both writes the cells change their value only from 1 to 0. For example, if the first message to be stored is 00, then 110 is written, programming only the last cell. If the second message is 10, then 010 is written, programming the first cell as well. Note that without special encoding, 00 cannot be overwritten by 10 without prior erasure. If the first and second messages are identical, then the cells do not change their value between the first and second writes. Thus, before performing a second write, the cell values must be *read* in order to determine the correct encoding.

Data bits	1st write	2nd write
11	111	000
01	011	100
10	101	010
00	110	001

Table 1: WOM code example

WOM code instances, or *constructions*, differ in the number of achievable writes and in the manner in which each successive write is encoded. The applicability of a WOM code construction to storage depends on three characteristics: (a) the *capacity overhead* —the number of extra cells required to encode the original message, (b) the encoding and decoding *efficiency*, and (c) the *success rate*—the probability of producing an encoded output that can be used for overwriting the chosen cells. Any two of these characteristics can be optimized at the cost of compromising the third.

Consider, for example, the code depicted in Table 1, where encoding and decoding are done by a simple table lookup, and therefore have complexity $O(1)$ and a success rate of 100%. However, this code incurs a capacity overhead of 50% on each write. This means that (1) only $\frac{2}{3}$ of the overall physical capacity can be utilized for logical data, and (2) every read and write must access 50% more cells than what is required by the logical data size.

The theoretical lower bound on capacity overhead for two writes is 29% [40]. Codes that incur this minimal overhead (*capacity achieving*) are not suitable for real systems. They either have exponential and thus inapplicable complexity, or complexity of $n \log n$ (where n is the number of encoded bits) but a failure rate that approaches 1 [10, 56]. Thus, early proposals for rewriting flash pages using WOM codes that were based on capacity achieving codes were impractical. In addition, they required partially programming additional pages on each write, modifying the physical page size [8, 18, 23, 30, 36, 50], or compressing the logical data prior to encoding [24].

Two recently suggested WOM code families, Polar [9, 10] and LDPC [56], have the same complexities as the error correction codes they are derived from. For these complexities, different constructions incur different capacity overheads, and the failure rate decreases as the capacity overhead increases. Of particular interest are constructions in which the overhead of the first write is 0, i.e., one logical page is written on one physical page. The data encoded for the second write requires two full physical pages for one logical page. Such a construction is used in the design of ReusableSSD [53], where the second write is performed by programming pages containing invalid data on two different blocks in parallel.

2.2 Multi-Level Cell (MLC) Flash

A flash chip is built from floating-gate cells whose state depends on the number of electrons they retain. Writing is done by *programming* the cell, increasing the *threshold voltage (V_{th})* required to activate it. Cells are organized in blocks, which are the unit of erasure. Blocks are further divided into pages, which are the read and program units.

Single-level cells (SLC) support two voltage levels, mapped to either 1 (in the initial state) or 0. Thus, SLC flash is a classic write-once memory, where pages can be reused by programming some of their 1's to 0's. We refer to programming without prior erasure as *reprogramming*. Multi-level cells (MLC) support four voltage levels, mapped to 11 (in the initial state), 01, 00 or 10. This mapping, in which a single bit is flipped between successive

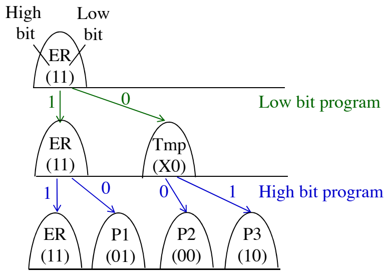


Figure 1: Normal programming order and states of MLC flash. ER is the initial (erased) state.

	A16	A27	B16	B29	C35
Feature size	16nm	27nm	16nm	29nm	35nm
Page size	16KB	8KB	16KB	4KB	8KB
Pages/block	256	256	512	256	128
Spare area (%)	10.15	7.81	11.42	5.47	3.12
Lifetime (T)	3K	5K	10K	10K	NA

Table 2: Evaluated flash chip characteristics. A, B and C represent different manufacturers. The C35 chip was examined in a previous study, and is included here for completeness.

states, minimizes bit errors if the cell's voltage level is disturbed. The least and most significant bits represented by the voltage levels of a multi-level cell are mapped to two separate pages, the *low page* and *high page*, respectively. These pages can be programmed and read independently. However, programming must be done in a certain order to ensure that all possible bit combinations can be read correctly. Triple-level cells (TLC) support eight voltage levels, and can thus store three bits. Their mapping schemes and programming constraints are similar to those of MLC flash. We focus our discussion on MLC flash, which is the most common technology in SSDs today.

Figure 1 depicts a normal programming order of the low and high bits in a multi-level cell. The cell's initial state is the erased (ER) state corresponding to 11. The low bit is programmed first: programming 1 leaves the cell in the erased state, while programming 0 raises its level and moves it to a temporary state. Programming the high bit changes the cell's state according to the state it was in after the low bit was programmed, as shown in the bottom part of the figure.[1] We discuss the implications of this mapping scheme on page reuse in the following section.

Bit errors occur when the state of the cell changes unintentionally, causing a bit value to flip. The reliability of a flash block is measured by its *bit error rate (BER)*—the average number of bit errors per page. The high voltage applied to flash cells during repeated program and erase operations gradually degrades their ability to retain the applied voltage level. This causes the BER to increase as the block approaches the end of its lifetime, which is measured in program/erase (P/E) cycles.

Bit errors in MLC flash are due mainly to *retention errors* and *program disturbance* [11]. Retention errors occur when the cell's voltage level gradually decreases below the boundaries of the state it was programmed to. Program disturbance occurs when a cell's state is altered during programming of cells in a neighboring page. In the following section, we discuss how program disturbance limits MLC page reuse, and evaluate the effects of reusing a block's pages on its BER.

Error correction codes (ECC) are used to correct some of the errors described above. The redundant bits of the ECC are stored in each page's *spare area*. The number of bit errors an ECC can correct increases with the number of redundant bits, chosen according to the expected BER at the end of a block's lifetime [56].

Write requests cannot update the data in the same place it is stored, because the pages must first be erased. Thus, writes are performed *out-of-place*: the previous data location is marked as invalid, and the data is written again on a clean page. The *flash translation layer (FTL)* is the SSD firmware component responsible for mapping logical addresses to physical pages. We discuss relevant components of the FTL further in Section 4.

3 Flash Reliability

Flash chips do not support reprogramming via their standard interfaces. Thus, the implications of reprogramming on the cells' state transitions and durability cannot be derived from standard documentation, and require experimentation with specialized hardware. We performed a series of experiments with several state-of-the-art flash chips to evaluate the limitations on reprogramming MLC flash pages and the implications of reprogramming on the chip's lifetime, reliability, and energy consumption.

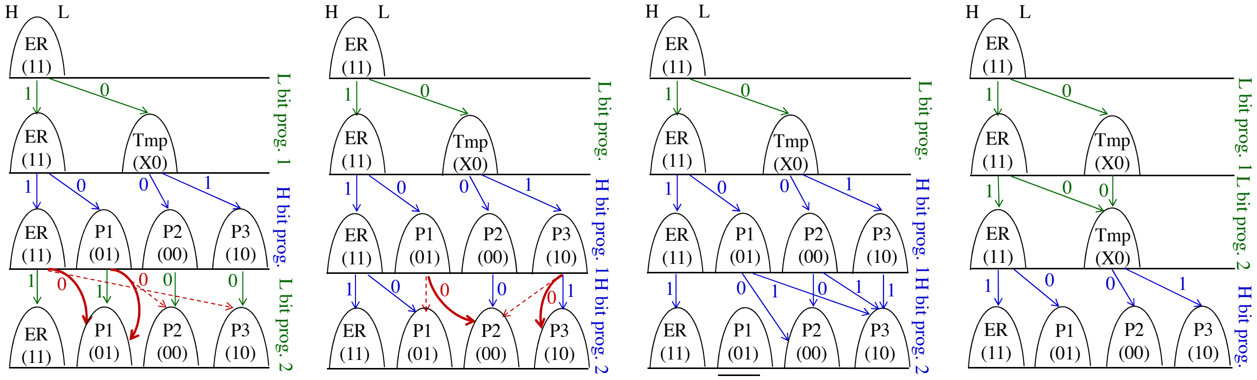
3.1 Flash Evaluation Setup

We used four NAND flash chips from two manufacturers and various feature sizes, detailed in Table 2. We also include in our discussion the observations from a previous study on a chip from a third manufacturer [31]. Thus, our analysis covers three out of four existing flash vendors.

Chip datasheets include the expected lifetime of the chip, which is usually the maximal number of P/E cycles that can be performed before the average BER reaches 10^{-3}. However, cycling the chips in a lab setup usually wears the cells faster than normal operation because they program and erase the same block continuously. Thus, the threshold BER is reached after fewer P/E cycles than expected. In our evaluation, we consider the lifetime (T) of the chips as the minimum of the expected number of cycles, and the number required to reach a BER of 10^{-3}.

Our experiments were conducted using the SigNASII commercial NAND flash tester [6]. The tester allows software control of the physically programmed flash blocks and pages within them. By disabling the ECC hardware we were able to examine the state of each cell, and to count the bit errors in each page.

[1] Partially programming the high bit in the temporary state is designed to reduce program disturbance.



(a) LHL reprogramming (b) LHH reprogramming (c) $\overline{\text{LHH}}$ reprogramming (d) LLH reprogramming

Figure 2: State transitions in the three reprogramming schemes. A thin arrow represents an attempted transition. A dashed arrow represents a failed transition, with a bold arrow representing the erroneous transition that takes place instead. Only LLH reprogramming achieves all the required transitions for page reuse without program disturbance.

Some manufacturers employ *scrambling* within their chip, where a random vector is added to the logical data before it is programmed. Scrambling achieves uniform distribution of the flash cell levels, thus reducing various disturbance effects. In order to control the exact data that is programmed on each page, we bypass the scrambling mechanism on the chips that employ it.

Our evaluation excludes retention errors, which occur when considerable time passes between programming and reading a page. Reprogramming might increase the probability of retention errors because it increases the cell's V_{th}. However, since it is intended primarily for hot data, we believe it will not cause additional retention errors.

3.2 Limitations on reprogramming

Flash cell reprogramming is strictly limited by the constraint that V_{th} can only increase, unless the block is erased. At the same time, WOM encoding ensures that reprogramming only attempts to change the value of each bit from 1 to 0. However, additional limitations are imposed by the scheme used for mapping voltage levels to bit values, and by the need to avoid additional program disturbance. Thus, page reuse must follow a *reprogramming scheme* which ensures that all reprogrammed cells reach their desired state. We use our evaluation setup to examine which state transitions are possible in practice. We first consider three reprogramming schemes in which a block has been fully programmed, and show why they are impractical. We then validate the applicability of reprogramming when only the low pages of the block have been programmed before.

Let us assume that the entire block's pages have been programmed before they are reused. Thus, the states of the cells are as depicted in the bottom row of Figure 1. In the *low-high-low (LHL)* reprogramming scheme, depicted in Figure 2(a), we attempt to program the low bit from this state. The thin arrows depict possible desired transitions in this scheme. Two such transitions are impossible, resulting in an undesired state (depicted by the bold arrow). In the *low-high-high (LHH)* reprogramming scheme, depicted in Figure 2(b), the high page is reprogrammed in a fully used block. Here, too, two state transitions fail.

A possible reason for the failed transitions in the LHL scheme is that the voltage applied by the command to program the low bit is not high enough to raise V_{th} from $P1$ to $P2$ and from ER to $P3$.[2] The transition from $P3$ to $P2$ in the LHH scheme is impossible, because it entails decreasing V_{th}. Another problem in the LHH scheme occurs in state $P1$ when we attempt to leave the already programmed high bit untouched. Due to an unknown disturbance, the cell transitions unintentionally to $P2$, corrupting the data on the corresponding low page.

Three of these problematic transitions can probably be made possible with proper manufacturer support—the transition from $P3$ to $P2$ in the LHH scheme would be possible with a different mapping of voltage levels to states, and the two transitions in the LHL scheme could succeed if a higher voltage was applied during reprogramming. While recent technology trends, such as one-shot programming and 3D V-NAND [21], eliminate some constraints on page programming, applying such architectural changes to existing MLC flash might amplify program disturbance and increase the BER. Thus, they require careful investigation and optimization.

An alternative to modifying the state mapping is modifying the WOM encoding, so that the requirement that 1's are only overwritten by 0's is replaced by the requirement that 0's are only overwritten by 1's. Figure 2(c) shows the resulting $\overline{\textit{low-high-high}}$ $(\overline{\textit{LHH}})$ reprogramming scheme. Its first drawback is that it corrupts the low pages, so a high page can be reused only if the data on the low page is either invalid, or copied elsewhere prior to reprogramming. Such reprogramming also corrupted the high pages adjacent to the reprogrammed one. Thus, this scheme allows safe reprogramming of only one out of two high pages.

[2] The transition from ER to $P3$ actually succeeded in the older, C35 chip [31]. All other problematic transitions discussed in this section failed in all the chips in Table 2.

The benefits from such a scheme are marginal, as these pages must also store the redundancy of the encoded data.

Interestingly, reprogramming the high bits in chips from manufacturer A returned an error code and did not change their state, regardless of the attempted transition. A possible explanation is that this manufacturer might block reprogramming of the high bit by some internal mechanism to prevent the corruption described above.

The problems with the LHL and LHH schemes motivated the introduction of the *low-low-high (LLH)* reprogramming scheme by Margaglia et al. [31]. Blocks in this scheme are programmed in two rounds. In the first round only the low pages are programmed. The second round takes place after most of the low pages have been invalidated. All the pages in the block are programmed in order, i.e., a low page is reprogrammed and then the corresponding high page is programmed for the first time, before moving on to the next pair of pages.

We validated the applicability of the LLH scheme on the chips of manufacturers A and B. Figure 2(d) depicts the corresponding state transitions of the cells. Since both programming and reprogramming of the low bit leave the cell in either the erased or temporary state, there are no limitations on the programming of the high page in the bottom row. This scheme works well in all the chips we examined. However, it has the obvious drawback of leaving half of the block's capacity unused in the first round. This leads to the first lesson from our low-level evaluation.

Lesson 1: *Page reuse in MLC flash is possible, but can utilize only half of the pages and only if some of its capacity has been reserved in advance. FTL designs must consider the implications of this reservation.*

3.3 Average V_{th} and BER

In analyzing the effects of reprogramming on a chip's durability, we distinguish between *short-term* effects on the BER due to modifications in the current P/E cycle, and *long-term* wear on the cell, which might increase the probability of errors in future cycles. With this distinction, we wish to identify a *safe* portion of the chip's lifetime, during which the resulting BER as well as the long term wear are kept at an acceptable level.

Reprogramming increases the probability that a cell's value is 0. Thus, the average V_{th} of reused pages is higher than that of pages that have only been programmed once. A higher V_{th} increases the probability of a bit error. The short-term effects of increased V_{th} include increased program disturbance and retention errors, which are a direct result of the current V_{th} of the cell and its neighboring cells. The long-term wear is due to the higher voltage applied during programming and erasure.

Our first set of experiments evaluated the short-term effects of increased V_{th} on a block's BER. In each chip, we performed T regular P/E cycles writing random data on one block, where T is the lifetime of the chip as detailed in Table 2. We repeated this processwith different distributions of 1 and 0. $P_{0.5}$, in which the probability of a bit to be 0 is 0.5, is our baseline. With P_{LLH} the probability of 0 was 0.75 and 0.5 in the low and high page, respectively. This corresponds to the expected probabilities after LLH reprogramming. We read the block's content and recorded the BER after every P/E cycle. We repeated each experiment on six blocks, and calculated the average.

Num. of P_{LLH} cycles	A16	A27	B16	B29
T (= entire lifetime)	32%	29%	20%	30.5%
$0.6 \times T$	8%	9%	8%	9%
$0.4 \times T$	6%	6.5%	6%	6.5%
$0.2 \times T$	2%	3%	3%	3.5%

Table 3: Expected reduction in lifetime due to increased V_{th}.

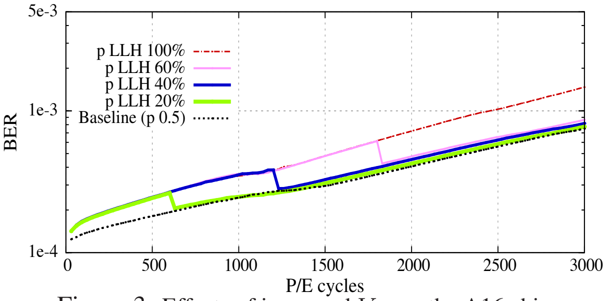


Figure 3: Effects of increased V_{th} on the A16 chip.

The implication of an increase in BER depends on whether it remains within the error correction capabilities of the ECC. A small increase in BER at the end of a block's lifetime might deem it unusable, while a large increase in a 'young' block has little practical effect. For a chip with lifetime T, let T' be the number of cycles required to reach a BER of 10^{-3} in this experiment. Then $T - T'$ is the *lifetime reduction* caused by increasing V_{th}. Our results, summarized in Table 3, were consistent in all the chips we examined.[3] Programming with P_{LLH}, which corresponds to a higher average V_{th}, shortened the chips' lifetime considerably, by 20–32%.

In the next set of experiments, we evaluated the long-term effects of V_{th}. Each experiment had two parts: we programmed the block with P_{LLH} in the first part, for a portion of its lifetime, and with $P_{0.5}$ in the second part, which consists of the remaining cycles. Thus, the BER in the second part represents the long-term effect of the biased programming in the first part. We varied the length of the first part between 20%, 40% and 60% of the block's lifetime. Figure 3 shows the BER of blocks in the A16 chip (the graphs for the different chips were similar), with the lifetime reduction of the rest of the chips in Table 3.

Our results show that the long-term effect of increasing V_{th} is modest, though nonnegligible—increasing V_{th} early in the block's lifetime shortened it by as much as 3.5%, 6.5% and 9%, with increased V_{th} during 20%, 40% and 60% of the block's lifetime, respectively.

[3] The complete set of graphs for all the experiments described in this section is available in our technical report [54].

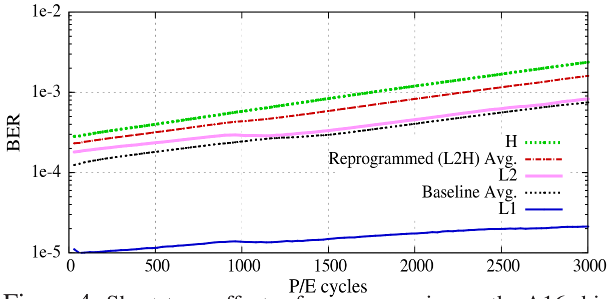


Figure 4: Short-term effects of reprogramming on the A16 chip.

Num. of LLH cycles	A16	A27	B16	B29
T (= entire lifetime)	38%	59.5%	99%	31%
$0.6 \times T$	8.5%	8%	7%	8.5%
$0.4 \times T$	5.2%	6%	5%	5.5%
$0.2 \times T$	1%	2.5%	3%	3%

Table 4: Expected reduction in lifetime due to reprogramming.

3.4 Reprogramming and BER

In the third set of experiments, we measured the effects of reprogramming by performing T LLH reprogramming cycles on blocks in each chip. Figure 4 shows the BER results for the A16 chip, and Table 4 summarizes the expected lifetime reduction for the remaining chips.

In all the chips, the BER in the first round of programming the low pages was extremely low, thanks to the lack of interference from the high pages. In the second round, however, the BER of all pages was higher than the baseline, and resulted in a reduction of lifetime greater than that caused by increasing V_{th}. We believe that a major cause of this difference are optimizations tailored for the regular LH programming order [39]. These optimizations are more common in recent chips, such as the B16 chip.

In the last set of experiments, we evaluated the long-term effects of reprogramming. Here, too, each experiment was composed of two parts: we programmed the block with LLH reprogramming in the first part, and with $P_{0.5}$ and regular programming in the second part. We varied the length of the first part between 20%, 40% and 60% of the block's lifetime. Figure 5 shows the BER results for the A16 chip, and Table 4 summarizes the expected lifetime reduction for the remaining chips.

We observe that the long-term effects of reprogramming are modest, and comparable to the long-term effects of increasing V_{th}. This supports our assumption that the additional short-term increase in BER observed in the previous set of experiments is not a result of the actual reprogramming process, but rather of the mismatch between the programming order the chips are optimized for and the LLH reprogramming scheme. This is especially evident in the B16 chip, in which the BER during the first part was high above the limit of 10^{-3}, but substantially smaller in the second part of the experiment.

Thus, schemes that reuse flash pages only at the beginning of the block's lifetime can increase its utilization without degrading its long-term reliability. Moreover, in all but the B16 chips, LLH reprogramming in the first 40% of the block's lifetime resulted in BER that was well within the error correction capabilities of the ECC. We rely on this observation in our FTL design in Section 4.

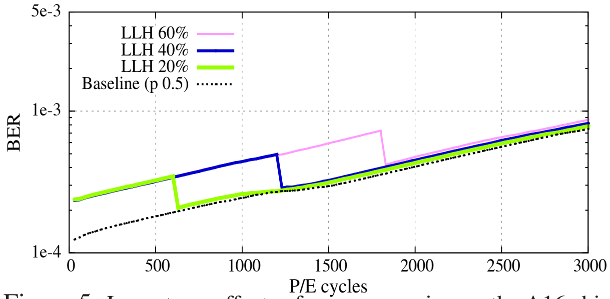


Figure 5: Long-term effects of reprogramming on the A16 chip.

Operation	Baseline (μJ)	LLH (μJ)
Erase	192.79	186.49
Read (L)	50.37	50.37
Read (H)	51.25	51.25
Program (L_1)	68.18	68.55
Reprogram (L_2)	NA	63.04
Program (H)	195.65	180.85
Average logical read	50.81	60.79
Average logical write	132.64	145.71

Table 5: Energy consumed by flash operations on chip A16.

We note, however, that the variance between the chips we examined is high, and that short and long-term effects do not depend only on the feature size. For example, the A16 chip is "better" than the A27 chip, but the B16 chip is "worse" than the B29 chip. This leads to the second lesson from our low-level evaluation.

Lesson 2: *The portion of the block's lifetime in which its pages can be reused safely depends on the characteristics of its chip. The FTL must take into account the long-term implications of reuse on the chips it is designed for.*

3.5 Energy consumption

Flash read, write and erase operations consume different amounts of energy, which also depend on whether the operation is performed on the high page or on the low one, and on its data pattern. We examined the effect of reprogramming on energy consumption by connecting an oscilloscope to the SigNAS tester. We calculated the energy consumed by each of the following operations on the A16 chip: an erasure of a block programmed with P_{LLH} and p=0.5, reading and writing a high and a low page, reprogramming a low page, and programming a high page on a partially-used block.

To account for the transfer overhead of WOM encoded data, our measurements of read, program and reprogram operations included the I/O transfer to/from the registers. Our results, averaged over three independent measurements, are summarized in Table 5. We also present the average energy consumption per read or write operation with baseline and with LLH reprogramming, taking into account the size of the programmed data, the reading of

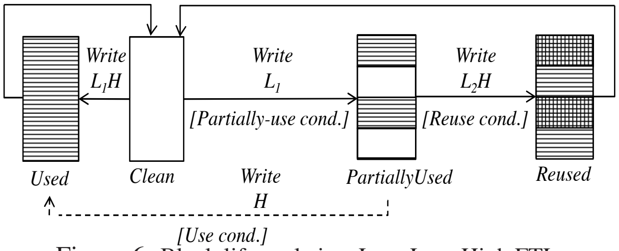


Figure 6: Block life cycle in a Low-Low-High FTL.

used pages for supplying the invalid data as input to the WOM encoder, and the number of pages that can be written before each erasure.

These results show that page reuse consumes more overall energy than the baseline. This is in contrast to previous studies showing possible energy savings. These studies assumed that the energy is proportional to the number of programmed *cells*, which is equivalent in a first and in a second write [18, 53]. However, our hardware evaluation shows that the number of reprogrammed *pages* is the dominant factor in energy consumption. While reprogramming a lower page consumes less energy than the average logical write in the baseline, the use of WOM encoding entails an extra read and page reprogram for each logical write. The low energy consumption of the saved erasures does not offset the additional energy consumed by those operations. We note, however, that when page reuse reduces the internal writes by the FTL, some energy savings may result. We examine that possibility further in Section 6, but can already draw the following lesson.

Lesson 3: *With WOM encoded data, the energy consumed by the additional flash operations is larger than that required by the saved erase operations. Energy savings are possible only if they reduce the number of write operations performed on the flash chip.*

4 FTL Design

Following our lessons from Section 3, we describe the general design principles for a *Low-Low-High FTL*—an FTL that reuses flash pages using the LLH reprogramming scheme. We assume such an FTL would run on the SSD controller, and utilize the physical page and block operations supported by the flash controller. Thus, it shares the following basic concepts with the standard FTL and SSD.

To accommodate out-of-place writes, the physical storage capacity of the drive is larger than its exported logical capacity. The drive's *overprovisioning* is defined as $\frac{T-U}{U}$, where T and U represent the number of physical and logical blocks, respectively [15]. Typical values of overprovisioning are 7% and 28% for consumer and enterprise class SSDs, respectively [45].

Whenever the number of clean blocks drops below a certain threshold, *garbage collection* is invoked. Garbage collection is typically performed *greedily*, picking the block with the minimum *valid count* (the lowest number of valid pages) as the victim for *cleaning*. The valid pages are *moved*—read and copied to another available block, and then the block is erased. These additional internal writes, referred to as *write amplification*, delay the cleaning process, and require, eventually, additional erasures. Write amplification does not accurately represent the utilization of drives that reuse pages for WOM encoded data, since some redundancy must always be added to the logical data to enable second writes [51, 52]. Thus, instead of deriving the number of erasures performed by the FTL from its write amplification, we measure them directly.

Low-Low-High (LLH) programming. Blocks in a Low-Low-High FTL cycle between four states, as depicted in Figure 6. In the initial, *clean* state all the cells are in the erased state, *ER*. If all the pages are programmed ($write\ L_1H$), the block reaches the *used* state. Alternatively, if only the low pages are used ($write\ L_1$), the block reaches the *partially-used* state. A partially-used block can be reused, in which case the FTL will reprogram all or some of the low pages and all the high pages ($write\ L_2H$), transitioning the block to the *reused* state. Alternatively, the FTL can program the high pages and leave the low pages untouched ($write\ H$), thus transitioning the block to the used state. Used and reused blocks return to the clean state when they are erased.

The choice of state transition is determined by the conditions depicted in Figure 6. The conditions that determine when to *partially use*, *use* or *reuse* a block, as well as the encoding scheme used for reprogrammed pages, are in turn determined by the specific FTL design. We next describe *LLH-FTL*—the FTL used for our evaluation.

WOM encoding. When WOM codes are employed for reusing flash pages, the FTL is responsible for determining whether a logical page is written in a first or a second write, and for recording the required metadata. The choice of WOM code determines the data written on the low pages of partially-used blocks, and the data written on them when they are reprogrammed. The encoding scheme in LLH-FTL is similar to that of ReusableSSD [53]. Data in the low pages of partially-used blocks is written as is, without storage or encoding overheads. Data written as a second write on low pages of reused blocks is encoded with a Polar WOM code that requires two physical pages to store the encoded data of one logical page [9, 10]. This WOM implementation has a 0.25% encoding failure rate.

We note that the mathematical properties of WOM codes ensure they can be applied to any data pattern, including data that was previously scrambled or compressed. In fact, WOM encoding also ensures an even distribution of zeroes throughout the page, and can thus replace data scrambling on second writes.

While manufacturers have increased the flash page size (see Table 2), the most common size used by file systems remains 4KB. Our LLH-FTL design distinguishes between the logical page used by the host and some larger physical page size. Thus, the FTL maps several logical pages onto each physical page. This allows LLH-FTL to program the encoded data for a second write on one physical page. In the rest of this section we assume that the physical page size is exactly twice the logical page size. We note that the changes required in the design if the physical pages are even larger are straightforward.

If the physical and logical page sizes are equal, a Low-Low-High FTL can utilize the multi-plane command that allows programming two physical pages in parallel on two different blocks, as in the ReusableSSD design. In both approaches, the latency required for reading or writing an encoded logical page on a second write is equal to the latency of one flash page write.

As in the design of ReusableSSD [53], LLH-FTL addresses the 0.25% probability of encoding failure by writing the respective logical page as a first write on a clean block, and prefetches the content of physical pages that are about to be rewritten to avoid the latency of an additional read. Pages are reprogrammed only in the safe portion of their block's lifetime (the first 40% in all but one of the chips we examined), thus limiting the long-term effect of reprogramming to an acceptable level.

Hot and cold data separation. Workloads typically exhibit a certain amount of skew, combining frequently updated *hot* data with infrequently written *cold* data. Separating hot and cold pages has been demonstrated as beneficial in several studies [16, 22, 25, 47]. Previous studies also showed that second writes are most beneficial for hot pages, minimizing the time in which the capacity of reused blocks is not fully utilized [31, 36, 52, 53]. In LLH-FTL, we write hot data on partially-used and reused blocks, and cold data on used blocks. Hot data on partially-used blocks is invalidated quickly, maximizing the benefit from reusing the low pages they are written on. Reused blocks store pages in first as well as in second writes. Nevertheless, we use them only for hot data, in order to maintain the separation of hot pages from cold ones. The classification of hot and cold pages is orthogonal to the design of LLH-FTL, and can be done using a variety of approaches [12, 22, 33, 47]. We describe the classification schemes used in our experiments in Section 5.

Partially-use, use and reuse conditions. The number of partially-used blocks greatly affects the performance of a Low-Low-High FTL. Too few mean that the blocks will be reused too soon, while they still contain too many valid low pages, thus limiting the benefit from reprogramming. Too many mean that too many high pages will remain unused, reducing the available overprovisioned space, which might increase internal page moves. The three conditions in Figure 6 control the number of partially-used blocks: if the partially-use condition does not hold, a clean block is used with regular LH programming. In addition, the FTL may define a use condition, which specifies the circumstances in which a partially-used block is reclaimed, and its high pages will be written without rewriting the low pages. Finally, the reuse condition ensures efficient reuse of the low pages. The FTL allows partially-used blocks to accumulate until the reuse condition is met.

Our LLH-FTL allows accumulation of at most $threshold_{pu}$ partially-used blocks. This threshold is updated in each garbage collection invocation. An increase in the valid count of the victim block compared to previous garbage collections indicates that the effective overprovisioned space is too low. In this case the threshold is *decreased*. Similarly, a decrease in the valid count indicates that page reuse is effective in reducing garbage collections, in which case the threshold is *increased* to allow more reuse. Thus, the partially-use and reuse conditions simply compare the number of partially-used blocks to the threshold. To maintain the separation between hot and cold pages, LLH-FTL does not utilize the use condition.

Expected benefit. The reduction in erasures in LLH-FTL depends on the amount of hot data in the workload, and on the number of valid pages that remain on partially-used blocks when they are reused. We assume, for the sake of this analysis, that the low pages on a reused block, as well as all the pages on an erased block, have all been invalidated. Without reprogramming, this means that there is no write amplification, and the expected number of erasures is $E{=}\frac{M}{N}$, where M is the number of logical page write requests, and N is the number of pages in each block. With LLH programming, every two low pages are reused to write an extra logical page, so $N{+}\frac{N}{4}$ logical pages are written on each block before it is erased. Let X be the portion of hot data in the workload, $0 \leq X < 1$, and recall that only blocks containing hot pages are reused. Then the expected number of erasures is $E'{=}(1{-}X)\frac{M}{N}{+}X\frac{M}{N+\frac{N}{4}}{=}E(\frac{5-X}{5})$. The maximal reduction in erasures is expected in traces where almost all the write requests access hot pages ($X \rightarrow 1$), where $E' = 0.8E$, a reduction of 20%.

For a rough estimate of the resulting lifetime extension, let us assume that all the blocks are reused in the first 40% of their lifetime, i.e., during $0.4T$ cycles. In each of these cycles, $\frac{5N}{4}$ logical pages are written on these blocks, a total of $0.5TN$. Assuming we can use the remaining $0.6T$ cycles, we write an additional $0.6TN$ pages. The total amount of data written is $1.1TN$, an increase of 10% compared to regular programming. However, we must also consider the reduction in lifetime observed in the experiments in Section 3.3. A 5%–6% reduction means that the reduction in erasures translates to a modest 4%–5% increase in lifetime.

Comparing our analysis to that of previous designs is not straightforward. Most studies, including of designs that reuse flash pages with WOM codes, did not consider the overall amount of *logical data* that could be written on the device. The only comparable analysis is that of ReusableSSD [53], which resulted in an estimated reduction of up to 33% of erasures, assuming that all the blocks (storing both hot and cold data) could be reused, and that both the low and high pages could be reprogrammed. This analysis also excluded the lifetime reduction due to reprogramming. This discrepancy leads to our next lesson.

Lesson 4: *A reduction in erasures does not necessarily translate to a substantial lifetime increase, due to the low utilization of pages that store WOM encoded data, and to the long-term effects of reprogramming. The increase in lifetime strongly depends on chip characteristics.*

5 SSD Evaluation Setup

In our FTL evaluation, we wish to quantify the possible benefit from reusing flash pages with WOM codes, when all the limitations of physical MLC flash and practical codes are considered. Thus, we measure the savings in erasures and the lifetime extension they entail, as well as the effects of LLH reprogramming on I/O performance.

5.1 OpenSSD evaluation board

We use the OpenSSD Jasmineboard [4] for our FTL evaluation. The board includes an ARM-based Indilinx$^{\text{TM}}$ Barefoot controller, 64MB of DRAM for storing the flash translation mapping and SATA buffers, and eight slots for custom flash modules, each of which can host four 64Gb 35nm MLC flash chips. The chips have two planes and 8KB physical pages. The device uses large 32KB virtual pages for improved parallelism. Thus, erase blocks are 4MB and consist of 128 contiguous virtual pages [4].

On the OpenSSD board, an FTL that uses 8KB pages rather than 32KB virtual pages incurs unacceptable latencies [43]. Thus, we use a mapping granularity of 4KB logical pages and a merge buffer that ensures that data is written at virtual-page granularity [31, 43]. The downside of this optimization is an exceptionally large block size (1024 logical pages) that increases the valid count of used and partially-used blocks. As a result, garbage collection entails more page moves, and reprogramming is possible on fewer pages.

We were also unable to fully implement WOM encoded second writes on the OpenSSD board. While mature and commonly used error correction codes are implemented in hardware, the Polar WOM codes used in our design are currently only available with software encoding and decoding. These implementations are prohibitively slow, and are thus impractical for latency evaluation purposes. In addition, in OpenSSD, only the ECC hardware accelerator is allowed to access the page spare area, and cannot be disabled. Thus, reprogrammed pages will always appear as corrupted when compared with their corresponding ECC. This also prevents the FTL from utilizing the page spare area for encoding purposes [53]. We address these limitations in our FTL implementation described below.

5.2 FTL Implementation

The FTL used on the OpenSSD board is implemented in software, and can thus also be used as an *emulator* of SSD performance when executed on a standard server without being connected to the actual board. Replaying a workload on the emulator is considerably faster than on the board itself, because it does not perform the physical flash operations. We validated this emulator, ensuring that it reports the same *count* of flash operations as would be performed on the actual board. Thus, using the emulator, we were able to experiment with a broad set of setups and parameters that are impractical on the Jasmine board. In particular, we were able to evaluate an FTL that uses 8KB physical pages, rather than the 32KB physical pages mandated by the limitations of the board.We refer to the FTL versions with 32KB pages as $\langle FTL\ name \rangle$-32.

We first implemented a *baseline* FTL that performs only first writes on all the blocks. It employs greedy garbage collection within each bank and separates hot and cold pages by writing them on two different active blocks. The identification of hot pages is described in Section 5.3. We also implemented LLH-FTL described in Section 4.[4] It uses greedy garbage collection for choosing the block with the minimum number of valid logical pages among used and reused blocks. Garbage collection is triggered whenever a clean block should be allocated and no such block is available. If the number of partially-used blocks is lower than the threshold and a hot active block is required, LLH-FTL allocates the partially-used block with the minimum valid count. If the threshold is exceeded or if a cold active block is required, it allocates a new clean block.

The threshold is updated after each garbage collection, taking into account the valid count in w previous garbage collections. Due to lack of space, we present results only for $w = 5$, and two initial threshold values, which were the most dominant factor in the performance of LLH-FTL.

LLH-FTL reuses low pages on partially-used blocks only if all the logical pages on them have been invalidated. LLH-FTL-32 writes four logical pages on each reused physical pages, requiring eight consecutive invalid logical pages in order to reuse a low page. We evaluate the effect of this limitation on LLH-FTL-32 in Section 6.

Our implementation of LLH-FTL does not include actual WOM encoding and decoding for the reasons described above. Instead, it writes arbitrary data during reprogramming of low pages, and ignores the ECC when reading reprogrammed data. In a real system, the WOM encoding and decoding would be implemented in hardware, and incur the same latency as the ECC. Thus, in our evaluation setup, their overheads are simulated by the ECC computations on the OpenSSD board. Coding failures are simulated by a random "coin flip" with the appropriate probability. To account for the additional prefetch-

[4] The code for the emulator and FTLs is available online [1, 2].

Volume	Requests (M)	Drive size (GB)	Requests/sec	Write ratio	Hot write ratio	Total writes (GB)
src1_2	2	16	3.15	0.75	0.22	45
stg_0	2		3.36	0.85	0.85	16
hm_0	4	32	6.6	0.64	0.7	23
rsrch_0	1.5		2.37	0.91	0.95	11
src2_0	1.5		2.58	0.89	0.91	10
ts_0	2		2.98	0.82	0.94	12
usr_0	2.5		3.7	0.6	0.86	14
wdev_0	1		1.89	0.8	0.85	7
prxy_0	12.5	64	20.7	0.97	0.67	83
proj_0	4		6.98	0.88	0.14	145
web_0	2		3.36	0.7	0.87	17
online	5.5	16	3.14	0.74	0.31	16
webresearch	3		1.8	1	0.41	13
webusers	8		4.26	0.9	0.32	27
webmail	8	32	4.3	0.82	0.3	24
web-online	14		7.88	0.78	0.31	43
zipf(0.9,0.95,1)	12.5	16	200	1	0.5	48

Table 6: Trace characteristics of MSR (top box), FIU (middle), and synthetic (bottom) workloads.

ing of the invalid data, this data is read from the flash into the DRAM, but is never transferred to the host.

5.3 Workloads

We use real world traces from two sources. The first is the MSR Cambridge workload [5, 35], which contains week-long traces from 36 volumes on 13 servers. The second is a set of traces from FIU [3, 29], collected during three weeks on servers of the computer science department. Some of the volumes are too big to fit on the drive size supported by our FTL implementation, which is limited by the DRAM available on the OpenSSD. We used the 16 traces whose address space fit in an SSD size of 64GB or less, and that include enough write requests to invoke the garbage collector on that drive. These traces vary in a wide range of parameters, summarized in Table 6. We also used three synthetic workloads with a Zipf distribution with exponential parameter $\alpha = 0.9, 0.95$ and 1.

We used a different hot/cold classification heuristic for each set of traces. For the MSR traces, pages were classified as cold if they were written in a request of size 64KB or larger. This simple online heuristic was shown to perform well in several previous studies [12, 22, 53]. In the FIU traces, all the requests are of size 4KB, so accesses to contiguous data chunks appear as sequential accesses. We applied a similar heuristic by tracking previously accessed pages, and classifying pages as cold if they appeared in a sequence of more than two consecutive pages. In the synthetically generated Zipf traces, the frequency of access to page n is proportional to $\frac{1}{\alpha^n}$. Thus, we extracted the threshold n for each Zipf trace, such that pages with logical address smaller than n were accessed 50% of the time, and pages with logical address larger than n were classified as cold. While this classification is impossible in real world settings, these traces demonstrate the benefit from page reuse under optimal conditions.

Each workload required a different device size, and thus, a different number of blocks. In order to maintain the same degree of parallelism in all experiments, we always configured the SSD with 16 banks, with 256, 512 and 1024 4MB blocks per bank for drives of size 16GB, 32GB and 64GB, respectively. Pages were striped across banks, so that page p belonged to bank $b = p \ mod \ 16$.

6 Evaluation

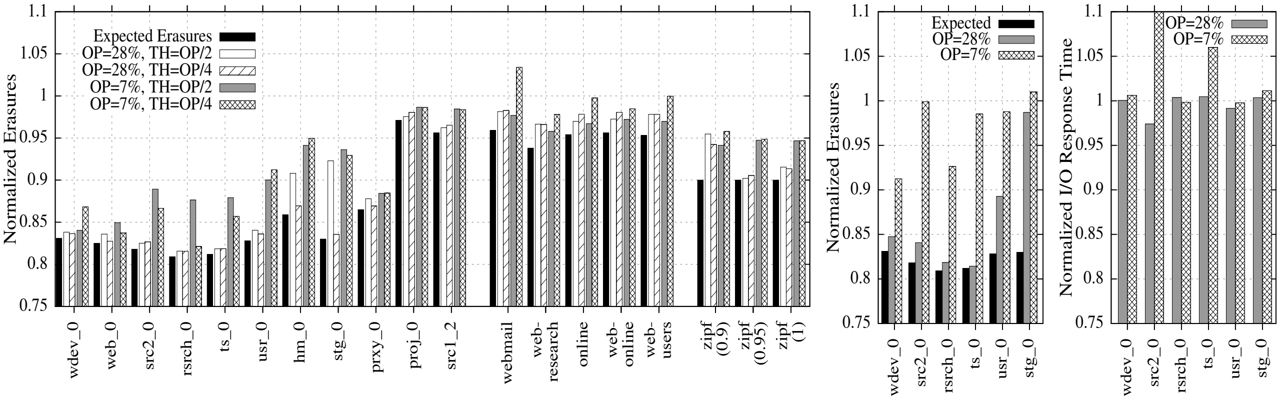
Reduction in erasures. To verify that the expected reduction in erasures from LLH reprogramming can be achieved in real workloads, we calculated the expected reduction for each workload according to the formula in Section 4. We then used the emulator to compare the number of erasures performed by the baseline and LLH-FTL. Our results are presented in Figure 7(a), where the workloads are aggregated according to their source (and corresponding hot page classification) and ordered by the amount of data written divided by the corresponding drive size. Our results show that the normalized number of erasures is between 0.8 and 1. The reduction in erasures mostly depends on the workload and the amount of hot data in it.

The amount of overprovisioning (OP) substantially affects the benefit from reprogramming. With 28% overprovisioning, the reduction in erasures is very close to the expected reduction. Low overprovisioning is known to incur excessive internal writes. Thus, with the already low 7% overprovisioning, reserving partially-used blocks for additional writes was not as efficient for reducing erasures; it might increase the number of erasures instead. The adaptive $threshold_{pu}$ avoids this situation quite well, as it is decreased whenever the valid count increases. Still, the reduction in erasures is smaller than with OP=28% because both the low overprovisioning and low threshold result in more valid logical pages on the partially-used blocks, allowing fewer pages to be reused.

The time required for the adaptive $threshold_{pu}$ to converge depends on its initial value. In setups where the reservation of partially-used blocks is useful, such as high overprovisioning, LLH-FTL with initial $threshold_{pu}$ = OP/2 achieves greater reduction than with $threshold_{pu}$ = OP/4, because a higher initial value means that the optimal value is found earlier. The difference between the two initial values is smaller for traces that write more data, allowing the threshold more time to adapt.

The quality of the hot data classification also affected the reduction in erasures. While the baseline and LLH-FTL use the same classification, misclassification interferes with page reuse in a manner similar to low overprovisioning, as it increases the number of valid logical pages during block erase and reuse. This effect is demonstrated in the lower reductions achieved on the FIU workloads, in which classification was based on a naive heuristic.

We repeated the above experiments with LLH-FTL-32,



(a) LLH-FTL (b) LLH-FTL-32 (c) LLH-FTL-32

Figure 7: (a) Normalized number of erasures (compared to baseline) of LLH-FTL (b) Normalized number of erasures (compared to baseline-32) of LLH-FTL-32 (c) Normalized I/O response time (compared to baseline-32) of LLH-FTL-32 (c).

to evaluate the effect of increasing the physical page size. Indeed, the reduction in erasures was smaller than with 8KB pages, although the differences were minor. The average difference was 1% with 28% overprovisioning, but it was 6% with 7% overprovisioning because of the higher number of leftover valid logical pages on each physical page in partially-used blocks.

I/O response time. To evaluate the effect of LLH reprogramming on I/O response time, we replayed the workloads on the OpenSSD board. We warmed up the board by filling the SSD and then replaying the workload twice, measuring the I/O response time in the last 12 hours of each workload. We accelerated the workloads by a factor of 10 in order to speed up the experiment. While maintaining the original access pattern, the accelerated request rate is more realistic for workloads that use SSDs.

We use LLH-FTL-32 with the optimal initial $threshold_{pu}$ for representative MSR traces. Figure 7(b) shows the normalized number of erasures compared to baseline-32, and Figure 7(c) shows the normalized I/O response time of LLH-FTL-32. Despite the considerable reduction in erasures, and thus, garbage collection invocations, the average I/O response time is almost unchanged. The 90th and 99th percentiles were also similar. This contradicts previous simulation results [53] that correlated the reduction in erasures with a reduction in I/O response time.

One reason for this discrepancy is that the accumulation of write requests in the merge buffers in OpenSSD causes writes to behave asynchronously—the write request returns to the host as complete once the page is written in the buffer. Flushing the merge buffer onto a physical flash page is the cause for latency in writes. The baseline flushes the buffer whenever eight logical pages are accumulated in it. However, a buffer containing WOM encoded data must be flushed after accumulating four logical pages, possibly incurring additional latency. This effect was not observed in previous studies that used a simulator that flushes all writes synchronously.

The average I/O response time does not *increase* because even though the trace is accelerated, the extra buffer flushes usually do not delay the following I/O requests. In addition, due to the allocation of partially-used and reused pages for hot data, this data is more likely to reside on low pages, which are faster to read and program [18].

The second reason for the discrepancy is the reservation of partially-used blocks for reprogramming. This reduces the available overprovisioned capacity, potentially increasing the number of valid pages that must be copied during each garbage collection. As a result, although the number of erasures decreased, the total amount of data copied by LLH-FTL-32 was similar to that copied by the baseline, and sometimes higher (by up to 50%). One exception is the src1_2 workload, where in the last 12 hours, garbage collection in LLH-FTL-32 moved less data than in baseline-32. In the other traces, the total delay caused by garbage collections was not reduced, despite the considerably lower number of erasures.

Energy consumption. We used the values from Table 5 and the operation counts from the emulator to compare the energy consumption of LLH-FTL-32 to that of baseline-32. The energy measurements were done on the A16 chip, whose page size is 16KB. We doubled the values for the read and program operations to estimate the energy for programming 32KB pages as in LLH-FTL-32. Figure 8 shows that when reprogramming reduced erasures, the energy consumed by LLH-FTL-32 increased with inverse proportion to this reduction. This is not surprising, since the reduction in erasures does not reduce the amount of internal data copying in most of the workloads. In the FIU traces with 7% OP, reprogramming increased the number of erasures due to increased internal writes, which, in turn, also increased the energy consumption.

Lesson 5: *A reduction in erasures does not necessarily translate to a reduction in I/O response time or energy consumption. These are determined by the overall amount*

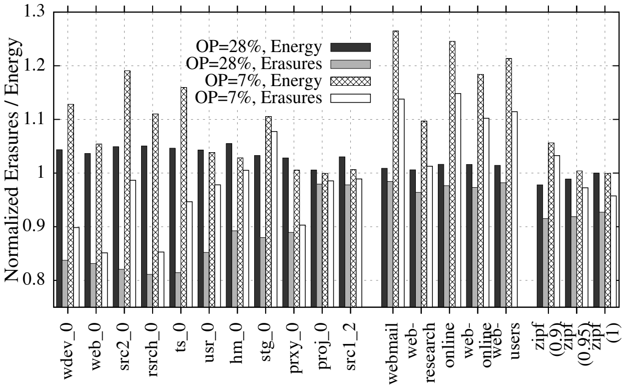


Figure 8: Normalized energy consumption (compared to baseline-32) of LLH-FTL-32 with two overprovisioning values.

of data moved during garbage collections. Designs that are aimed at reducing energy consumption or I/O response time should address these objectives explicitly.

7 Related Work

Several studies proposed FTL designs that reuse pages to extend SSD lifetime. Some are based on capacity achieving codes, and bound the resulting capacity loss by limiting second writes to several blocks [36] or by assuming the logical data has been compressed by the upper level [24]. The overheads and complexities in these designs are addressed in the design of ReusableSSD [53]. However, none of these studies addressed the limitations of reprogramming MLC flash pages. Some of these limitations were addressed in the design of an overwrite compatible B^+-tree data structure, assuming the mapping of V_{th} to bits can be modified [26]. Like the previous approaches, it has been implemented only in simulation. Extended P/E cycles [31] were implemented on real hardware, but the FTL that uses them relies on the host to supply and indicate data that is overwrite compatible. LLH-FTL is the first general-purpose FTL that addresses all practical limitations of WOM codes as well as MLC flash. Thus, we were able to demonstrate its strengths and weaknesses on real hardware and workloads.

Numerous studies explored the contributors to BER in flash, on a wide variety of chip technologies and manufacturers. They show the effects of erasures, retention, program disturbance and scaling down technology on the BER [11, 18, 32, 48]. These studies demonstrate a trend of increased BER as flash feature sizes scale down, and the need for specialized optimizations employed by manufacturers as a result. Thus, we believe that some of the interference effects observed in our experiments are a result of optimizing the chips for regular LH programming. Adjusting these optimizations to LLH reprogramming is a potential approach to increase the benefit from page reuse.

Several studies examined the possibility of reprogramming flash cells. Most used either SLC chips [24], or MLC chips as if they were SLC [17]. A thorough study on 50nm and 72nm MLC chips demonstrated that after a full use of the block (LH programming), half of the pages are "WOM-safe" [18]. However, they do not present the exact reprogramming scheme, nor the problems encountered when using other schemes. A recent study [31] mapped all possible state transitions with reprogramming on a 35nm MLC chip, and proposed the LLH reprogramming scheme. Our results in Section 3 show that smaller feature sizes impose additional restrictions on reprogramming, but that LLH reprogramming is still possible.

Previous studies examined the energy consumption of flash chips as a factor of the programmed pattern and page [34], and suggested methods for reducing the energy consumption of the flash device [41]. To the best of our knowledge, this study is the first to measure the effect of reprogramming on the energy consumption of a real flash chip and incorporate it into the evaluation of the FTL.

8 Conclusions

Our study is the first to evaluate the possible benefit from reusing flash pages with WOM codes on real flash chips and an end-to-end FTL implementation. We showed that page reuse in MLC flash is possible, but can utilize only half of the pages and only if some of its capacity has been reserved in advance. While reprogramming is safe for at least 40% of the lifetime of the chips we examined, it incurs additional *long-term* wear on their blocks. Thus, even with an impressive 20% reduction in *erasures*, the increase in *lifetime* strongly depends on chip physical characteristics, and is fairly modest.

A reduction in erasures does not necessarily translate to a reduction in I/O response time or energy consumption. These are determined by the overall amount of data moved during garbage collections, which strongly depends on the overprovisioning. The reduction in physical flash page writes is limited by the storage overhead of WOM encoded data, and is mainly constrained by the limitation of reusing only half of the block's pages.

This study exposed a considerable gap between the previously shown benefits of page reuse, which were based on theoretical analysis and simulations, and those that can be achieved on current state-of-the-art hardware. However, we believe that most of the limitations on these benefits can be addressed with manufacturer support, and that the potential benefits of page reuse justify reevaluation of current MLC programming constraints.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Andrea Arpaci-Dusseau, whose suggestions helped improve this paper. We also thank Alex Yucovich and Hila Arobas for their help with the low-level experiments. This work was supported in part by BSF grant 2010075, NSF grant CCF-1218005, ISF grant 1624/14 and EU Marie Curie Initial Training Network SCALUS grant 238808.

References

[1] https://github.com/zdvresearch/fast2016-ftl.

[2] https://github.com/zdvresearch/fast2016-openssd-emulator.

[3] I/O deduplication traces. http://sylab-srv.cs.fiu.edu/doku.php?id=projects:iodedup:start. Retrieved: 2014.

[4] Jasmine OpenSSD platform. http://www.openssd-project.org/.

[5] SNIA IOTTA. http://iotta.snia.org/traces/388. Retrieved: 2014.

[6] NAND flash memory tester (SigNASII). http://www.siglead.com/eng/innovation_signas2.html, 2014.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference (ATC)*, 2008.

[8] A. Berman and Y. Birk. Retired-page utilization in write-once memory – a coding perspective. In *IEEE International Symposium on Information Theory (ISIT)*, 2013.

[9] D. Burshtein. Coding for asymmetric side information channels with applications to polar codes. In *IEEE International Symposium on Information Theory (ISIT)*, 2015.

[10] D. Burshtein and A. Strugatski. Polar write once memory codes. *IEEE Transactions on Information Theory*, 59(8):5088–5101, 2013.

[11] Y. Cai, O. Mutlu, E. Haratsch, and K. Mai. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *31st IEEE International Conference onComputer Design (ICCD)*, 2013.

[12] M.-L. Chiao and D.-W. Chang. ROSE: A novel flash translation layer for NAND flash memory based on hybrid address translation. *IEEE Transactions on Computers*, 60(6):753–766, 2011.

[13] G. D. Cohen, P. Godlewski, and F. Merkx. Linear binary code for write-once memories. *IEEE Transactions on Information Theory*, 32(5):697–700, 1986.

[14] 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 *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[15] P. Desnoyers. What systems researchers need to know about NAND flash. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[16] P. Desnoyers. Analytic models of SSD write performance. *Trans. Storage*, 10(2):8:1–8:25, Mar. 2014.

[17] E. En Gad, H. W., Y. Li, and J. Bruck. Rewriting flash memories by message passing. In *IEEE International Symposium on Information Theory (ISIT)*, 2015.

[18] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[19] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[20] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with lazy adaptive replacement. In *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, 2013.

[21] J.-W. Im et al. A 128Gb 3b/cell V-NAND flash memory with 1gb/s i/o rate. In *IEEE International Solid-State Circuits Conference (ISSCC)*, 2015.

[22] S. Im and D. Shin. ComboFTL: Improving performance and lifespan of MLC flash memory using SLC flash buffer. *J. Syst. Archit.*, 56(12):641–653, Dec. 2010.

[23] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin. Writing cosets of a convolutional code to increase the lifetime of flash memory. In *50th Annual Allerton Conference on Communication, Control, and Computing*, 2012.

[24] A. Jagmohan, M. Franceschini, and L. Lastras. Write amplification reduction in NAND flash through multi-write coding. In *26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[25] X. Jimenez, D. Novo, and P. Ienne. Wear unleveling: Improving NAND flash lifetime by balancing page endurance. In *12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[26] J. Kaiser, F. Margaglia, and A. Brinkmann. Extending SSD lifetime in database applications with page overwrites. In *6th International Systems and Storage Conference (SYSTOR)*, 2013.

[27] T. Kgil, D. Roberts, and T. Mudge. Improving NAND flash based disk caches. In *35th Annual International Symposium on Computer Architecture (ISCA)*, 2008.

[28] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *6th USENIX Conference on File and Storage*

Technologies (FAST), 2008.

[29] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *Trans. Storage*, 6(3):13:1–13:26, Sept. 2010.

[30] X. Luojie, B. M. Kurkoski, and E. Yaakobi. WOM codes reduce write amplification in NAND flash memory. In *IEEE Global Communications Conference (GLOBECOM)*, 2012.

[31] F. Margaglia and A. Brinkmann. Improving MLC flash performance and endurance with extended P/E cycles. In *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[32] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. Nevill. Bit error rate in NAND flash memories. In *Reliability Physics Symposium (IRPS). IEEE International*, 2008.

[33] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[34] V. Mohan, T. Bunker, L. Grupp, S. Gurumurthi, M. Stan, and S. Swanson. Modeling power consumption of NAND flash memories using FlashPower. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(7):1031–1044, July 2013.

[35] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23, Nov. 2008.

[36] S. Odeh and Y. Cassuto. NAND flash architectures reducing write amplification through multi-write codes. In *IEEE 30th Symposium on Mass Storage Systems and Technologies (MSST)*, 2014.

[37] Y. Oh, J. Choi, D. Lee, and S. H. Noh. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[38] H. Park, J. Kim, J. Choi, D. Lee, and S. Noh. Incremental redundancy to reduce data retention errors in flash-based SSDs. In *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[39] K.-T. Park, M. Kang, D. Kim, S.-W. Hwang, B. Y. Choi, Y.-T. Lee, C. Kim, and K. Kim. A zeroing cell-to-cell interference page architecture with temporary LSB storing and parallel MSB program scheme for MLC NAND flash memories. *IEEE Journal of Solid-State Circuits*, 43(4):919–928, April 2008.

[40] R. L. Rivest and A. Shamir. How to Reuse a Write-Once Memory. *Inform. and Contr.*, 55(1-3):1–19, Dec. 1982.

[41] M. Salajegheh, Y. Wang, K. Fu, A. Jiang, and E. Learned-Miller. Exploiting half-wits: Smarter storage for low-power devices. In *9th USENIX Conference on File and Stroage Technologies (FAST)*, 2011.

[42] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: A lightweight, consistent and durable storage cache. In *7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[43] M. Saxena, Y. Zhang, M. M. Swift, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Getting real: Lessons in transitioning research simulations into hardware systems. In *11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[44] A. Shpilka. Capacity achieving multiwrite WOM codes. *IEEE Transactions on Information Theory*, 60(3):1481–1487, 2014.

[45] K. Smith. Understanding SSD over-provisioning. *EDN Network*, January 2013.

[46] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD lifetimes with disk-based write caches. In *8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[47] R. Stoica and A. Ailamaki. Improving flash write performance by using update frequency. *Proc. VLDB Endow.*, 6(9):733–744, July 2013.

[48] E. Yaakobi, L. Grupp, P. Siegel, S. Swanson, and J. Wolf. Characterization and error-correcting codes for TLC flash memories. In *International Conference on Computing, Networking and Communications (ICNC)*, 2012.

[49] E. Yaakobi, S. Kayser, P. H. Siegel, A. Vardy, and J. K. Wolf. Codes for write-once memories. *IEEE Transactions on Information Theory*, 58(9):5985–5999, 2012.

[50] E. Yaakobi, J. Ma, L. Grupp, P. H. Siegel, S. Swanson, and J. K. Wolf. Error characterization and coding schemes for flash memories. In *IEEE GLOBECOM Workshops (GC Wkshps)*, 2010.

[51] E. Yaakobi, A. Yucovich, G. Maor, and G. Yadgar. When do WOM codes improve the erasure factor in flash memories? In *IEEE International Symposium on Information Theory (ISIT)*, 2015.

[52] G. Yadgar, R. Shor, E. Yaakobi, and A. Schuster. It's not where your data is, it's how it got there. In *7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2015.

[53] G. Yadgar, E. Yaakobi, and A. Schuster. Write once, get 50% free: Saving SSD erase costs using WOM codes. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[54] G. Yadgar, A. Yucovich, H. Arobas, E. Yaakobi, Y. Li, F. Margaglia, A. Brinkmann, and A. Schuster. Limitations on MLC flash page reuse and its effects on durability. Technical Report CS-2016-02, Computer Science Department, Technion, 2016.

[55] J. Yang, N. Plasson, G. Gillis, and N. Talagala. HEC: Improving endurance of high performance flash-based cache devices. In *6th International Systems and Storage Conference (SYSTOR)*, 2013.

[56] K. Zhao, W. Zhao, H. Sun, X. Zhang, N. Zheng, and T. Zhang. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

Reducing Solid-State Storage Device Write Stress Through Opportunistic In-Place Delta Compression

Xuebin Zhang[⋆], Jiangpeng Li[⋆], Hao Wang[⋆], Kai Zhao[†] and Tong Zhang[⋆]
[⋆]*ECSE Department, Rensselaer Polytechnic Institute, USA*
[†]*SanDisk Corporation, USA*
{*xuebinzhang.rpi@gmail.com, tzhang@ecse.rpi.edu*}

Abstract

Inside modern SSDs, a small portion of MLC/TLC NAND flash memory blocks operate in SLC-mode to serve as write buffer/cache and/or store hot data. These SLC-mode blocks absorb a large percentage of write operations. To balance memory wear-out, such MLC/TLC-to-SLC configuration rotates among all the memory blocks inside SSDs. This paper presents a simple yet effective design approach to reduce write stress on SLC-mode flash blocks and hence improve the overall SSD lifetime. The key is to implement well-known delta compression without being subject to the read latency and data management complexity penalties inherent to conventional practice. The underlying theme is to leverage the partial programmability of SLC-mode flash memory pages to ensure that the original data and all the subsequent deltas always reside in the same memory physical page. To avoid the storage capacity overhead, we further propose to combine intra-sector lossless data compression with intra-page delta compression, leading to opportunistic in-place delta compression. This paper presents specific techniques to address important issues for its practical implementation, including data error correction, and intra-page data placement and management. We carried out comprehensive experiments, simulations, and ASIC (application-specific integrated circuit) design. The results show that the proposed design solution can largely reduce the write stress on SLC-mode flash memory pages without significant latency overhead and meanwhile incurs relatively small silicon implementation cost.

1 Introduction

Solid-state data storage built upon NAND flash memory is fundamentally changing the storage hierarchy for information technology infrastructure. Unfortunately, technology scaling inevitably brings the continuous degradation of flash memory endurance and write speed. Motivated by data access locality and heterogeneity in real-world applications, researchers have well demonstrated the effectiveness of complementing bulk MLC/TLC NAND flash memory with small-capacity SLC NAND flash memory to improve the endurance and write speed (e.g., see [1–3]). The key is to use SLC memory blocks serve as write buffer/cache and/or store relatively hot data. Such a design strategy has been widely adopted in commercial solid-state drives (SSDs) [4–6], where SSD controllers dynamically configure a small portion of MLC/TLC flash memory blocks to operate in SLC mode. The MLC/TLC-to-SLC configuration rotates throughout all the MLC/TLC flash memory blocks in order to balance the flash memory wear-out.

This paper is concerned with reducing the write stress on those SLC-mode flash memory blocks in SSDs. Aiming to serve as write buffer/cache and/or store hot data, SLC-mode flash memory blocks account for a large percentage of overall data write traffic [7]. Reducing their write stress can directly reduce the flash memory wear-out. Hence, when these SLC-mode memory blocks are configured back to operate as normal MLC/TLC memory blocks, they could have a long cycling endurance. Since a specific location tends to be repeatedly visited/updated within a short time (like consecutive metadata updates or in-place minor revisions of file content), it is not uncommon that data written into this SLC-mode flash based cache have abundant temporal redundancy. Intuitively, this feature makes the *delta compression* an appealing option to reduce the write stress. In fact, the abundance of data temporal redundancy in real systems has inspired many researchers to investigate the practical implementation of delta compression at different levels, such as filesystems [8, 9], block device [10–13] and FTL (Flash Translation Layer) [14]. Existing solutions store the original data and all the subsequent compressed deltas separately at different physical pages of the storage devices. As a result, to serve a read request, they must fetch the original data and all the subsequent deltas from different physical pages, leading to inherent read amplification, particularly for small read request or largely accumulated delta compression. In addition, the system needs to keep the mapping information for the original data and all the compressed deltas, leading to a sophisticated data structure in the filesystem and/or firmware. These issues inevitably lead to significant read latency and hence a system performance penalty.

This paper aims to implement delta compression for SLC-mode flash memory blocks with small read latency penalty and very simple data management. First, we note that the read latency penalty inherent to existing delta compression design solutions is fundamentally due to the per-sector/page *atomic write* inside storage devices, which forces us to store the original data and all the subsequent deltas across different sectors/pages. Although per-sector atomic write is essential in hard disk drives (i.e., hard disk drives cannot perform partial write/update within one 4kB sector), per-page atomic write is not absolutely necessary in NAND flash memory. Through experiments with 20nm MLC NAND flash memory chips, we observed that SLC-mode pages can support partial programming, i.e., different portions of the same SLC-mode page can be programmed at different times. For example, given a 16kB flash memory page size, we do not have to write one entire 16kB page at once, and instead we can write one portion (e.g., 4kB or even a few bytes) at a time and finish writing the entire 16kB page over a period of time. This clearly warrants re-thinking the implementation of delta compression.

Leveraging the per-page partial-programming support of SLC-mode flash memory, we propose a solution to implement delta compression without incurring significant read latency penalty and complicating data management. The key idea is simple and can be described as follows. When a 4kB sector is being written the first time, we always try to compress it before writing to an SLC-mode flash memory page. Assume the flash memory page size is 16kB, we store four 4kB sectors in each page as normal practice. The use of per-sector lossless compression leaves some memory cells unused in the flash memory page. Taking advantage of the per-page partial-programming support of SLC-mode flash memory, we can directly use those unused memory cells to store subsequent deltas later on. As a result, the original data and all its subsequent deltas reside in the same SLC flash memory physical page. Since the runtime compression/decompression can be carried out by SSD controllers much faster than a flash memory page read, this can largely reduce the data access latency overhead in the realization of delta compression. In addition, it can clearly simplify data management since everything we need to re-construct the latest data is stored in a single flash page. This design strategy is referred to as *opportunistic in-place delta compression*.

For the practical implementation of the proposed design strategy, this paper presents two different approaches to layout the data within each SLC-mode flash memory page, aiming at different trade-offs between write stress reduction and flash-to-controller data transfer latency. We further develop a hybrid error correction coding (ECC) design scheme to cope with the significantly different data size among original data and compressed deltas. We carried out experiments and simulations to evaluate the effectiveness of proposed design solutions. First, we verified the feasibility of SLC-mode flash memory page partial programming using a PCIe FPGA-based flash memory characterization hardware prototype with 20nm MLC NAND flash memory chips. For the two different data layout approaches, we evaluated the write stress reduction under a variety of delta compression values, and quantitatively studied their overall latency comparison. To estimate the silicon cost induced by the hybrid ECC design scheme and on-the-fly compression/decompression, we further carried out ASIC (application-specific integrated circuit) design, and the results show that the silicon cost is not significant. In summary, the contributions of this paper include:

- We for the first time propose to cohesively integrate SLC-mode flash memory partial programmability, data compressibility and delta compressibility to reduce write stress on SLC-mode pages in SSDs without incurring significant read latency and storage capacity penalty;
- We develop specific solutions to address the data error correction and data management design issues in the proposed opportunistic delta-compression design strategy;
- We carried out comprehensive experiments to demonstrate its effectiveness on reducing write stress at small read latency overhead and show its practical silicon implementation feasibility.

2 Background and Motivation

2.1 Write Locality

The content temporal locality in storage system implies that one specific page could be visited for multiple times within a short time period. To quantitatively investigate this phenomenon, we analyzed several typical traces including Finance-1, Finance-2 [15], Homes [16] and Webmail Server traces [16], and their information is listed in Table 1. We analyzed the percentage of repeated LBA (logical block address) in the collected traces. Figure 1 shows the distribution of repeated overwrite times within one hour. In the legend, '1' means a specific LBA is only visited once while '2-10' means an LBA is visited more than twice and less than 10 times. We can find more than 90% logical blocks are updated more than once in Finance-1 and Finance-2.

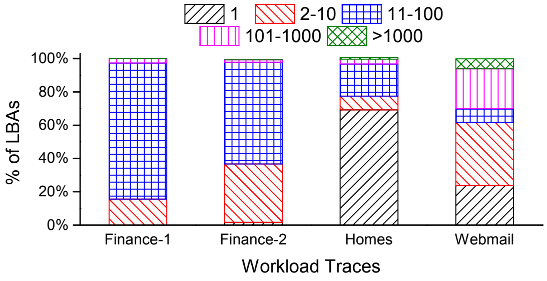


Figure 1: Percentage of repeated overwrite times of several typical workload traces.

Table 1: Disk traces information

Name	duration	# of unique LBAs	# of total LBAs
Finance-1	1h	109,177	3,051,388
Finance-2	1h	31,625	571,529
Homes	24h	20,730	28,947
Webmail	24h	6,853	16,514

Another noticeable characteristic in most applications is the partial page content overwrite or update. Authors in [17] revealed that more than 60% of write operations involve partial page overwrites and some write operations even only update less than 10 bytes. This implies a significant content similarity (or temporal redundancy) among consecutive data writes to the same LBA. However, due to the page-based data write in flash memory, such content temporal redundancy is however left unexplored in current conventional practice.

2.2 Delta Compression

Although delta compression can be realized at different levels spanning filesystems [8, 9], block device [10–13] and FTL [14], their basic strategy is very similar and can be illustrated in Figure 2. For the sake of simplicity, we consider the case of applying delta compression to the 4kB content at the LBA of L_a. Let C_0 denote the original content at the LBA of L_a, which is stored in one flash memory physical page P_0. At time T_1, we update the 4kB content at LBA of L_a with C_1. Under delta compression, we obtain the compressed delta between C_0 and C_1, denoted as d_1, and store in another flash memory physical page P_1. At time T_2, we update the content again with C_2. To maximize the delta compression efficiency, we obtain and store the compressed delta between C_2 and C_1, denoted as d_2. The process continues as we keep updating the content at the LBA of L_a, for which we need to keep the original content C_0 and all the subsequent deltas (i.e., d_1, d_2, $\cdots$).

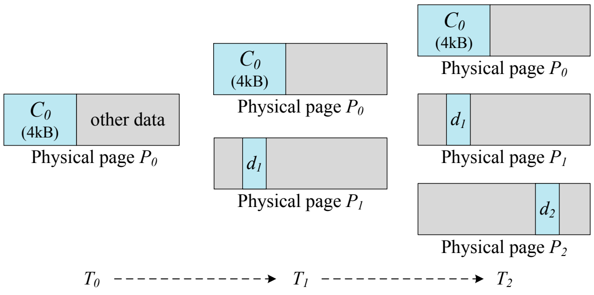


Figure 2: Illustration of conventional method for realizing temporal redundancy data compression.

Clearly, conventional practice could result in noticeable read latency penalty. In particular, to serve each read request, we must fetch the original data and all the deltas in order to re-construct the current content, leading to read amplification and hence latency penalty. In addition, it comes with sophisticated data structure and hence complicates data management, which could further complicate flash memory garbage collection. As a result, although delta compression can very naturally exploit abundant temporal redundancy inherent in many applications, it has not been widely deployed in practice.

2.3 Partial Programming

Through experiments with flash memory chips, we observed that SLC-mode NAND flash memory can readily support partial programming, i.e., different portions in an SLC flash memory page can be programmed at different time. This feature can be explained as follows. Each SLC flash memory cell can operate in either *erased* state or *programmed* state, corresponding to the storage of '1' and '0', respectively. At the beginning, all the memory cells within the same flash memory page are erased simultaneously, i.e., the storage of each memory cell is reset to be '1'. During runtime, if we write a '1' to one memory cell, memory chip internal circuits simply apply a prohibitive bit-line voltages to prevent this cell from being programmed; if we write a '0' to one memory cell, memory chip internal circuits apply a programming bit-line voltage to program this cell (i.e., move from erased state to programmed state). Meanwhile, a series of high voltage are applied to the word-line to enable programming. This can directly enable partial programming as illustrated in Figure 3: At the beginning of T_0, all the four memory cells m_1, m_2,

m_3 and m_4 are in the erased state, and we write ‘0’ to memory cell m_3 and write ‘1’ to the others. Internally, the memory chip applies programming bit-line voltage to m_3 and prohibitive bit-line voltage to the others, hence the storage content becomes $\{$‘1’, ‘1’, ‘0’, ‘1’$\}$. Later at time T_1, if we want to switch memory cell m_1 from ‘1’ to ‘0’, we write ‘0’ to memory cell m_1 and ‘1’ to the others. Accordingly, memory chip applies prohibitive bit-line voltage to the other three cells so that their states remain unchanged. As a result, the storage content becomes $\{$‘0’, ‘1’, ‘0’, ‘1’$\}$.

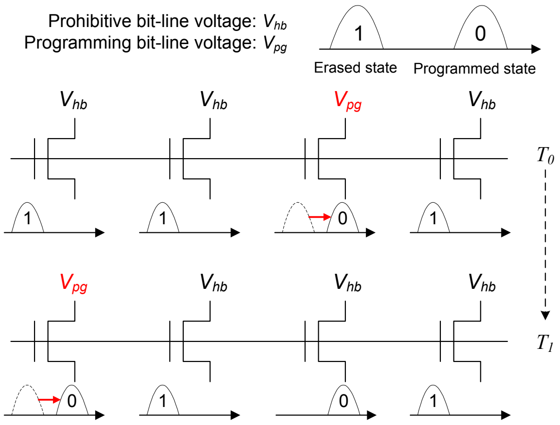


Figure 3: Illustration of the underlying physics enabling SLC-mode flash memory partial programming.

Therefore, we can carry out partial programming to SLC-mode flash memory pages as illustrated in Figure 4. Let $\mathbf{I}_s$ denote an all-one bit vector with the length of s. Given an erased SLC flash memory page with the size of L, we first write $[d_1, \cdots, d_n, \mathbf{I}_{L-n}]$ to partially program the first n memory cells and leave the rest $L-n$ memory cells intact. Later on, we can write $[\mathbf{I}_n, c_1, \cdots, c_m, \mathbf{I}_{L-n-m}]$ to partially program the next m memory cells and leave all the other memory cells intact. The same process can continue until the entire page has been programmed.

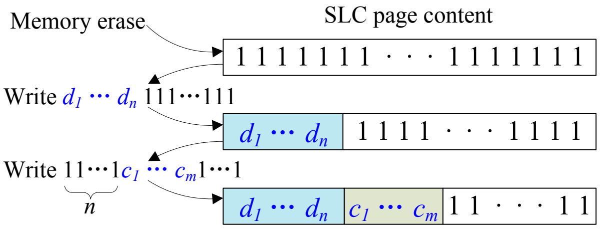


Figure 4: Illustration of SLC-mode flash memory partial programming.

Using 20nm MLC NAND flash memory chips, we carried out experiments and the results verify that the chips can support the partial programming when being operated in the SLC mode. In our experiments, we define “one cycle” as progressively applying partial programming for 8 times before one entire page is filled up and then being erased. In contrast, the conventional “one cycle” is to fully erase before each programming. Figure 5 demonstrates the bit error rate comparison of these two schemes. The flash memory can be used for 8000 cycles with the conventional way. The progressive partial programming can work for more than 7100 cycles. And this modest endurance reduction indicates that the partial programming mechanism does not bring noticeable extra physical damage to flash memory cells.

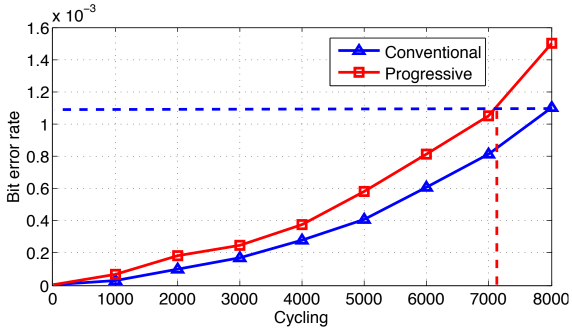


Figure 5: Comparison of the bit error rate of conventional programming and progressive partial programming.

3 Proposed Design Solution

Leveraging the partial programmability of SLC-mode flash memory, very intuitively we can deploy in-place delta compression, as illustrated in Figure 6, to eliminate the read latency penalty inherent to conventional design practice as described in Section 2.2. As shown in Figure 6, the original data content C_0 and all the subsequent deltas d_i’s are progressively programmed into a single physical page. Once the physical page is full after the k-th update, or the number of deltas reaches a threshold T (we don’t expect to accumulate too many deltas in case of a larger retrieval latency), we allocate a new physical page, write the latest version data C_{k+1} to the new physical page, and reset the delta compression for subsequent updates. This mechanism can guarantee that we only need to read a single flash memory page to retrieve the current data content.

In spite of the very simple basic concept, its practical implementation is subject to several non-trivial issues: (i) *Storage capacity utilization*: Suppose each flash memory page can store m (e.g., 4 or 8) 4kB sectors. The straightforward implementation of in-place delta compression explicitly reserves certain storage capacity within each SLC flash memory page for storing deltas. As a result, we can only store at most $m-1$ 4kB sectors per page at the very beginning. Due to the runtime

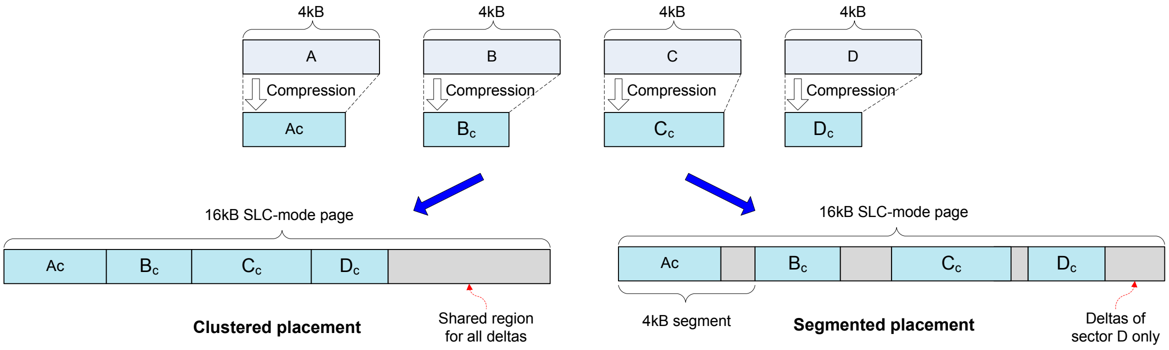


Figure 7: Illustration of opportunistic in-place delta compression and two different data placement strategies.

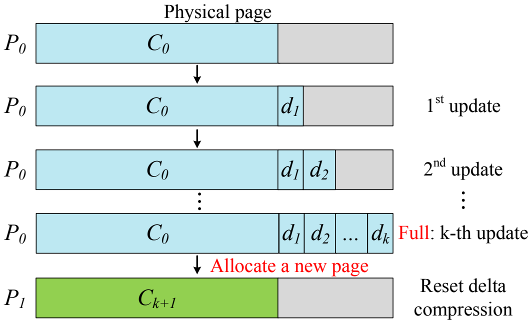


Figure 6: Illustration of the basic concept of in-place delta compression.

variation of the delta compressibility among all the data, these explicitly reserved storage space may not be highly utilized. This clearly results in storage capacity penalty. In addition, by changing the number of 4kB sectors per page, it may complicate the design of FTL. (ii) *Error correction*: All the data in flash memory must be protected by ECC. Due to the largely different size among the original data and all the deltas, the ECC must be devised differently. In particular, the widely used low-density parity-check (LDPC) codes are only suitable for protecting large data chunk size (e.g., 2kB or 4kB), while each delta can only be a few tens of bytes. In the remainder of this section, we present design techniques to address these issues and discuss the involved trade-offs.

3.1 Opportunistic In-place Delta Compression

To eliminate the storage capacity penalty, we propose to complement delta compression with intra-sector lossless data compression. In particular, we apply lossless data compression to each individual 4kB sector being written to an SLC-mode flash memory page, and opportunistically utilize the storage space left by compression for storing subsequent deltas. This is referred to as opportunistic in-place delta compression. This is illustrated in Figure 7, where we assume the flash memory page size is 16kB. Given four 4kB sectors denoted as A, B, C, and D, we first apply lossless data compression to each sector individually and obtain A_c, B_c, C_c, and D_c. As shown in Figure 7, we can place these four compressed sectors into a 16kB SLC-mode flash memory page in two different ways:

1. *Clustered placement*: All the four compressed sectors are stored consecutively, and the remaining space within the 16kB page can store any deltas associated with these four sectors.
2. *Segmented placement*: Each 16kB SLC-mode flash memory page is partitioned into four 4kB segments, and each segment is dedicated for storing one compressed sector and its subsequent deltas.

These two different placement strategies have different trade-offs between delta compression efficiency and read latency. For the clustered placement, the four sectors share a relatively large residual storage space for storing subsequent deltas. Hence, we may expect that more deltas can be accumulated within the same physical page, leading to a higher delta compression efficiency. However, since the storage of original content and deltas of all the four sectors are mixed together, we have to transfer the entire 16kB from flash memory to SSD controller in order to reconstruct the current version of any one sector, leading to a longer flash-to-controller data transfer latency. On the other hand, in the case of segmented placement, we only need to transfer a 4kB segment from flash memory to SSD controller to serve one read request. Meanwhile, since the deltas associated with each sector can only stored within one 4kB segment, leading to lower delta compression efficiency compared with the case of clustered placement. In addition, segmented placement tends to have lower computational complexity than clustered placement, which will be further elaborated later in Section 3.3.

3.2 Hybrid ECC and Data Structure

The above opportunistic in-place delta compression demands a careful design of data error correction and overall data structure. As illustrated in Figure 8, we must store three types of data elements: (1) compressed sector, (2) delta, and (3) header. Each compressed sector and delta follows one header that contains all the necessary metadata (e.g., element length and ECC configuration). Each element must be protected individually by one ECC codeword. In addition, each header should contain an unique marker to identify a valid header. Since all the unwritten memory cells have the value of 1, we can use an all-zero bit vector as the header marker.

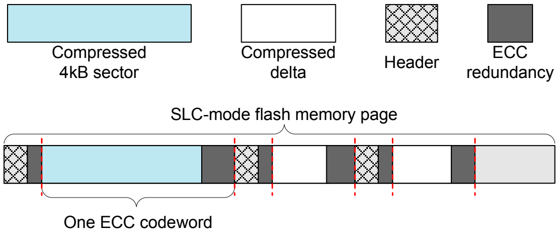


Figure 8: Illustration of three types of data elements, all of which must be protected by ECC.

Since all the elements have different different size, the ECC coding must natively support variable ECC codeword length, for which we can use the codeword puncturing [18]. Given an (n,k) ECC that protects k-bit user data with $(n-k)$-bit redundancy. If we want to use this ECC to protect m-bit user data $\mathbf{u}_m$ (where $m<k$), we first pad $(k-m)$-bit all-zero vector $\mathbf{O}_{k-m}$ to form a k-bit vector $[\mathbf{u}_m, \mathbf{O}_{k-m}]$. We encode the k-bit vector to generate $(n-k)$-bit $\mathbf{r}_{n-k}$ of redundancy, leading to an n-bit codeword $[\mathbf{u}_m, \mathbf{O}_{k-m}, \mathbf{r}_{n-k}]$. Then we remove the $(k-m)$-bit all-zero vector $\mathbf{O}_{k-m}$ from the codeword to form an $(n+m-k)$-bit *punctured* ECC codeword $[\mathbf{u}_m, \mathbf{r}_{n-k}]$, which is stored into flash memory. To read the data, we retrieve the noisy version of the codeword, denoted as $[\tilde{\mathbf{u}}_m, \tilde{\mathbf{r}}_{n-k}]$, and insert $(k-m)$-bit all-zero vector $\mathbf{O}_{k-m}$ back to form an n-bit vector $[\tilde{\mathbf{u}}_m, \mathbf{O}_{k-m}, \tilde{\mathbf{r}}_{n-k}]$, to which we apply ECC decoding to recover the user data $\mathbf{u}_m$.

In order to avoid wasting too much coding redundancy, the ratio of m/k in ECC puncturing should not be too small (i.e., we should not puncture too many bits). Hence, instead of using a single ECC, we should use multiple ECCs with different codeword length to accommodate the large variation of data element length. To protect relatively long data elements (in particular the compressed 4kB sectors), we can use three LDPC codes with different codeword length, denoted as $LDPC_{4kB}$, $LDPC_{2kB}$, and $LDPC_{1kB}$. The code $LDPC_{4kB}$ protects all the elements with the length bigger than 2kB, the code $LDPC_{2kB}$ protects all the elements with the length within 1kB and 2kB, and the code $LDPC_{1kB}$ protects all the elements with the length within 512B and 1kB. Thanks to recent work on versatile LDPC coding system design [19, 20], all the three LDPC codes can share the same silicon encoder and decoder, leading to negligible silicon penalty in support of multiple LDPC codes. Since LDPC codes can only work with relatively large codeword length (i.e., 1kB and beyond) due to the error floor issue [21], we have to use a set of BCH codes to protect all the elements with the length less than 512B. BCH codes with different codeword length are constructed under different Galois Fields, hence cannot share the same silicon encoder and decoder. In this work, we propose to use three different BCH codes, denoted as BCH_{4B}, BCH_{128B}, and BCH_{512B}, which can protect 4B, 128B, and 512B, respectively. We fix the size of element header as 4B, and the BCH_{4B} aims to protect each element header. The code BCH_{512B} protects all the elements with the length within 128B and 512B, and the code BCH_{128B} protects all the non-header elements with the length of less than 128B.

3.3 Overall Implementation

Based upon the above discussions, this subsection presents the overall implementation flow of the proposed opportunistic in-place delta compression design framework. Figure 9 shows the flow diagram for realizing delta compression to reduce write stress. Upon a request of writing 4kB sector C_k at a given LBA within the SLC-mode flash memory region, we retrieve and re-construct the current version of the data C_{k-1} from an SLC-mode physical page. Then we obtain the compressed delta between C_k and C_{k-1}, denoted as d_k. Accordingly we generate its header and apply ECC encoding to both the header and compressed delta d_k, which altogether form a bit-vector denoted as p_k. If there is enough space in this SLC-mode page and the number of existing deltas is smaller than the threshold T, we write p_k into the page through partial programming; otherwise we allocate a new physical page, compress the current version C_k and write it to this new page to reset the delta compression. In addition, if the original sector is not compressible, like video or photos, we simply write the original content to flash memory without adding a header. Meanwhile, we write a special marker bit to the reserved flash page metadata area [22]. During the read operation, if the controller detected the marker, it will know that this sector is written uncompressed.

The key operation in the process shown in Figure 9 is the data retrieval and reconstruction. As discussed in Section 3.1, we can use two different intra-page data placement strategies, i.e., clustered placement and segmented placement, for which the data retrieval

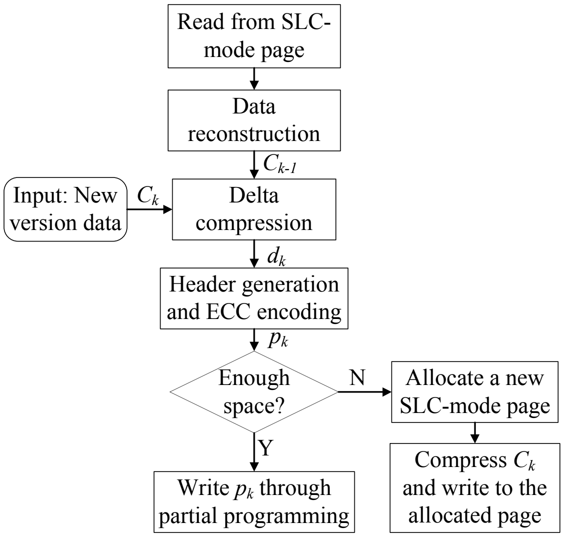

Read from SLC-mode page
Data reconstruction
C_{k-1}
Input: New version data
C_k
Delta compression
d_k
Header generation and ECC encoding
p_k
Enough space?
N
Allocate a new SLC-mode page
Y
Write p_k through partial programming
Compress C_k and write to the allocated page

Figure 9: Flow diagram for realizing delta compression.

and reconstruction operation involves different latency overhead and computational complexity. In short, compared with clustered placement, segmented placement has shorter latency and less computational complexity. This can be illustrated through the following example. Suppose a single 16kB flash page contains four compressed 4kB sectors, A_c, B_c, C_c, and D_c. Associated with each sector, there is one compressed delta, $d_{A,1}$, $d_{B,1}$, $d_{C,1}$, and $d_{D,1}$. Each of these eight data elements follows a header, hence we have total eight headers. Suppose we need to read the current content of sector B, the data retrieval and reconstruction process can be described as follows:

- In the case of clustered placement, the SSD controller must retrieve and scan the entire 16kB flash memory page. It must decode and analyze all the eight headers to determine whether to decode or skip the next data element (compressed sector or delta). During the process, it carries out further ECC decoding to obtain B_c and $d_{B,1}$, based upon which it performs decompression and accordingly reconstruct the current content of sector B.
- In the case of segmented placement, the SSD controller only retrieves and scans the second 4kB from from the 16kB flash memory page. As a result, it only decodes and analyzes two headers, and accordingly decodes and decompresses B_c and $d_{B,1}$, and finally reconstructs the current content of sector B.

From above simple example, it is clear that, compared with clustered placement, segmented placement largely reduces the amount of data being transferred from flash memory chips to SSD controller, and involves a fewer number of header ECC decoding. This leads to lower latency and less computation. On the other hand, clustered placement tends to have a better storage efficiency by allowing different sectors to share the same storage region for storing deltas.

Thus the proposed design solution essentially eliminates read amplification and filesystem/firmware design overhead, which are two fundamental drawbacks inherent to conventional practice. Meanwhile, by opportunistically exploiting lossless compressibility inherent to data content itself, this design solution does not incur a storage capacity penalty on the SLC-mode flash memory region in SSDs.

Based upon the above discussions, we may find that a noticeable write traffic reduction could be expected with a good compression efficiency and delta compression efficiency. So if the data content is not compressible (like multimedia data or encrypted data), the reduction would be limited. In addition, another application condition is that the proposed design solution favors update-in-place file system because only the write requests to the same LBA have a chance to be combined to the same physical page. Therefore, the proposed technique could not be very conveniently applied to some log-structured file system like F2FS, LFS because the in-place update is not inherently supported in the logging area of these file systems. And besides, the proposed design solution can be integrated with other appearing features of SSD such as encryption. SSDs are using high performance hardware modules to implement encryption. And the data/delta compression will not be affected if the encryption module is placed after compression.

4 Evaluations

This section presents our experimental and simulation results to quantitatively demonstrate the effectiveness and involved trade-offs of our proposed design solution.

4.1 Per-sector Compressibility

To evaluate the potential of compressing each original 4kB sector to opportunistically create space for deltas, we measured the per-4kB-sector data compressibility on different data types. We collected a large amount of 4kB data sectors from various database files, document files, and filesystem metadata. These types of data tend to be relatively hot and frequently updated, hence more likely reside in the SLC-mode region in SSDs.

We use the sample databases from [23, 24] to test the compressibility of MySQL database files. MySQL database uses pre-allocated data file, hence we ignored the unfilled data segments when we measured the compression ratio distribution. The Excel/Text datasets were collected from an internal experiment lab server. We used Linux Kernel 3.11.10 source [25] as the source code dataset. We collected the metadata (more than

34MB) of files in an ext4 partition as the metadata dataset. Figure 10 shows the compressibility of different data types with LZ77 compression algorithm. The *compression ratio* is defined as the ratio of the size after compression to before compression, thus a smaller ratio means a better compressibility. As shown in Figure 10, data compression ratio tends to follow a Gaussian-like distribution, while different datasets have largely different mean and variation. Because each delta tends to be much smaller than 4kB, the results show that the simple LZ77 compression is sufficient to leave enough storage space for storing multiple deltas.

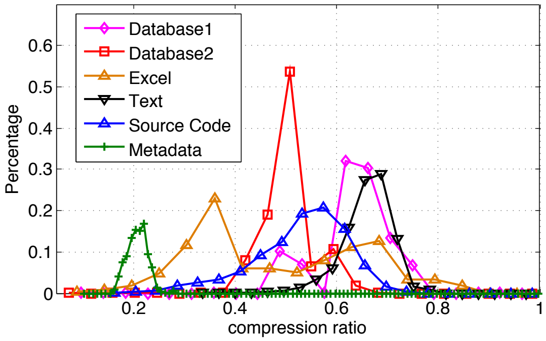


Figure 10: Compression ratio distribution of different data types with LZ77 compression.

4.2 Write Stress Reduction

We further evaluated the effectiveness of using the proposed opportunistic in-place delta compression to reduce the flash memory write stress. Clearly, the effectiveness heavily depends on the per-sector data compressibility and delta compressibility. Although per-sector data compressibility can be relatively easily obtained as shown in Section 4.1, empirical measurement of the delta compressibility is non-trivial. Due to the relative update regularity and controllability of filesystem metadata, we empirically measured the delta compressibility of metadata, based upon which we analyzed the write stress reduction for metadata. To cover the other types of data, we carried out analysis by assuming a range of Gaussian-like distributions of delta compressibility following prior work [10, 13].

4.2.1 A Special Case Study: Filesystem Metadata

To measure the metadata delta compressibility, we modified Mobibench [26] to make it work as the I/O workload benchmark under Linux Ubuntu 14.04 Desktop. We use a large set of SQLite workloads (`create, insert, update, delete`) and general filesystem tasks (file `read, update, append`) to trigger a large amount of file metadata updates. To monitor the characteristics of metadata, based upon the existing tool `debugfs` [27], we implemented a metadata analyzer tool [28] to track, extract, and analyze the filesystem metadata. We use an ext4 filesystem as the experimental environment and set the system page cache write back period as 500ms. Every time before we collect the file metadata, we wait for 1s to ensure that file metadata are flushed back to the storage device. For each workload, we collected 1000 consecutive versions of metadata.

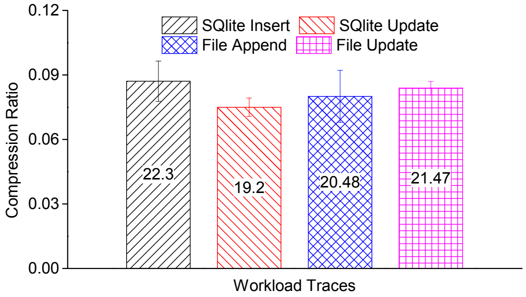


Figure 11: Delta compression ratio of consecutive versions of metadata for different workloads.

Based on the collected consecutive versions of metadata, we measured the delta compressibility as shown in Figure 11. The number inside the bar indicates the average number of bytes needed to store the difference between two consecutive versions of metadata, while the complete size of ext4 file metadata is 256 byte. The average delta compression ratio is 1:0.087 with the standard deviation of 0.0096. The results indicate that the delta compression ratio is quite stable with a very small deviation.

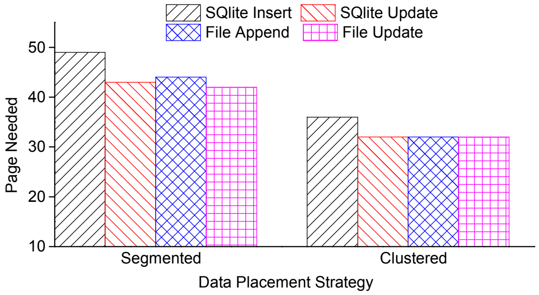


Figure 12: Number of flash memory pages being programmed for storing 1000 consecutive versions of metadata. (In comparison with conventional practice, we need at most 1000 pages to store these versions.)

The results in Figure 11 clearly suggest the significant data volume reduction potential by applying delta compression for metadata. To estimate the corresponding

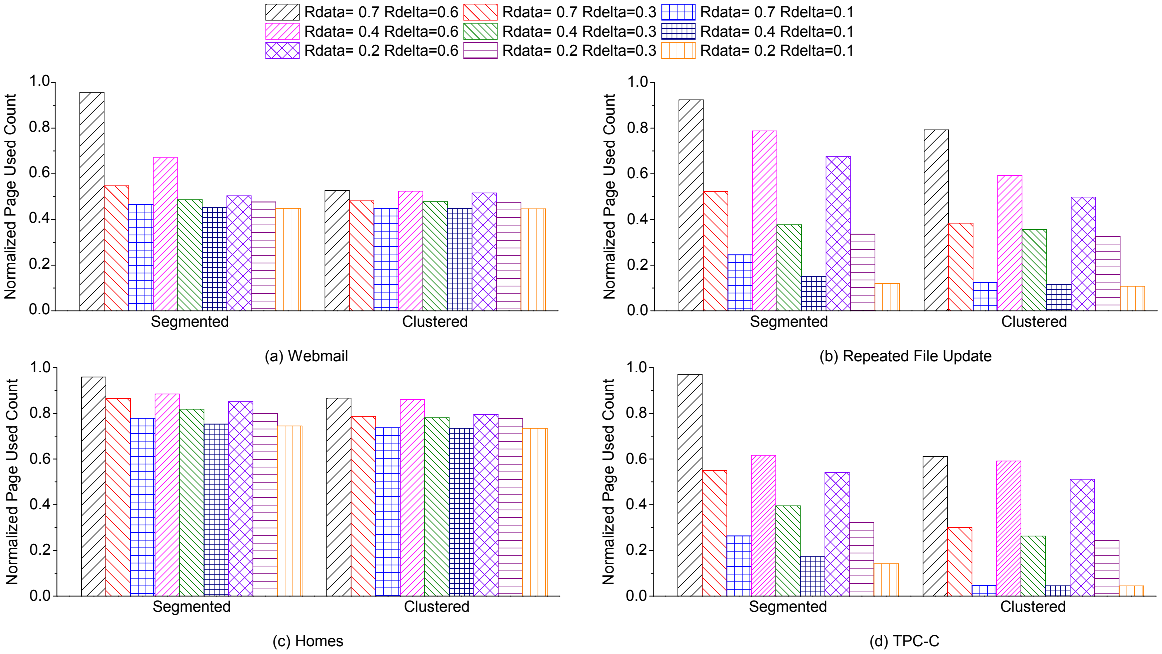


Figure 13: Reduction of the number of programmed flash memory pages under different workloads and over different data compressibility.

write stress reduction, we set that each SLC-mode flash memory page is 16kB and stores four compressed 4kB sectors and their deltas. Figure 12 shows the average number of flash memory pages that must be programmed in order to store 1000 consecutive versions of metadata pages. We considered the use of both segmented placement and clustered placement design strategies as presented in Section 3.1. Thanks to the very good per-sector compressibility and delta compressibility of metadata, the flash memory write stress can be reduced by over 20×. In addition, by allowing all the four sectors share the space for storing deltas, clustered placement can achieve higher write stress reduction than segmented placement, as shown in Figure 12.

4.2.2 Analytical Results for General Cases

Prior work [10, 13, 14] modeled delta compressibility to follow Gaussian-like distributions. To facilitate the evaluation over a broader range of data types, we follow this Gaussian distribution based model in these work as well. Let R_{data} denote the mean of the per-sector compression ratio of original data, and let R_{delta} denote the mean of delta compression ratio. Based upon the results shown in Section 4.1, we considered three different values of R_{data}, i.e., 0.2, 0.4, and 0.7. scenarios. According to prior work [10, 13, 14], we considered three different values of R_{delta}, i.e., 0.1, 0.3, and 0.6. Meanwhile, we set the value of deviation to 10% of the corresponding value of mean according to our measurements in Section 4.1.

In this section, we carried out simulations to estimate the flash memory write stress reduction over different workloads, and the results are shown in Figure 13. We chose the following four representative workloads:

- *Webmail Server*: We used Webmail Server block trace from [16], which was obtained from a department mail sever and the activities include mail editing, saving, backing up, etc.
- *Repeated File Update*: We enhanced the benchmark in [26] to generate a series of file updating in an Android Tablet, and accordingly captured the block IO traces.
- *Home*: We used the Homes Traces in [16], which include a research group activities of developing, testing, experiments, technical writing, plotting, etc.
- *Transaction*: We executed TPC-C benchmarks (10 warehouses) for transaction processing on MySQL 5.1 database system. We ran the benchmarks and use blktrace tool to obtain the corresponding traces.

As shown in Figure 13, the write stress can be noticeably reduced by using the proposed design solution (a smaller value in figure indicates a better stress reduction). In the "Repeated File Update" and TPC-C workloads, the number of programmed flash memory pages can be reduced by over 80%. The results clearly show that the flash memory write stress reduction is reversely proportional to R_{data} and R_{delta}, which can be intuitively justified. When both the original data and delta information cannot be compressed efficiently (such as R_{data} is 0.7 and R_{delta} is 0.6), the write stress can be hardly reduced because the compressed delta cannot be placed in the same page with the original data. However, with the clustered data placement strategy, some deltas could be placed because of a larger shared spare space. Thus the clustered data placement strategy has a better performance than the segmented approach in most of the cases, especially when the compression efficiency is relatively poor.

The write stress reduction varies among different workloads and strongly depends on the data update operation frequency. For example, with a large percentage of data updates than "Homes", "Repeated File Update" can achieve noticeably better write stress reduction as shown in Figure 13. In essence, there exists a upper bound of write stress reduction, which is proportional to the percentage of update operations. This explains why the write stress reduction cannot be further noticeably reduced even with better data compressibility, as shown in Figure 13.

4.3 Implementation Overhead Analysis

This subsection discusses and analyzes the overhead caused by the proposed design solution in terms of read latency, update latency, and SSD controller silicon cost.

4.3.1 Read Latency Overhead

Figure 14 illustrates the read process to recover the latest data content. After the flash memory sensing and flash-to-controller data transfer, the SSD controller parses the data elements and accordingly carries out the ECC decoding and data/delta decompression, based upon which it combines the original data and all the subsequent deltas to obtain the latest data content. As explained in Section 3.2, different segments are protected by different ECC codes (LDPC codes or BCH codes) according to the length of information bits. Hence the controller must contain several different ECC decoders.

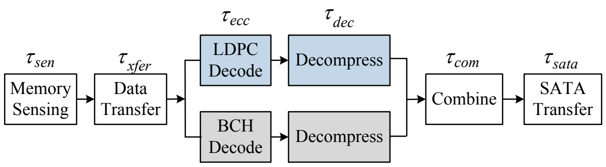


Figure 14: Illustration the process to obtain the latest data content.

Let τ_{sen} denote the flash memory sensing latency(the latency to read out the data content from flash cells using sensing circuits [29]), $\tau_{xfer}(\Omega)$ denote the latency of transferring Ω amount of data from flash memory chip to SSD controller, $\tau_{LDPC}^{(dec)}$ and $\tau_{BCH}^{(dec)}$ denote the LDPC and BCH decoding latency, $\tau_{sec}^{(dec)}$ and $\tau_{delta}^{(dec)}$ denote the latency of decompressing the original data and deltas, τ_{com} denote the latency to combine the original data and all the deltas to obtain the latest data content, and τ_{sata} denote the latency of transferring 4kB from SSD to host. In the conventional design practice without delta compression, to serve a single 4kB read request, the overall latency can be expressed as:

$$\tau_{read} = \tau_{sen} + \tau_{xfer}(4kB) + \tau_{LDPC}^{(dec)} + \tau_{sata}. \qquad (1)$$

When using the proposed design solution to realize delta compression, the read latency can be expressed as:

$$\begin{aligned} \tau_{read} =& \tau_{sen} + \tau_{xfer}(n \cdot 4kB) + \max(\tau_{LDPC}^{(dec)}, \tau_{BCH}^{(dec)}) \\ &+ \max(\tau_{sec}^{(dec)}, \tau_{delta}^{(dec)}) + \tau_{com} + \tau_{sata}, \end{aligned} \qquad (2)$$

where n denotes the number of 4kB sectors being transferred from flash memory chip to SSD controller. We have that $n = 1$ in the case of segmented placement, and n is the number of 4kB in each flash memory physical page in the case of clustered placement. Since there could be multiple elements that are decoded by the LDPC decoder or the same BCH decoder, $\tau_{LDPC}^{(dec)}$ and $\tau_{BCH}^{(dec)}$ in Eq. 2 are the aggregated LDPC and BCH decoding latency. In addition, $\tau_{delta}^{(dec)}$ in Eq. 2 is the aggregated delta decompression latency because there could be multiple deltas to be decompressed by the same decompression engine.

We can estimate the read latency based on the following configurations. The SLC-mode sensing latency τ_{sen} is about 40μs in sub-20nm NAND flash memory. We set the flash memory physical page size as 16kB. Under the latest ONFI 4.0 flash memory I/O specification with the throughput of 800MB/s, the transfer latency $\tau_{xfer}(4kB)$ is 5μs. We set the throughput of both LDPC and BCH decoding as 1GBps. Data decompression throughput is set as 500MBps, and delta decompression throughput is set as 4GBps due to its very simple operations. When combining the original data and all the deltas, we simply use parallel XOR operations and hence set τ_{com} as 1μs. Under the SATA 3.0 I/O specification with the throughput of 6Gbps, the SSD-to-host data transfer latency τ_{sata} is set as 5.3μs.

Based upon the above configurations, we have that, to serve a 4kB read request, the overall read latency is 54μs under the conventional practice without delta compression. When using the proposed design solution, the overall latency depends on the number of deltas involved in the read operation. With the two different data placement

Table 2: Read/Update latency overhead comparison of different cases.

Operation	Technique	Average-case (μs)	Worst-case (μs)
Read	Conventional	54	
	Clustered	76	102
	Segmented	56	63
Update	Conventional	186	
	Clustered	246	272
	Segmented	226	233

strategies, we estimate the *worst-case* and *average-case* read latency as shown in Table 2:

- *Clustered placement*: In this case, the flash-to-controller data transfer latency is $\tau_{xfer}(16kB)$=20μs. In the worse case, the compressed 4kB sector being requested and all its deltas almost completely occupy the entire 16kB flash memory physical page, and are all protected by the same ECC (LDPC or BCH). And the total information bit length will be nearly 32kB at most due to ECC code word puncturing (as explained in Section 3.2). As a result, the decoding latency is 32μs at most and delta decompression latency is 4μs. Hence, the overall worst-case read latency is 102μs, representing a 88% increase compared with the conventional practice. In the average case, the latency of decoding/decompressing the original 4kB sector is longer than that of its deltas. Assuming the original 4kB sector is compressed to 3kB, we can estimate the decoding and decompression latency as 4μs and 6μs. Hence, the overall average-case read latency is 76μs, representing a 41% increase compared with the conventional practice.
- *Segmented placement*: In this case, the flash-to-controller data transfer latency is $\tau_{xfer}(4kB)$=5μs. The worst-case scenario occurs when the data compressibility is low and hence the compressed sector is close to to 4kB, leading to the decoding and decompression latency of 4μs (using $LDPC_{4kB}$) and 8μs, respectively. Hence, the worst-case overall read latency is 63μs, representing a 17% increase compared with the conventional practice. Under the average case, the compression ratio is modest and multiple deltas are stored, for which the latency could be about 2$\sim$4μs. Hence the average-case overall latency is about 56μs, representing a 4% increase compared with conventional practice.

4.3.2 Update Latency Overhead

In conventional practice without using delta compression, a data update operation simply invokes a flash memory write operation. However, in our case, a data update operation invokes data read, delta compression, and flash memory page partial programming. Let τ_{read} denote the latency to read and reconstruct one 4kB sector data (as discussed in the above), $\tau_{delta}^{(enc)}$ denote the delta compression latency, and $\tau_{program}$ denote the latency of flash memory page partial programming. Hence the update latency can be expressed as:

$$\tau_{write} = \tau_{read} + \tau_{delta}^{(enc)} + \tau_{ecc}^{(enc)} + \tau_{xfer} + \tau_{program} \qquad (3)$$

Based upon our experiments with sub-20nm NAND flash memory, we set $\tau_{program}$ as of 150μs. We set the delta compression throughput $\tau_{delta}^{(enc)}$ as 4GBps and the ECC encoding throughput $\tau_{ecc}^{(enc)}$ as 1GBps. Therefore, the overall of writing one flash memory page is 186μs. When using the proposed design solution, as illustrated in Table 2, the value of τ_{read} could largely vary. In the case of clustered placement, the worst-case and average-case update latency is 272μs and 246μs, representing 32% and 46% increase compared with the conventional practice. In the case of segmented placement, the worst-case and average-case update latency is 233μs and 226μs, representing 25% and 22% increase compared with the conventional practice.

4.3.3 Silicon Cost

Finally, we evaluated the silicon cost overhead when using the proposed design solution. In particular, the SSD controller must integrate several new processing engines, including (1) multiple BCH code encoders/decoders, (2) per-sector lossless data compression and decompression engines, and (3) delta compression and decompression engines. As discussed in Section 3.2, we use three different BCH codes, BCH_{4B}, BCH_{128B}, and BCH_{512B}, which protect upto 4B, 128B, and 512B, respectively. Setting the worst-case SLC-mode flash memory bit error rate (BER) as 2×10^{-3} and the decoding failure rate as 10^{-15}, we constructed the code BCH_{4B} as the (102, 32) binary BCH code over GF(2^7), BCH_{128B} as the (1277, 1024) binary BCH code over GF(2^{11}), and BCH_{512B} as the (4642, 4096) binary BCH code over GF(2^{13}). To evaluate the entire BCH coding system silicon cost, we carried out HDL-based ASIC design using Synopsys synthesis tool set and results show that the entire BCH coding system occupies 0.24mm^2 of silicon area at the 22nm node, while achieving 1GBps throughput.

Regarding the per-sector lossless data compression and decompression, we chose the LZ77 compression algorithm [30], and designed the LZ77 compression and decompression engines with HDL-based design entry and Synopsys synthesis tool set. The results show that the LZ77 compression and decompression engine occupies 0.15mm^2 of silicon area at the 22nm node (memory costs included), while achieving 500MBps throughput. Regarding delta compression and decompression, since they mainly involve simple XOR and counting operations, it is reasonable to expect that their silicon implementation cost is negligible compared with BCH coding and LZ77 compression. Therefore, we estimate that the overall silicon cost for implementing the proposed design solution is 0.39mm^2 at the 22nm node. According to our knowledge, the LDPC decoder module accounts for up to 10% of a typical SSD controller, meanwhile our silicon cost (including the logical resources such as gates, registers, memory, etc) is about 1/3 of an LDPC decoder. Therefore, we can estimate that the involved silicon area in proposed solution will occupy less than 5% of the silicon area of an SSD controller, which is a relatively small cost compared to the entire SSD controller.

5 Related Work

Aiming to detect the data content similarity and store the compressed difference, delta compression has been well studied in the open literature. Dropbox [31] and Github use delta compression to reduce the network bandwidth and storage workload using a pure application software level solution. Design solutions in [10, 11, 13] reduce the waste of space by detecting and eliminating the duplicate content in block device level while the proposed solution could further reduce the redundancy of similar but not identical writes. The FTL-level approach presented in [14] stores the compressed deltas to a temporary buffer and commits them together to the flash memory when the buffer is full, thus the number of writes could be reduced. Authors of [32] proposed a design solution to extend the NAND flash lifetime by detecting the identical writes. Authors of [33] developed an approach to utilize the content similarity to improve the IO performance while the proposed techniques pay more attention on the write stress reduction to extend the SSD lifetime. To improve the performance of data backup workloads in disks, authors of [9] proposed an approach to implement delta compression on top of deduplication to further eliminate redundancy among similar data. The key difference between proposed solution and existing solutions is that we can make sure the deltas and original data content locate in the same physical flash memory page, which will eliminate the read latency overhead fundamentally.

General-purpose lossless data compression also has been widely studied in flash-based storage system. The authors of [34, 35] presented a solution to realize transparent compression at the block layer to improve the space efficiency of SSD based cache. A mathematic framework to estimate how data compression can improve NAND flash memory lifetime is presented in [12]. The authors of [36] proposed to integrate database compression and flash-aware FTL to effectively support database compression on SSDs. The authors of [37] evaluated several existing compression solutions and compared their performance. Different from all the prior work, we for the first time present a design solution that cohesively exploits data compressibility and SLC-mode flash memory page partial-programmability to implement delta compression at minimal read latency and data management overhead.

6 Conclusion

In this paper, we present a simple design solution to most effectively reduce the write stress on SLC-mode region inside modern SSDs. The key is to leverage the fact that SLC-mode flash memory pages can naturally support partial programming, which makes it possible to use intra-page delta compression to reduce write stress without incurring significant read latency and data management complexity penalties. To further eliminate the impact on storage capacity, we combine intra-page delta compression with intra-sector lossless data compression, leading to the opportunistic in-place delta compression. Its effectiveness has been well demonstrated through experiments and simulations.

Acknowledgments

We would like to thank our shepherd Michael M. Swift and the anonymous reviewers for their insight and suggestions that help us to improve the quality and presentation of this paper. This work was supported by the National Science Foundation under Grants No. ECCS-1406154.

References

[1] X. Jimenez, D. Novo, and P. Ienne, “Libra: Software-controlled cell bit-density to balance wear in NAND Flash,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 2, pp. 28:1–28:22, Feb. 2015.

[2] Y. Oh, E. Lee, J. Choi, D. Lee, and S. Noh, “Hybrid solid state drives for improved performance and enhanced lifetime,” in *IEEE Symposium on Mass Storage Systems and Technologies (MSST’13)*, May 2013, pp. 1–5.

[3] L.-P. Chang, “A hybrid approach to NAND-Flash-based solid-state disks,” *IEEE Transactions on*

Computers, vol. 59, no. 10, pp. 1337–1349, Oct 2010.

[4] D. Sharma, "System design for mainstream TLC SSD," in *Proc. of Flash Memory Summit*, Aug. 2014.

[5] *Micron's M600 SSD accelerates writes with dynamic SLC cache*. http://techreport.com/news/27056/micron-m600-ssd-accelerates-writes-with-dynamic-slc-cache, Sept., 2014.

[6] *Samsung's 840 EVO solid-state drive reviewed TLC NAND with a shot of SLC cache*. http://techreport.com/review/25122/samsung-840-evo-solid-state-drive-reviewed, July, 2013.

[7] Y. Oh, E. Lee, J. Choi, D. Lee, and S. H. Noh, "Efficient use of low cost SSDs for cost effective solid state caches," *Science and Technology*, vol. 2010, p. 0025282.

[8] U. Manber, S. Wu *et al.*, "Glimpse: A tool to search through entire file systems." in *Usenix Winter*, 1994, pp. 23–32.

[9] P. Shilane, G. Wallace, M. Huang, and W. Hsu, "Delta compressed and deduplicated storage using stream-informed locality," in *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems (HotStorage'12)*, Berkeley, CA, 2012.

[10] C. B. Morrey III and D. Grunwald, "Peabody: The time travelling disk," in *Proceedings of 20th IEEE Conference on Mass Storage Systems and Technologies (MSST'03)*. IEEE, 2003, pp. 241–253.

[11] Q. Yang and J. Ren, "I-CASH: Intelligently coupled array of ssd and hdd," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*, Feb 2011, pp. 278–289.

[12] J. Li, K. Zhao, X. Zhang, J. Ma, M. Zhao, and T. Zhang, "How much can data compressibility help to improve nand flash memory lifetime?" in *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST'15)*, Santa Clara, CA, Feb. 2015, pp. 227–240.

[13] Q. Yang, W. Xiao, and J. Ren, "Trap-array: A disk array architecture providing timely recovery to any point-in-time," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, 2006, pp. 289–301.

[14] G. Wu and X. He, "Delta-FTL: Improving SSD lifetime via exploiting content locality," in *Proceedings of the 7th European conference on Computer systems (Eurosys'12)*, New York, NY, USA, 2012, pp. 253–266.

[15] UMass Trace Repository. http://traces.cs.umass.edu/index.php/Storage/Storage/.

[16] FIU IODedup Trace-Home. http://iotta.snia.org/traces/391.

[17] D. Campello, H. Lopez, R. Koller, R. Rangaswami, and L. Useche, "Non-blocking writes to files," in *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST'15)*, Santa Clara, CA, Feb. 2015, pp. 151–165.

[18] S. Lin, T. Kasami, T. Fujiwara, and M. Fossorier, *Trellises and Trellis-Based Decoding Algorithms for Linear Block Codes*. Kluwer Academic Publishers, 1998.

[19] Z. Wang, Z. Cui, and J. Sha, "VLSI design for low-density parity-check code decoding," *IEEE Circuits and Systems Magazine*, vol. 11, no. 1, pp. 52–69, 2011.

[20] C. Fewer, M. Flanagan, and A. Fagan, "A versatile variable rate LDPC codec architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 10, pp. 2240–2251, Oct 2007.

[21] T. Richardson, "Error floors of LDPC codes," in *Proc. of 41st Allerton Conf. Communications, Control and Computing*, Oct. 2003.

[22] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proceedings of 11th USENIX Conference on File and Storage Technologies (FAST'13)*, San Jose, CA, 2013, pp. 257–270.

[23] Employees Sample Database. https://dev.mysql.com/doc/employee/en/index.html.

[24] BIRT Sample Database. http://www.eclipse.org/birt/documentation/sample-database.php.

[25] Linux Kernel 3.11.10. https://www.kernel.org/pub/linux/kernel/v3.x/.

[26] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," in *USENIX Annual Technical Conference (ATC'13)*, San Jose, CA, 2013, pp. 309–320.

[27] T. Tso, "Debugfs." [Online]. Available: http://linux.die.net/man/8/debugfs

[28] X. Zhang, J. Li, K. Zhao, H. Wang, and T. Zhang, "Leveraging progressive programmability of SLC flash pages to realize zero-overhead delta compression for metadata storage," in *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'15)*, Santa Clara, CA, Jul. 2015.

[29] R. Micheloni, L. Crippa, and A. Marelli, *Inside NAND Flash Memories*. Dordrecht: Springer, 2010.

[30] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transaction on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[31] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: understanding personal cloud storage services," in *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM, 2012, pp. 481–494.

[32] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives." in *Proc. of USENIX Conference on File and Storage Technologies (FAST'11)*, vol. 11, 2011.

[33] R. Koller and R. Rangaswami, "I/O deduplication: Utilizing content similarity to improve I/O performance," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, p. 13, 2010.

[34] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas, "Using transparent compression to improve SSD-based I/O caches," in *Proceedings of the 5th European conference on Computer systems (Eurosys'10)*. ACM, 2010, pp. 1–14.

[35] T. Makatos, Y. Klonatos, M. Marazakis, M. Flouris, and A. Bilas, "ZBD: Using transparent compression at the block level to increase storage space efficiency," in *2010 International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI'10)*, May 2010, pp. 61–70.

[36] D. Das, D. Arteaga, N. Talagala, T. Mathiasen, and J. Lindström, "NVM compression—hybrid flash-aware application level compression," in *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*, Oct. 2014.

[37] A. Zuck, S. Toledo, D. Sotnikov, and D. Harnik, "Compression and SSDs: Where and how?" in *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*, Oct. 2014.

Access Characteristic Guided Read and Write Cost Regulation for Performance Improvement on Flash Memory

Qiao Li[§], Liang Shi[§*], Chun Jason Xue[‡]

Kaijie Wu[§], Cheng Ji[‡], Qingfeng Zhuge[§], and Edwin H.-M. Sha[§]

[§]*College of Computer Science, Chongqing University*

[‡]*Department of Computer Science, City University of Hong Kong*

∗*Corresponding author: shi.liang.hk@gmail.com*

Abstract

The relatively high cost of write operations has become the performance bottleneck of flash memory. Write cost refers to the time needed to program a flash page using incremental-step pulse programming (ISPP), while read cost refers to the time needed to sense and transfer a page from the storage. If a flash page is written with a higher cost by using a finer step size during the ISPP process, it can be read with a relatively low cost due to the time saved in sensing and transferring, and vice versa.

We introduce AGCR, an access characteristic guided cost regulation scheme that exploits this tradeoff to improve flash performance. Based on workload characteristics, logical pages receiving more reads will be written using a finer step size so that their read cost is reduced. Similarly, logical pages receiving more writes will be written using a coarser step size so that their write cost is reduced. Our evaluation shows that AGCR incurs negligible overhead, while improving performance by 15% on average, compared to previous approaches.

1 Introduction

NAND flash memory sees an increasing deployment in embedded systems, personal computers, mobile devices and servers over the last decades due to its advantages, such as light weight, high performance, and small form factors [1][2]. Although flash-based storage outperforms most magnetic-based storage, the costs of write and read operations are still the performance bottleneck of current systems. The write operation of flash memory is performed by the incremental-step pulse programming (ISPP) [3] scheme. ISPP is designed to iteratively increase the program voltage and use a small verification voltage to reliably program flash cells to their specified voltages. In each iteration, the program voltage is increased by a predefined program step size. Generally, write cost is much higher than read cost, usually by 10 to 20 times, and has been identified as a major performance bottleneck [4][5]. At the same time, read cost has also increased significantly with the increasing bit density and technology scaling [6][7][8].

The following tradeoffs of read and write costs have been identified [9][10][11][12][13]. First, read cost highly depends on the maximum raw bit error rate (RBER) of data pages and the deployed error correcting code (ECC). For a specific ECC, the lower the maximum RBER, the lower the read cost [6][7][14]. Second, write cost can be reduced by increasing the program step sizes in the ISPP process, at the cost of increasing the maximum RBER of the programmed pages. This in turn reduces the maximum retention time [9][10] and increases the read cost of the programmed page [13].

Previous approaches differ in the strategies used to select the data pages that will be written with different write costs. Pan et al. [9] and Liu et al. [10] proposed to apply low-cost writes (i.e., using coarser step sizes) to the data pages through retention time relaxation. Their approach was motivated by the significantly short lifetime of most data in several workloads compared to the predefined retention time. On the other hand, a high-cost write reduces the cost of the following reads to the same page [6][14]. Based on this characteristic, Li et al. [13] proposed to apply low-cost writes when there are several queued requests, and high-cost writes otherwise. Wu et al. [12] proposed to apply high-cost writes when, according to their estimation, the next access operation will not be delayed due to the increased cost. However, none of these works exploit the access characteristics of workloads.

This work is the first to exploit the access characteristics of workloads for cost regulation. We define three page access characteristics: If almost all the accesses to a data page are read (write) requests, the page is characterized as *read-only* (*write-only*). If the accesses to a data page are interleaved with reads and writes, the page is characterized as *interleaved-access*. *Our approach is based on the observation that most accesses* *from the host are performed on either read-only or write-only pages.* We exploit this observation to regulate access costs: for accesses identified as read-only, low-cost reads are preferred; for accesses identified as write-only, low-cost writes are preferred. The proposed approach chooses low-density parity code (LPDC) as the ECC, which is the best candidate for the advanced flash memory storage systems [6][14]. For the pages that are written with low cost, the required retention time can still be maintained as long as the resulting error rates do not exceed the error correction capability of the deployed LPDC code. Thus, read and write costs can be regulated to significantly improve overall performance. Our main contributions are as follows:

- We propose AGCR, a comprehensive approach to regulate the cost of writes and reads;
- We present a preliminary study to show the potential performance improvement of AGCR;
- We present an efficient implementation of AGCR with negligible overhead. Experimental results on a trace driven simulator [15][16] with 12 workloads [17] show that AGCR achieves significant performance improvement.

The rest of the paper is organized as follows. Section 2 presents the motivation and problem statement. Section 3 presents our proposed approach. Experiments and analysis are presented in Section 4. Section 5 concludes this work.

2 Motivation and Preliminary Study

2.1 Read and Write in Flash Memory

Write operations are the performance bottleneck of flash memory, not only because they require more time than read operations, but also because they block waiting operations. Moreover, the cost of write operations implicitly determines the cost of subsequent reads. Flash technology uses incremental-step pulse programming (ISPP) to program pages [3]. ISPP is designed to reliably program flash cells to their specific voltage levels using an iterative program-verify algorithm. The iterative algorithm has two stages in each iteration step: it first programs a flash cell with an incremental program voltage, and then verifies the voltage of the cell. If the voltage is lower than the predefined threshold, the process continues. In each iteration, the program voltage increases by a step size of ΔV_{pp}. The step size ΔV_{pp} determines the write cost – a coarser step size indicates a smaller number of steps (hence a lower write cost), but results in higher raw bit error rate (RBER) in the programmed page, and vice versa.

Low-density parity code (LDPC) is deployed as the default ECC for most advanced flash memory technologies [14]. For a high RBER, LDPC requires fine-grained memory-cell sensing, which is realized by comparing a series of N reference voltages. The error correction capability of LDPC increases with N, but a larger N results in a higher read cost.

Thus, there is a strong relationship between ISPP and LDPC on the read and write cost of flash memory. With a large step size in the ISPP process, the write cost is reduced but the RBER and read cost increase, and vice versa. The step size also affects the retention time of flash pages. A longer retention time is expected for pages written with finer step sizes [9][10]. In this work, we focus on read and write cost interaction by tuning the step size, where the tuning is constrained by the minimum required retention time [18].

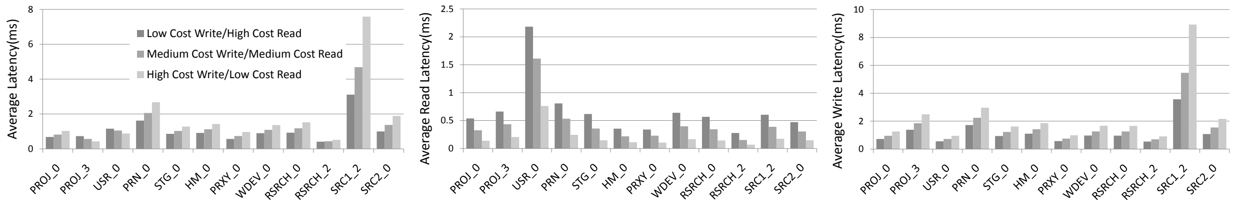
2.2 Read and Write Cost Regulator

Several strategies have been recently proposed for exploiting the reliability characteristics of flash memory to regulate read and write costs. Pan et al. [9] and Liu et al. [10] proposed to reduce write costs by relaxing the retention time requirement of the programmed pages. Wu et al. [12] proposed to apply a high-cost write to reduce the cost of read requests performed on the same page. The high-cost write is performed on the premise that upcoming requests will not be delayed by the increased cost. Li et al. [13] proposed to apply low-cost writes when there are queued requests to reduce the queueing delay, and apply high-cost writes otherwise, allowing low-cost reads of the page. However, none of these works exploit the access characteristics for cost regulation. Several types of workload access characteristics have been studied in previous works, including hot and cold accesses [19], inter-reference gaps [20], and temporal and spacial locality [21][22][23]. These access characteristics have been employed to guide the design of buffer caches [20][21][22] and flash translation layers (FTL) [19][23]. However, these characteristics imply the access frequency, which cannot be exploited for read and write cost regulation.

In this work, we applied the cost regulator proposed in Li et al. [13], which has been validated by simulation in previous studies [14][12][24]. The regulator consists of a write cost regulator and a read cost regulator. The write cost is inversely proportional to the program step size of ISPP [3][9][10]. Thus, the write cost regulator is defined as:

$$\begin{aligned} WC(\Delta V_{pp}) &= \gamma \times \frac{1}{\Delta V_{pp}} \\ \textit{where} \quad RBER(\Delta V_{pp}) &< CBER_{LDPC}(N); \end{aligned} \tag{1}$$

$WC(\Delta V_{pp})$ denotes the write cost when the program step size is ΔV_{pp}, and γ is a variable. A coarser step size results in a lower write cost, but a higher RBER. In addition, the reduction of the write cost is limited by the



(a) Overall average access latency (b) Average access latency for read requests (c) Average access latency for write requests

Figure 1: IO latency comparison for read and write requests with different access costs.

condition that the resulting RBER should be within the error correction capability of the deployed LDPC code, $CBER_{LDPC}(N)$, with N reference voltages.

The read cost is composed of the time to sense the page, which is proportional to N, and the time to transfer the data from the page to the controller, which is proportional to the size of the transferred information [6][7][14]. Thus, the read cost regulator is defined as:

$$\begin{aligned} & RC(N) = \alpha \times N + \beta \times \lceil \log(N+1) \rceil \\ where \quad & RBER(\Delta V_{pp}) < CBER_{LDPC}(N); \end{aligned} \tag{2}$$

where $RC(N)$ denotes the read cost with N reference voltages in the deployed LDPC code, and α and β are two variables. With a larger N, longer on-chip memory sensing time and longer flash-to-controller data transfer time are required, which lead to increased read cost.

Based on the two parts of the regulator, a low-cost write leaves the page with a higher RBER, thereby requiring an LDPC with more reference voltages, which further results in higher-cost reads. On the other hand, a high-cost write leaves the page with a lower RBER, thereby requiring an LDPC with less reference voltages, which further results in lower-cost reads. Note that the regulator ensures that the retention time requirement is always satisfied when writing data using different costs.

2.3 Preliminary Study

In the preliminary study, we evaluate three combinations of read and write costs: 1) All the writes are performed with a high cost, followed by low-cost reads (HCW/LCR); 2) All the writes are performed with a medium cost, followed by medium-cost reads (MCW/MCR); 3) All the writes are performed with a low cost, followed by high-cost reads (LCW/HCR). The detailed settings for the regulator can be found in Section 4.1.

Figure 1 presents the comparison of access latency for 12 representative traces of the enterprise servers from Microsoft Research (MSR) Cambridge [17]. Compared to the default MCW/MCR, HCW/LCR improves read performance by 54%, and LCW/HCR improves write performance by 26%. The significant performance improvement comes from the corresponding reduction in access costs. Comparing LCW/HCR and HCW/LCR, the differences in their read and write latencies are 114% and 61%, respectively. The performance gap indicates that the read and write cost regulator should be applied carefully. While LCW/HCR is able to improve the overall performance, it introduces the worst read performance, as shown Figure 1(b). However, read operations are always in the critical path, which motivates our approach.

3 AGCR: Access Characteristic Guided Cost Regulation

In this section, we propose AGCR, an access characteristic guided cost regulation scheme. We first present the study on the access characteristics of several workloads. Then, based on the observations from this study, we propose a comprehensive approach that includes a high-accuracy access characteristic identification method and a guided cost regulation scheme. Finally, we present the implementation and the overhead analysis.

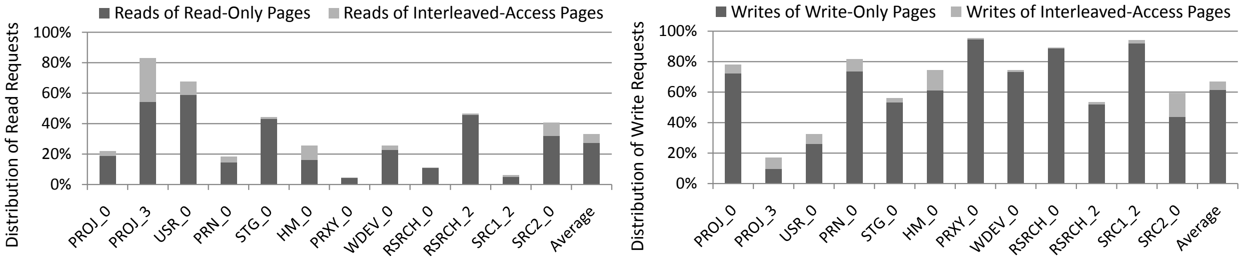
3.1 Access Characteristics of Workloads

In order to guide the read and write cost regulation, the access characteristics of workloads are very important. For example, if the accesses to a data page (logical page at the host system) are dominated by read requests, accessing the data page with low-cost reads can significantly improve performance.

Figure 2 presents the statistical results for the workloads analyzed in our preliminary study. We collected access characteristics for all data pages at the host system and distinguished between three types of data accesses:

1. Read-only: If almost all the accesses (>95%) to a data page are read requests, we characterize this page as read-only. This is typical when accessing media files or other read-only files;
2. Write-only: If almost all the accesses (>95%) to a data page are write requests, we characterize this page as write-only. This is typical in periodical data flushes from the memory system to storage for consistency maintenance;
3. Interleaved-access: If the accesses to a data page are interleaved with reads and writes, we characterize this page as interleaved-access.

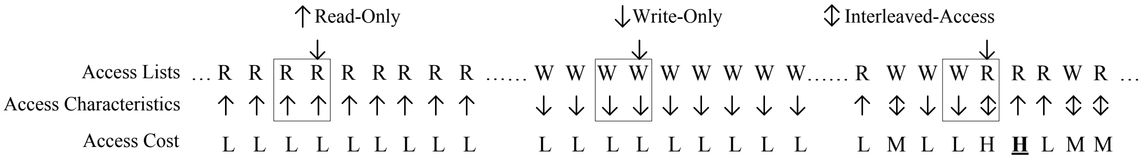
Figure 2 shows the request distributions for these logical data page access characteristics. Figure 2(a) shows the



(a) Distribution of read requests

(b) Distribution of write requests

Figure 2: Distribution of read and write requests on the three access characteristics.


↑ Read-Only
↓ Write-Only
⇕ Interleaved-Access
Access Lists … R R R R R R R R R …… W W W W W W W W W …… R W W W R R R W R …
Access Characteristics ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↑ ⇕ ↓ ↓ ⇕ ↑ ↑ ⇕ ⇕
Access Cost L L L L L L L L L L L L L L L L L L L M L L H H L M M

(a) Case 1

(b) Case 2

(c) Case 3

Figure 3: Example on the access characteristic identification and cost regulation: access characteristic, ↑ – read-only, ↓ – write-only, and ⇕ – interleaved-access; access cost, `L` – low-cost, `M` – medium-cost, and `H` – high-cost.

distribution of read requests of read-only pages and of interleaved-access pages. Similarly, Figure 2(b) shows the distribution of write request of write-only pages and of interleaved-access pages. For each trace, the accesses from both Figure 2(a) and 2(b) sum to 100%. We make three observations from Figure 2:

- Observation 1 – Most read requests access read-only pages, more than 85% on average;
- Observation 2 – Most write requests access write-only pages, more than 91% on average;
- Observation 3 – Only a small part of all requests access interleaved-access pages.

These observations guide the design of our read and write cost regulation. We first present a simple, yet accurate, access characteristic identification method.

3.2 Access Characteristic Identification

We identify the access characteristic of a data page based on its most recent requests and the upcoming request which are recorded using a history window. If the most recent requests and the upcoming request are all reads (writes), the page is characterized as read-only (write-only). Otherwise, the page is characterized as interleaved-access. Essentially, the identification method looks for consecutive reads or writes to characterize the page.

Figure 3 shows examples of the identification method. The first line shows the access lists. Case 1 has read requests only, case 2 has write requests only, and case 3 has interleaved read and write requests. The length of the history window is set to 2 in the examples. Thus it includes the upcoming request (indicated by the arrow above the window) and the most recent request to this page. The arrows below the access lists represent the identified access type of the page. As can be seen, the identification is updated on the arrival of each access to the page. If the access characteristic of the page is indeed read-only or write-only, the identification is accurate and stable, as shown in Figure 3(a) and 3(b). However, if the access characteristic of the page is interleaved, such as in Figure 3(c), the identification is unstable and may change from time to time.

This simple identification method works well, because, as observed above, most accesses are either *write-only* or *read-only*. The accuracy of the identification method is further evaluated in Section 3.4, where we analyze the effects of the size of the history window.

3.3 Access Cost Regulation

Based on the identified access characteristics of a page, the cost of its upcoming requests can be regulated. The main idea is to apply low-cost writes for write-only pages, low-cost reads (enabled by applying high-cost write), for read-only pages, and medium-cost accesses for interleaved-access pages.

Write cost regulation. The cost of a write is regulated when it is issued. As shown in Figure 3, there are two cases for write cost regulation. Upon arrival of a write request, the page is characterized and its cost is regulated as follows:

- A *write-only* page will be written with a low-cost write to improve write performance;
- An *interleaved-access* page , as in Figure 3(c), will be written with a medium-cost write to avoid read performance impact.

When a page is written for the first time, a high-cost write is performed.

Read cost regulation. A low-cost read can only be issued if the page was written with a high-cost write. Thus, a high-cost write will be *inserted* to the list of reads to re-write the page before upcoming reads, if the page was not written with a high cost before. As shown in Figure 3, there are three cases of read cost regulation. Upon arrival of a read request, the page is characterized, and its cost is regulated as follows:

- For a *read-only* page with a low-cost read, nothing should be done;
- For an *interleaved-access* page, the read cost is not regulated;
- For a *read-only* page with a high-cost read, a high-cost re-write operation is inserted to the re-write queue and performed during idle time to reduce the cost of upcoming reads (denoted with **<u>H</u>** in Figure 3(c)).

3.4 Implementation and Overheads

Figure 4 shows the implementation of AGCR in the flash memory controller. We add three new components: Access Characteristic Identification, Cost Regulator and Re-Write Queue. In addition, each mapping entry in the FTL is extended with two fields, as shown in Figure 5. The first is the access history, and the second is a 1-bit low-cost write tag. When the low-cost write tag is set for a read-only page, this indicates that the page must be re-written. When an I/O request is issued, the page is characterized, and its cost is regulated according to the rules in Section 3.3. If a re-write operation is needed for read cost regulation, the logical address of the data page is added to the re-write queue. During idle time, re-write operations are triggered to program the data in the queue with a high cost, which guarantees low-cost reads on these read-only pages. The re-write overhead is evaluated in the experiments. This implementation incurs three types of overheads: storage, hardware and firmware overhead. The storage overhead includes, for each page, a number of bits for access history and one bit for the low-cost write tag. Assuming access history is set to one bit, which records only the most recent request to the page, the storage overhead is 4MB for a 64GB flash memory with 4KB pages. The hardware overhead includes the voltage thresholds needed to support the three sets of reads and writes with different costs. This overhead is negligible according to previous studies [10][25]. The firmware overhead includes the processes involved in access characterization, cost regulation and re-write queuing. The overhead of these simple processes is negligible. AGCR does not introduce reliability issues thanks to the constraints in the read and write cost regulators. In addition, the energy consumption of the additional components is negligible.

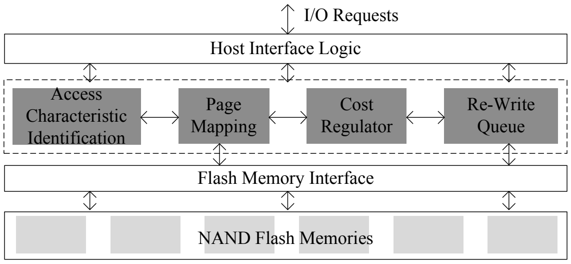


Figure 4: The implementation of the proposed approach.

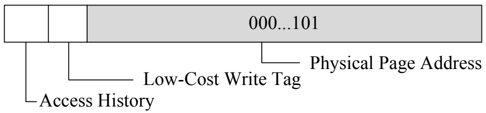


Figure 5: An example on the FTL mapping entry.

4 Evaluation

4.1 Experimental Setup

Table 1: The access cost configurations.

Read Cost (μs)	Low	Medium	High
	70	**170**	**310**
Write Cost (μs)	High	Medium	Low
	800	**600**	**450**
Erase Cost (μs)	3000	3000	3000

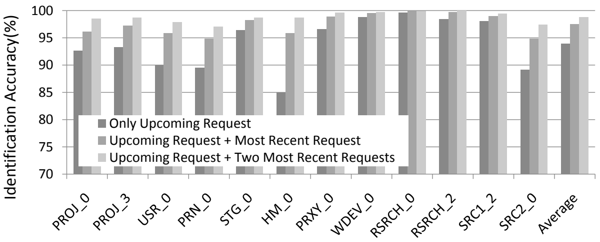


Figure 6: Window size impact on identification accuracy.

We use SSDsim [15] and workloads from MSR [17] to evaluate our cost-regulation approach. From our studies, we found that all workloads in MSR have similar access characteristics. We select the 12 representative traces from Figure 2 for our evaluation. The simulated storage system is configured with two bits per cell MLC flash memory. It has eight channels, with eight chips per channel and four planes per chip. Each chip has 2048 blocks and each block has 64 4KB pages. Default page mapping based FTL, garbage collection, and wear leveling are implemented in the simulator, representing

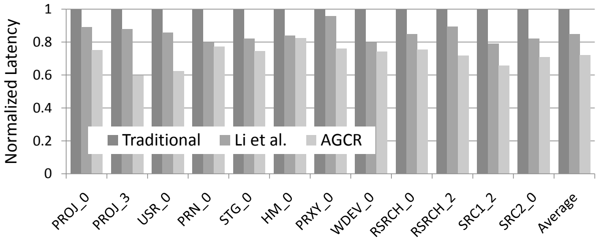
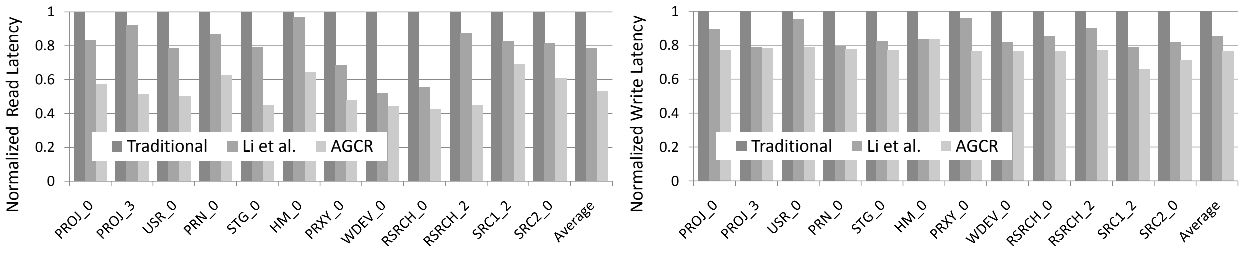


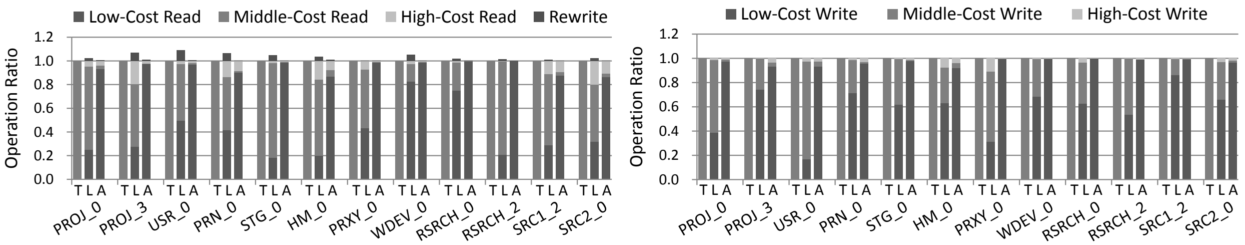
Figure 7: Overall performance comparison.



(a) IO latency for read requests

(b) IO latency for write requests

Figure 8: Normalized access latency compared with traditional case and Li et al.[13].



(a) Operation Ratios for read requests and Re-write Operations

(b) Operation Ratios for write requests

Figure 9: Distributions of different cost operations: T–Traditional, L–Li et al., and A–AGCR.

state-of-the-art storage systems [16]. The internal read and write operations for these mechanisms are processed with the regulated access costs. For fair comparison, we use the parameters for the cost regulator from Li et al. [13]. Table 1 shows the access costs. The cost of the erase operation does not depend on the write cost.

One of the most important parameters is the size of history window in the identification method. It affects the storage overhead as well as the identification accuracy. Figure 6 shows the identification accuracy for three window sizes. The results show that a larger window results in a higher accuracy. However, the increase in accuracy decreases as the size increases. In the following experiments, we consider only the most recent request and the upcoming request for identification to trade-off the storage overhead and identification accuracy.

4.2 Experimental Results

In this section, read, write and overall performance of AGCR are evaluated and compared with the traditional case and the state-of-the-art work from Li et al. [13]. In the traditional case, all reads and writes are performed with medium-cost, while Li et al. used low-cost reads or writes to reduce queueing delay. Cost regulation highly depends on the access history in AGCR. If there is no history for the accessed page, a high-cost write is used for write requests and a low-cost read is used for read requests. Figures 7 and 8 show the normalized access latency compared to the traditional, non-regulated costs (MCW/MCR in Figure 1) and to Li et al. Figure 7 shows that, compared to Li et al., AGCR achieves the best overall performance improvement, 15%, on average. In addition, as shown in Figures 8(a) and 8(b), AGCR achieves up to 48.3% and 20.4% latency improvement for reads and writes, respectively, over Li et al.

To understand the performance variations, Figure 9 shows the distributions of operations of different costs, including fast, medium, and slow reads and writes, as well as re-writes. Comparing to Li et al.'s work, AGCR issues considerably more low-cost reads and writes, thus achieving great improvement. In addition, the percentage of re-write operations is negligible, no more than 1% of all accesses issued by the host, thanks to the small portion of interleaved-access page requests.

5 Conclusions and Future Work

In this paper, we introduced AGCR, a cost regulation approach for flash memory to improve access performance. The proposed approach is motivated by observations derived from widely studied workloads. Based on these observations, we propose an accurate access characterization method, and regulate access costs to improve performance. Our simulation results show that AGCR is effective, reducing I/O latency by as much as 48.3% and 20.4% for read and write requests respectively, compared to the state-of-the-art approach.

Acknowledgments

We thank our shepherd Gala Yadgar and the anonymous reviewers for their comments and suggestions. This work is supported by the Fundamental Research Funds for the Central Universities (CDJZR185502), NSFC 61402059, National 863 Programs 2015AA015304 and 2013AA013202, NSFC 61472052, and Chongqing Research Program cstc2014yykfB40007.

References

[1] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proceedings of the USENIX Conference on File and Storage Technologies*, 2015, pp. 273–286.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proceedings of USENIX Annual Technical Conference*, 2008, pp. 226–229.

[3] K.-D. Suh, B.-H. Suh, Y.-H. Lim, J.-K. Kim, Y.-J. Choi, Y.-N. Koh, S.-S. Lee, S.-C. Kwon, B.-S. Choi, J.-S. Yum, J.-H. Choi, J.-R. Kim, and H.-K. Lim, "A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 11, pp. 1149–1156, 1995.

[4] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo, "Characterizing the performance of flash memory storage devices and its impact on algorithm design," in *Proceedings of the International Conference on Experimental Algorithms*, 2008, pp. 208–219.

[5] G. Yadgar, E. Yaakobi, and A. Schuster, "Write once, get 50% free: Saving SSD erase costs using WOM codes," in *Proceedings of the USENIX Conference on File and Storage Technologies*, 2015, pp. 257–271.

[6] G. Dong, N. Xie, and T. Zhang, "Enabling NAND flash memory use soft-decision error correction codes at minimal read latency overhead," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 60, no. 9, pp. 2412–2421, 2013.

[7] K.-C. Ho, P.-C. Fang, H.-P. Li, C.-Y. Wang, and H.-C. Chang, "A 45nm 6b/cell charge-trapping flash memory using LDPC-based ECC and drift-immune soft-sensing engine," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, 2013, pp. 222–223.

[8] X. Jimenez, D. Novo, and P. Ienne, "Phoenix: reviving MLC blocks as SLC to extend NAND flash devices lifetime," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013, pp. 226–229.

[9] Y. Pan, G. Dong, Q. Wu, and T. Zhang, "Quasi-nonvolatile SSD: Trading flash memory nonvolatility to improve storage system performance for enterprise applications," in *Proceedings of the High Performance Computer Architecture*, 2012, pp. 1–10.

[10] R.-S. Liu, C.-L. Yang, and W. Wu, "Optimizing NAND flash-based SSDs via retention relaxation," in *Proceedings of the USENIX conference on File and Storage Technologies*, 2012, pp. 1–11.

[11] Y. Luo, Y. Cai, S. Ghose, J. Choi, and O. Mutlu, "WARM: Improving NAND flash memory lifetime with write-hotness aware retention management," in *Proceedings of Mass Storage Systems and*, 2015, pp. 1–14.

[12] G. Wu, X. He, N. Xie, and T. Zhang, "Exploiting workload dynamics to improve ssd read latency via differentiated error correction codes," *ACM Transactions on Design Automation of Electronic Systems*, vol. 18, no. 4, pp. 55:1–55:22, 2013.

[13] Q. Li, L. Shi, C. Gao, K. Wu, C. J. Xue, Q. Zhuge, and E. H.-M. Sha, "Maximizing IO performance via conflict reduction for flash memory storage systems," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, 2015, pp. 904–907.

[14] K. Zhao, W. Zhao, H. Sun, T. Zhang, X. Zhang, and N. Zheng, "LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives," in *Proceedings of the USENIX Conference on File and Storage Technologies*, 2013, pp. 244–256.

[15] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1141–1155, 2013.

[16] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," in *Proceedings of the international conference on Supercomputing*, 2011, pp. 96–107.

[17] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to SSDs: analysis of tradeoffs," in *Proceedings of ACM European conference on Computer systems*, 2009, pp. 145–158.

[18] H. Park, J. Kim, J. Choi, D. Lee, and S. H. Noh, "Incremental redundancy to reduce data retention errors in flash-based SSDs," in *Proceedings of Mass Storage Systems and Technologies*, 2015, pp. 1–13.

[19] D. Park and D. H. Du, “Hot data identification for flash-based storage systems using multiple bloom filters,” in *IEEE Symposium on Mass Storage Systems and Technologies*, 2011, pp. 1–11.

[20] G. Wu, B. Eckart, and X. He, “BPAC: An adaptive write buffer management scheme for flash-based Solid State Drives,” in *IEEE Symposium on Mass Storage Systems and Technologies*, 2010, pp. 1–6.

[21] L. Shi, J. Li, C. J. Xue, C. Yang, and X. Zhou, “ExLRU: a unified write buffer cache management for flash memory,” in *Proceedings of ACM international conference on Embedded software*, 2011, pp. 339–348.

[22] H. Kim and S. Ahn, “BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage,” in *Proceedings of the USENIX Conference on File and Storage Technologies*, 2008, pp. 1–14.

[23] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, “LAST: locality-aware sector translation for NAND flash memory-based storage systems,” *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 36–42, 2008.

[24] K. Zhao, K. S. Venkataraman, X. Zhang, J. Li, N. Zheng, and T. Zhang, “Over-clocked SSD: Safely running beyond flash memory chip I/O clock specs,” in *Proceedings of High Performance Computer Architecture*, 2014, pp. 536–545.

[25] Y. Pan, G. Dong, and T. Zhang, “Exploiting memory device wear-out dynamics to improve NAND flash memory system performance,” in *Proceedings of the USENIX conference on File and Storage Technologies*, 2011, pp. 1–14.

WiscKey: Separating Keys from Values in SSD-Conscious Storage

Lanyue Lu, Thanumalayan Sankaranarayana Pillai,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin, Madison

Abstract

We present WiscKey, a persistent LSM-tree-based key-value store with a performance-oriented data layout that separates keys from values to minimize I/O amplification. The design of WiscKey is highly SSD optimized, leveraging both the sequential and random performance characteristics of the device. We demonstrate the advantages of WiscKey with both microbenchmarks and YCSB workloads. Microbenchmark results show that WiscKey is 2.5×–111× faster than LevelDB for loading a database and 1.6×–14× faster for random lookups. WiscKey is faster than both LevelDB and RocksDB in all six YCSB workloads.

1 Introduction

Persistent key-value stores play a critical role in a variety of modern data-intensive applications, including web indexing [16, 48], e-commerce [24], data deduplication [7, 22], photo stores [12], cloud data [32], social networking [9, 25, 51], online gaming [23], messaging [1, 29], software repository [2] and advertising [20]. By enabling efficient insertions, point lookups, and range queries, key-value stores serve as the foundation for this growing group of important applications.

For write-intensive workloads, key-value stores based on Log-Structured Merge-Trees (LSM-trees) [43] have become the state of the art. Various distributed and local stores built on LSM-trees are widely deployed in large-scale production environments, such as BigTable [16] and LevelDB [48] at Google, Cassandra [33], HBase [29] and RocksDB [25] at Facebook, PNUTS [20] at Yahoo!, and Riak [4] at Basho. The main advantage of LSM-trees over other indexing structures (such as B-trees) is that they maintain sequential access patterns for writes. Small updates on B-trees may involve many random writes, and are hence not efficient on either solid-state storage devices or hard-disk drives.

To deliver high write performance, LSM-trees batch key-value pairs and write them sequentially. Subsequently, to enable efficient lookups (for both individual keys as well as range queries), LSM-trees continuously read, sort, and write key-value pairs in the background, thus maintaining keys and values in sorted order. As a result, the same data is read and written multiple times throughout its lifetime; as we show later (§2), this I/O amplification in typical LSM-trees can reach a factor of 50x or higher [39, 54].

The success of LSM-based technology is tied closely to its usage upon classic hard-disk drives (HDDs). In HDDs, random I/Os are over 100× slower than sequential ones [43]; thus, performing additional sequential reads and writes to continually sort keys and enable efficient lookups represents an excellent trade-off.

However, the storage landscape is quickly changing, and modern solid-state storage devices (SSDs) are supplanting HDDs in many important use cases. As compared to HDDs, SSDs are fundamentally different in their performance and reliability characteristics; when considering key-value storage system design, we believe the following three differences are of paramount importance. First, the difference between random and sequential performance is not nearly as large as with HDDs; thus, an LSM-tree that performs a large number of sequential I/Os to reduce later random I/Os may be wasting bandwidth needlessly. Second, SSDs have a large degree of internal parallelism; an LSM built atop an SSD must be carefully designed to harness said parallelism [53]. Third, SSDs can wear out through repeated writes [34, 40]; the high write amplification in LSM-trees can significantly reduce device lifetime. As we will show in the paper (§4), the combination of these factors greatly impacts LSM-tree performance on SSDs, reducing throughput by 90% and increasing write load by a factor over 10. While replacing an HDD with an SSD underneath an LSM-tree does improve performance, with current LSM-tree technology, the SSD's true potential goes largely unrealized.

In this paper, we present WiscKey, an SSD-conscious persistent key-value store derived from the popular LSM-tree implementation, LevelDB. The central idea behind WiscKey is the separation of keys and values [42]; only keys are kept sorted in the LSM-tree, while values are stored separately in a log. In other words, we decouple key sorting and garbage collection in WiscKey while LevelDB bundles them together. This simple technique can significantly reduce write amplification by avoiding the unnecessary movement of values while sorting. Furthermore, the size of the LSM-tree is noticeably decreased, leading to fewer device reads and better caching during lookups. WiscKey retains the benefits of LSM-tree technology, including excellent insert and lookup performance, but without excessive I/O amplification.

Separating keys from values introduces a number of challenges and optimization opportunities. First, range query (scan) performance may be affected because values are not stored in sorted order anymore. WiscKey solves this challenge by using the abundant internal parallelism of SSD devices. Second, WiscKey needs garbage collection to reclaim the free space used by invalid values. WiscKey proposes an online and lightweight garbage collector which only involves sequential I/Os and impacts the foreground workload minimally. Third, separating keys and values makes crash consistency challenging; WiscKey leverages an interesting property in modern file systems, that appends never result in garbage data on a crash. WiscKey optimizes performance while providing the same consistency guarantees as found in modern LSM-based systems.

We compare the performance of WiscKey with LevelDB [48] and RocksDB [25], two popular LSM-tree key-value stores. For most workloads, WiscKey performs significantly better. With LevelDB's own microbenchmark, WiscKey is 2.5×–111× faster than LevelDB for loading a database, depending on the size of the key-value pairs; for random lookups, WiscKey is 1.6×–14× faster than LevelDB. WiscKey's performance is not always better than standard LSM-trees; if small values are written in random order, and a large dataset is range-queried sequentially, WiscKey performs worse than LevelDB. However, this workload does not reflect real-world use cases (which primarily use shorter range queries) and can be improved by log reorganization. Under YCSB macrobenchmarks [21] that reflect real-world use cases, WiscKey is faster than both LevelDB and RocksDB in all six YCSB workloads, and follows a trend similar to the load and random lookup microbenchmarks.

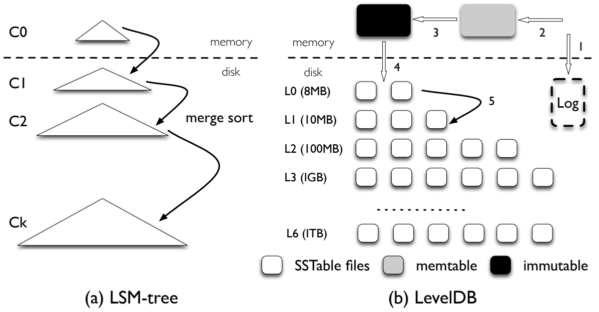
The rest of the paper is organized as follows. We first describe the background and motivation in Section 2. Section 3 explains the design of WiscKey, and Section 4 analyzes its performance. We briefly describe related work in Section 5, and conclude in Section 6.

2 Background and Motivation

In this section, we first describe the concept of a Log-Structured Merge-tree (LSM-tree). Then, we explain the design of LevelDB, a popular key-value store based on LSM-tree technology. We investigate read and write amplification in LevelDB. Finally, we describe the characteristics of modern storage hardware.

2.1 Log-Structured Merge-Tree

An LSM-tree is a persistent structure that provides efficient indexing for a key-value store with a high rate of inserts and deletes [43]. It defers and batches data writes into large chunks to use the high sequential bandwidth of hard drives. Since random writes are nearly two orders of magnitude slower than sequential writes on hard drives, LSM-trees provide better write performance than traditional B-trees, which require random accesses.



(a) LSM-tree

(b) LevelDB

Figure 1: **LSM-tree and LevelDB Architecture.** *This figure shows the standard LSM-tree and LevelDB architecture. For LevelDB, inserting a key-value pair goes through many steps: (1) the log file; (2) the memtable; (3) the immutable memtable; (4) a SSTable in L0; (5) compacted to further levels.*

An LSM-tree consists of a number of components of exponentially increasing sizes, C_0 to C_k, as shown in Figure 1. The C_0 component is a memory-resident update-in-place sorted tree, while the other components C_1 to C_k are disk-resident append-only B-trees.

During an insert in an LSM-tree, the inserted key-value pair is appended to an on-disk sequential log file, so as to enable recovery in case of a crash. Then, the key-value pair is added to the in-memory C_0, which is sorted by keys; C_0 allows efficient lookups and scans on recently inserted key-value pairs. Once C_0 reaches its size limit, it will be merged with the on-disk C_1 in an approach similar to merge sort; this process is known as *compaction*. The newly merged tree will be written to disk sequentially, replacing the old version of C_1. Compaction (i.e., merge sorting) also happens for on-disk components, when each C_i reaches its size limit. Note that compactions are only performed between adjacent levels (C_i and C_{i+1}), and they can be executed asynchronously in the background.

To serve a lookup operation, LSM-trees may need to search multiple components. Note that C_0 contains the freshest data, followed by C_1, and so on. Therefore, to retrieve a key-value pair, the LSM-tree searches components starting from C_0 in a cascading fashion until it locates the desired data in the smallest component C_i. Compared with B-trees, LSM-trees may need multiple reads for a point lookup. Hence, LSM-trees are most useful when inserts are more common than lookups [43].

2.2 LevelDB

LevelDB is a widely used key-value store based on LSM-trees that is inspired by BigTable [16, 48]. LevelDB supports range queries, snapshots, and other features that are useful in modern applications. In this section, we briefly describe the core design of LevelDB.

The overall architecture of LevelDB is shown in Figure 1. The main data structures in LevelDB are an on-disk log file, two in-memory sorted skiplists (*memtable* and *immutable memtable*), and seven levels (L_0 to L_6) of on-disk Sorted String Table (*SSTable*) files. LevelDB initially stores inserted key-value pairs in a log file and the in-memory *memtable*. Once the memtable is full, LevelDB switches to a new memtable and log file to handle further inserts from the user. In the background, the previous memtable is converted into an immutable memtable, and a compaction thread then flushes it to the disk, generating a new SSTable file (about 2 MB usually) at level 0 (L_0); the previous log file is discarded.

The size of all files in each level is limited, and increases by a factor of ten with the level number. For example, the size limit of all files at L_1 is 10 MB, while the limit of L_2 is 100 MB. To maintain the size limit, once the total size of a level L_i exceeds its limit, the compaction thread will choose one file from L_i, merge sort with all the overlapped files of L_{i+1}, and generate new L_{i+1} SSTable files. The compaction thread continues until all levels are within their size limits. Also, during compaction, LevelDB ensures that all files in a particular level, except L_0, do not overlap in their key-ranges; keys in files of L_0 can overlap with each other since they are directly flushed from memtable.

To serve a lookup operation, LevelDB searches the memtable first, immutable memtable next, and then files L_0 to L_6 in order. The number of file searches required to locate a random key is bounded by the maximum number of levels, since keys do not overlap between files within a single level, except in L_0. Since files in L_0 can contain overlapping keys, a lookup may search multiple files at L_0. To avoid a large lookup latency, LevelDB slows down the foreground write traffic if the number of files at L_0 is bigger than eight, in order to wait for the compaction thread to compact some files from L_0 to L_1.

2.3 Write and Read Amplification

Write and read amplification are major problems in LSM-trees such as LevelDB. Write (read) amplification is defined as the ratio between the amount of data written to (read from) the underlying storage device and the amount of data requested by the user. In this section, we analyze the write and read amplification in LevelDB.

To achieve mostly-sequential disk access, LevelDB writes more data than necessary (although still sequentially), i.e., LevelDB has high write amplification. Since the size limit of L_i is 10 times that of L_{i-1}, when merging a file from L_{i-1} to L_i during compaction, LevelDB may read up to 10 files from L_i in the worst case, and write back these files to L_i after sorting. Therefore, the write amplification of moving a file across two levels can be up to 10. For a large dataset, since any newly generated table file can eventually migrate from L_0 to L_6 through a series of compaction steps, write amplification can be over 50 (10 for each gap between L_1 to L_6).

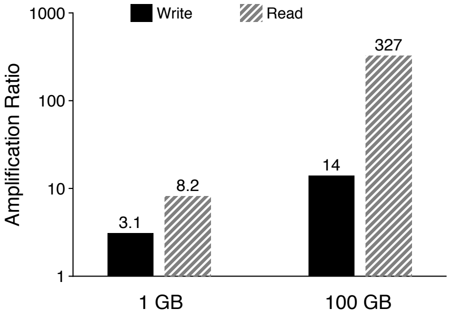


Figure 2: **Write and Read Amplification.** *This figure shows the write amplification and read amplification of LevelDB for two different database sizes, 1 GB and 100 GB. Key size is 16 B and value size is 1 KB.*

Read amplification has been a major problem for LSM-trees due to trade-offs made in the design. There are two sources of read amplification in LevelDB. First, to lookup a key-value pair, LevelDB may need to check multiple levels. In the worst case, LevelDB needs to check eight files in L_0, and one file for each of the remaining six levels: a total of 14 files. Second, to find a key-value pair within a SSTable file, LevelDB needs to read multiple metadata blocks within the file. Specifically, the amount of data actually read is given by `(index block + bloom-filter blocks + data block)`. For example, to lookup a 1-KB key-value pair, LevelDB needs to read a 16-KB index block, a 4-KB bloom-filter block, and a 4-KB data block; in total, 24 KB. Therefore, considering the 14 SSTable files in the worst case, the read amplification of LevelDB is 24 × 14 = 336. Smaller key-value pairs will lead to an even higher read amplification.

To measure the amount of amplification seen in practice with LevelDB, we perform the following experiment. We first load a database with 1-KB key-value pairs, and then lookup 100,000 entries from the database; we use two different database sizes for the initial load, and choose keys randomly from a uniform distribution. Figure 2 shows write amplification during the load phase and read amplification during the lookup phase. For a 1-GB database, write amplification is 3.1, while for a 100-GB database, write amplification increases to 14. Read amplification follows the same trend: 8.2 for the 1-GB database and 327 for the 100-GB database. The reason write amplification increases with database size is straightforward. With more data inserted into a database, the key-value pairs will more likely travel further along

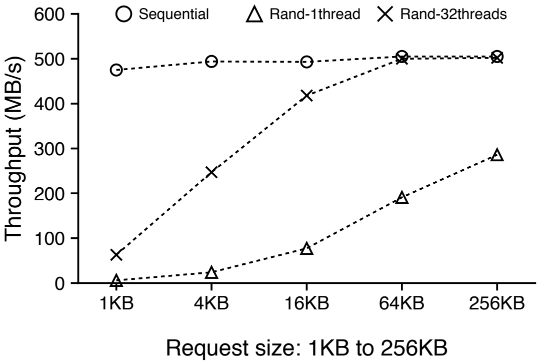


Figure 3: **Sequential and Random Reads on SSD.** *This figure shows the sequential and random read performance for various request sizes on a modern SSD device. All requests are issued to a 100-GB file on ext4.*

the levels; in other words, LevelDB will write data many times when compacting from low levels to high levels. However, write amplification does not reach the worst-case predicted previously, since the average number of files merged between levels is usually smaller than the worst case of 10. Read amplification also increases with the dataset size, since for a small database, all the index blocks and bloom filters in SSTable files can be cached in memory. However, for a large database, each lookup may touch a different SSTable file, paying the cost of reading index blocks and bloom filters each time.

It should be noted that the high write and read amplifications are a justified tradeoff for hard drives. As an example, for a given hard drive with a 10-ms seek latency and a 100-MB/s throughput, the approximate time required to access a random 1K of data is 10 ms, while that for the next sequential block is about 10 μs – the ratio between random and sequential latency is 1000:1. Hence, compared to alternative data structures such as B-Trees that require random write accesses, a sequential-write-only scheme with write amplification less than 1000 will be faster on a hard drive [43, 49]. On the other hand, the read amplification for LSM-trees is still comparable to B-Trees. For example, considering a B-Tree with a height of five and a block size of 4 KB, a random lookup for a 1-KB key-value pair would require accessing six blocks, resulting in a read amplification of 24.

2.4 Fast Storage Hardware

Many modern servers adopt SSD devices to achieve high performance. Similar to hard drives, random writes are considered harmful also in SSDs [10, 31, 34, 40] due to their unique erase-write cycle and expensive garbage collection. Although initial random-write performance for SSD devices is good, the performance can significantly drop after the reserved blocks are utilized. The LSM-tree characteristic of avoiding random writes is hence a natural fit for SSDs; many SSD-optimized key-value stores are based on LSM-trees [25, 50, 53, 54].

However, unlike hard-drives, the relative performance of random reads (compared to sequential reads) is significantly better on SSDs; furthermore, when random reads are issued concurrently in an SSD, the aggregate throughput can match sequential throughput for some workloads [17]. As an example, Figure 3 shows the sequential and random read performance of a 500-GB Samsung 840 EVO SSD, for various request sizes. For random reads by a single thread, the throughput increases with the request size, reaching half the sequential throughput for 256 KB. With concurrent random reads by 32 threads, the aggregate throughput matches sequential throughput when the size is larger than 16 KB. For more high-end SSDs, the gap between concurrent random reads and sequential reads is much smaller [3, 39].

As we showed in this section, LSM-trees have a high write and read amplification, which is acceptable for hard drives. Using LSM-trees on a high-performance SSD may waste a large percentage of device bandwidth with excessive writing and reading. In this paper, our goal is to improve the performance of LSM-trees on SSD devices to efficiently exploit device bandwidth.

3 WiscKey

The previous section explained how LSM-trees maintain sequential I/O access by increasing I/O amplification. While this trade-off between sequential I/O access and I/O amplification is justified for traditional hard disks, they are not optimal for modern hardware utilizing SSDs. In this section, we present the design of WiscKey, a key-value store that minimizes I/O amplification on SSDs.

To realize an SSD-optimized key-value store, WiscKey includes four critical ideas. First, WiscKey separates keys from values, keeping only keys in the LSM-tree and the values in a separate log file. Second, to deal with unsorted values (which necessitate random access during range queries), WiscKey uses the parallel random-read characteristic of SSD devices. Third, WiscKey utilizes unique crash-consistency and garbage-collection techniques to efficiently manage the value log. Finally, WiscKey optimizes performance by removing the LSM-tree log without sacrificing consistency, thus reducing system-call overhead from small writes.

3.1 Design Goals

WiscKey is a single-machine persistent key-value store, derived from LevelDB. It can be deployed as the storage engine for a relational database (e.g., MySQL) or a distributed key-value store (e.g., MongoDB). It provides the same API as LevelDB, including `Put(key, value)`, `Get(key)`, `Delete(key)` and `Scan(start, end)`. The design of WiscKey follows these main goals.

Low write amplification. Write amplification introduces extra unnecessary writes. Even though SSD devices have higher bandwidth compared to hard drives, large write amplification can consume most of the write bandwidth (over 90% is not uncommon) and decrease the SSD's lifetime due to limited erase cycles. Therefore, it is important to minimize write amplification, so as to improve workload performance and SSD lifetime.

Low read amplification. Large read amplification causes two problems. First, the throughput of lookups is significantly reduced by issuing multiple reads for each lookup. Second, the large amount of data loaded into memory decreases the efficiency of the cache. WiscKey targets a small read amplification to speedup lookups.

SSD optimized. WiscKey is optimized for SSD devices by matching its I/O patterns with the performance characteristics of SSD devices. Specifically, sequential writes and parallel random reads are effectively utilized so that applications can fully utilize the device's bandwidth.

Feature-rich API. WiscKey aims to support modern features that have made LSM-trees popular, such as range queries and snapshots. Range queries allow scanning a contiguous sequence of key-value pairs. Snapshots allow capturing the state of the database at a particular time and then performing lookups on the state.

Realistic key-value sizes. Keys are usually small in modern workloads (e.g., 16 B) [7, 8, 11, 22, 35], though value sizes can vary widely (e.g., 100 B to larger than 4 KB) [6, 11, 22, 28, 32, 49]. WiscKey aims to provide high performance for this realistic set of key-value sizes.

3.2 Key-Value Separation

The major performance cost of LSM-trees is the compaction process, which constantly sorts SSTable files. During compaction, multiple files are read into memory, sorted, and written back, which could significantly affect the performance of foreground workloads. However, sorting is required for efficient retrieval; with sorting, range queries (i.e., scan) will result mostly in sequential access to multiple files, while point queries would require accessing at most one file at each level.

WiscKey is motivated by a simple revelation. Compaction only needs to sort keys, while values can be managed separately [42]. Since keys are usually smaller than values, compacting only keys could significantly reduce the amount of data needed during the sorting. In WiscKey, only the location of the value is stored in the LSM-tree with the key, while the actual values are stored elsewhere in an SSD-friendly fashion. With this design, for a database with a given size, the size of the LSM-tree of WiscKey is much smaller than that of LevelDB. The smaller LSM-tree can remarkably reduce the write amplification for modern workloads that have a moderately large value size. For example, assuming a 16-B key, a 1-KB value, and a write amplification of 10 for keys (in the LSM-tree) and 1 for values, the effective write amplification of WiscKey is only (10 × 16 + 1024) / (16 + 1024) = **1.14**. In addition to improving the write performance of applications, the reduced write amplification also improves an SSD's lifetime by requiring fewer erase cycles.

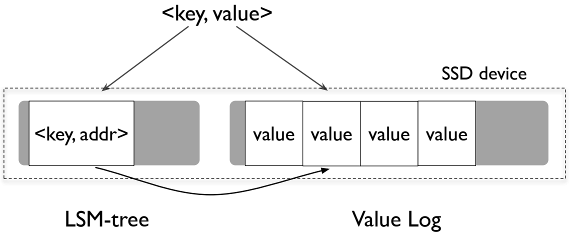


Figure 4: **WiscKey Data Layout on SSD.** *This figure shows the data layout of WiscKey on a single SSD device. Keys and value's locations are stored in LSM-tree while values are appended to a separate value log file.*

WiscKey's smaller read amplification improves lookup performance. During lookup, WiscKey first searches the LSM-tree for the key and the value's location; once found, another read is issued to retrieve the value. Readers might assume that WiscKey will be slower than LevelDB for lookups, due to its extra I/O to retrieve the value. However, since the LSM-tree of WiscKey is much smaller than LevelDB (for the same database size), a lookup may search fewer levels of table files in the LSM-tree and a significant portion of the LSM-tree can be easily cached in memory. Hence, each lookup only requires a single random read (for retrieving the value) and thus achieves a lookup performance better than LevelDB. For example, assuming 16-B keys and 1-KB values, if the size of the entire key-value dataset is 100 GB, then the size of the LSM-tree is only around 2 GB (assuming a 12-B cost for a value's location and size), which can be easily cached in modern servers which have over 100-GB of memory.

WiscKey's architecture is shown in Figure 4. Keys are stored in an LSM-tree while values are stored in a separate value-log file, the *vLog*. The artificial value stored along with the key in the LSM-tree is the address of the actual value in the vLog.

When the user inserts a key-value pair in WiscKey, the value is first appended to the vLog, and the key is then inserted into the LSM tree along with the value's address (`<vLog-offset, value-size>`). Deleting a key simply deletes it from the LSM tree, without touching the vLog. All valid values in the vLog have corresponding keys in the LSM-tree; the other values in the vLog are invalid and will be garbage collected later (§ 3.3.2).

When the user queries for a key, the key is first searched in the LSM-tree, and if found, the corresponding value's address is retrieved. Then, WiscKey reads the value from the vLog. Note that this process is applied to both point queries and range queries.

Although the idea behind key-value separation is simple, it leads to many challenges and optimization opportunities described in the following subsections.

3.3 Challenges

The separation of keys and values makes range queries require random I/O. Furthermore, the separation makes both garbage collection and crash consistency challenging. We now explain how we solve these challenges.

3.3.1 Parallel Range Query

Range queries are an important feature of modern key-value stores, allowing users to scan a range of key-value pairs. Relational databases [26], local file systems [30, 46, 50], and even distributed file systems [37] use key-value stores as their storage engines, and range queries are a core API requested in these environments.

For range queries, LevelDB provides the user with an iterator-based interface with `Seek(key)`, `Next()`, `Prev()`, `Key()` and `Value()` operations. To scan a range of key-value pairs, users can first `Seek()` to the starting key, then call `Next()` or `Prev()` to search keys one by one. To retrieve the key or the value of the current iterator position, users call `Key()` or `Value()`, respectively.

In LevelDB, since keys and values are stored together and sorted, a range query can sequentially read key-value pairs from SSTable files. However, since keys and values are stored separately in WiscKey, range queries require random reads, and are hence not efficient. As we see in Figure 3, the random read performance of a single thread on SSD cannot match the sequential read performance. However, parallel random reads with a fairly large request size can fully utilize the device's internal parallelism, getting performance similar to sequential reads.

To make range queries efficient, WiscKey leverages the parallel I/O characteristic of SSD devices to prefetch values from the vLog during range queries. The underlying idea is that, with SSDs, *only* keys require special attention for efficient retrieval. So long as keys are retrieved efficiently, range queries can use parallel random reads for efficiently retrieving values.

The prefetching framework can easily fit with the current range query interface. In the current interface, if the user requests a range query, an iterator is returned to the user. For each `Next()` or `Prev()` requested on the iterator, WiscKey tracks the access pattern of the range query. Once a contiguous sequence of key-value pairs is requested, WiscKey starts reading a number of following keys from the LSM-tree sequentially. The corresponding value addresses retrieved from the LSM-tree are inserted into a queue; multiple threads will fetch these addresses from the vLog concurrently in the background.

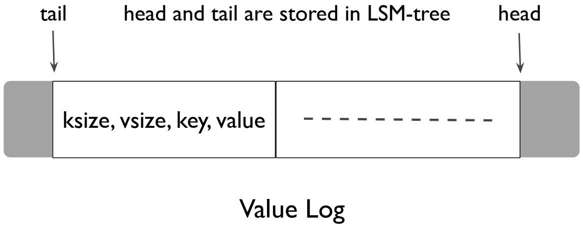


Figure 5: **WiscKey New Data Layout for Garbage Collection.** *This figure shows the new data layout of WiscKey to support an efficient garbage collection. A head and tail pointer are maintained in memory and stored persistently in the LSM-tree. Only the garbage collection thread changes the tail, while all writes to the vLog are append to the head.*

3.3.2 Garbage Collection

Key-value stores based on standard LSM-trees do not immediately reclaim free space when a key-value pair is deleted or overwritten. Rather, during compaction, if data relating to a deleted or overwritten key-value pair is found, the data is discarded and space is reclaimed. In WiscKey, only invalid keys are reclaimed by the LSM-tree compaction. Since WiscKey does not compact values, it needs a special garbage collector to reclaim free space in the vLog.

Since we only store the values in the vLog file (§ 3.2), a naive way to reclaim free space from the vLog is to first scan the LSM-tree to get all the valid value addresses; then, all the values in the vLog without any valid reference from the LSM-tree can be viewed as invalid and reclaimed. However, this method is too heavyweight and is only usable for offline garbage collection.

WiscKey targets a lightweight and online garbage collector. To make this possible, we introduce a small change to WiscKey's basic data layout: while storing values in the vLog, we also store the corresponding key along with the value. The new data layout is shown in Figure 5: the tuple `(key size, value size, key, value)` is stored in the vLog.

WiscKey's garbage collection aims to keep valid values (that do not correspond to deleted keys) in a contiguous range of the vLog, as shown in Figure 5. One end of this range, the *head*, always corresponds to the end of the vLog where new values will be appended. The other end of this range, known as the *tail*, is where garbage collection starts freeing space whenever it is triggered. Only the part of the vLog between the head and the tail contains valid values and will be searched during lookups.

During garbage collection, WiscKey first reads a chunk of key-value pairs (e.g., several MBs) from the tail of the vLog, then finds which of those values are valid (not yet overwritten or deleted) by querying the LSM-tree. WiscKey then appends valid values back to the head of the vLog. Finally, it frees the space occupied previously by the chunk, and updates the tail accordingly.

To avoid losing any data if a crash happens during garbage collection, WiscKey has to make sure that the newly appended valid values and the new tail are persistent on the device before actually freeing space. WiscKey achieves this using the following steps. After appending the valid values to the vLog, the garbage collection calls a `fsync()` on the vLog. Then, it adds these new value's addresses and current tail to the LSM-tree in a synchronous manner; the tail is stored in the LSM-tree as `<‘‘tail’’, tail-vLog-offset>`. Finally, the free space in the vLog is reclaimed.

WiscKey can be configured to initiate and continue garbage collection periodically or until a particular threshold is reached. The garbage collection can also run in offline mode for maintenance. Garbage collection can be triggered rarely for workloads with few deletes and for environments with overprovisioned storage space.

3.3.3 Crash Consistency

On a system crash, LSM-tree implementations usually guarantee atomicity of inserted key-value pairs and in-order recovery of inserted pairs. Since WiscKey's architecture stores values separately from the LSM-tree, obtaining the same crash guarantees can appear complicated. However, WiscKey provides the same crash guarantees by using an interesting property of modern file systems (such as ext4, btrfs, and xfs). Consider a file that contains the sequence of bytes $\langle b_1 b_2 b_3 ... b_n \rangle$, and the user appends the sequence $\langle b_{n+1} b_{n+2} b_{n+3} ... b_{n+m} \rangle$ to it. If a crash happens, after file-system recovery in modern file systems, the file will be observed to contain the sequence of bytes $\langle b_1 b_2 b_3 ... b_n b_{n+1} b_{n+2} b_{n+3} ... b_{n+x} \rangle \; \exists \; x < m$, i.e., only some prefix of the appended bytes will be added to the end of the file during file-system recovery [45]. It is not possible for random bytes or a non-prefix subset of the appended bytes to be added to the file. Since values are appended sequentially to the end of the vLog file in WiscKey, the aforementioned property conveniently translates as follows: if a value X in the vLog is lost in a crash, all future values (inserted after X) are lost too.

When the user queries a key-value pair, if WiscKey cannot find the key in the LSM-tree because the key had been lost during a system crash, WiscKey behaves exactly like traditional LSM-trees: even if the value had been written in vLog before the crash, it will be garbage collected later. If the key could be found in the LSM tree, however, an additional step is required to maintain consistency. In this case, WiscKey first verifies whether the value address retrieved from the LSM-tree falls within the current valid range of the vLog, and then whether the value found corresponds to the queried key. If the verifications fail, WiscKey assumes that the value was lost during a system crash, deletes the key from the LSM-tree, and informs the user that the key was not found. Since each value added to the vLog has a header including the corresponding key, verifying whether the key and the value match is straightforward; if necessary, a magic number or checksum can be easily added to the header.

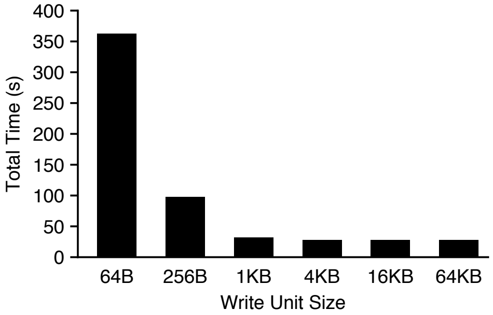


Figure 6: **Impact of Write Unit Size.** *This figure shows the total time to write a 10-GB file to an ext4 file system on an SSD device, followed by a* `fsync()` *at the end. We vary the size of each* `write()` *system call.*

LSM-tree implementations also guarantee the user durability of key value pairs after a system crash if the user specifically requests synchronous inserts. WiscKey implements synchronous inserts by flushing the vLog before performing a synchronous insert into its LSM-tree.

3.4 Optimizations

Separating keys from values in WiscKey provides an opportunity to rethink how the value log is updated and the necessity of the LSM-tree log. We now describe how these opportunities can lead to improved performance.

3.4.1 Value-Log Write Buffer

For each `Put()`, WiscKey needs to append the value to the vLog by using a `write()` system call. However, for an insert-intensive workload, issuing a large number of small writes to a file system can introduce a noticeable overhead, especially on a fast storage device [15, 44]. Figure 6 shows the total time to sequentially write a 10-GB file in ext4 (Linux 3.14). For small writes, the overhead of each system call aggregates significantly, leading to a long run time. With large writes (larger than 4 KB), the device throughput is fully utilized.

To reduce overhead, WiscKey buffers values in a userspace buffer, and flushes the buffer only when the buffer size exceeds a threshold or when the user requests a synchronous insertion. Thus, WiscKey only issues large writes and reduces the number of `write()` system calls. For a lookup, WiscKey first searches the vLog buffer, and if not found there, actually reads from the vLog. Obviously, this mechanism might result in some data (that is buffered) to be lost during a crash; the crash-consistency guarantee obtained is similar to LevelDB.

3.4.2 Optimizing the LSM-tree Log

As shown in Figure 1, a log file is usually used in LSM-trees. The LSM-tree tracks inserted key-value pairs in the log file so that, if the user requests synchronous inserts and there is a crash, the log can be scanned after reboot and the inserted key-value pairs recovered.

In WiscKey, the LSM-tree is only used for keys and value addresses. Moreover, the vLog also records inserted keys to support garbage collection as described in the previous section. Hence, writes to the LSM-tree log file can be avoided without affecting correctness.

If a crash happens before the keys are persistent in the LSM-tree, they can be recovered by scanning the vLog. However, a naive algorithm would require scanning the entire vLog for recovery. So as to require scanning only a small portion of the vLog, WiscKey records the head of the vLog periodically in the LSM-tree, as a key-value pair `<‘‘head’’, head-vLog-offset>`. When a database is opened, WiscKey starts the vLog scan from the most recent head position stored in the LSM-tree, and continues scanning until the end of the vLog. Since the head is stored in the LSM-tree, and the LSM-tree inherently guarantees that keys inserted into the LSM-tree will be recovered in the inserted order, this optimization is crash consistent. Therefore, removing the LSM-tree log of WiscKey is a safe optimization, and improves performance especially when there are many small insertions.

3.5 Implementation

WiscKey is based on LevelDB 1.18. WiscKey creates a vLog when creating a new database, and manages the keys and value addresses in the LSM-tree. The vLog is internally accessed by multiple components with different access patterns. For example, a lookup is served by randomly reading the vLog, while the garbage collector sequentially reads from the tail and appends to the head of the vLog file. We use `posix_fadvise()` to predeclare access patterns for the vLog under different situations.

For range queries, WiscKey maintains a background thread pool with 32 threads. These threads sleep on a thread-safe queue, waiting for new value addresses to arrive. When prefetching is triggered, WiscKey inserts a fixed number of value addresses to the worker queue, and then wakes up all the sleeping threads. These threads will start reading values in parallel, caching them in the buffer cache automatically.

To efficiently garbage collect the free space of the vLog, we use the hole-punching functionality of modern file systems (`fallocate()`). Punching a hole in a file can free the physical space allocated, and allows WiscKey to elastically use the storage space. The maximal file size on modern file systems is big enough for WiscKey to run a long time without wrapping back to the beginning of the file; for example, the maximal file size is 64 TB on ext4, 8 EB on xfs and 16 EB on btrfs. The vLog can be trivially adapted into a circular log if necessary.

4 Evaluation

In this section, we present evaluation results that demonstrate the benefits of the design choices of WiscKey.

All experiments are run on a testing machine with two Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz processors and 64-GB of memory. The operating system is 64-bit Linux 3.14, and the file system used is ext4. The storage device used is a 500-GB Samsung 840 EVO SSD, which has 500 MB/s sequential-read and 400 MB/s sequential-write maximal performance. Random read performance of the device is shown in Figure 3.

4.1 Microbenchmarks

We use `db_bench` (the default microbenchmarks in LevelDB) to evaluate LevelDB and WiscKey. We always use a key size of 16 B, but perform experiments for different value sizes. We disable data compression for easier understanding and analysis of performance.

4.1.1 Load Performance

We now describe the results for the sequential-load and random-load microbenchmarks. The former benchmark constructs a 100-GB database by inserting keys in a sequential order, while the latter inserts keys in a uniformly distributed random order. Note that the sequential-load benchmark does not cause compaction in either LevelDB or WiscKey, while the random-load does.

Figure 7 shows the sequential-load throughput of LevelDB and WiscKey for a wide range of value sizes: the throughput of both stores increases with the value size. But, even for the largest value size considered (256 KB), LevelDB's throughput is far from the device bandwidth. To analyze this further, Figure 8 shows the distribution of the time spent in different components during each run of the benchmark, for LevelDB; time is spent in three major parts: writing to the log file, inserting to the memtable, and waiting for the memtable to be flushed to the device. For small key-value pairs, writing to the log file accounts for the most significant percentage of the total time, for the reasons explained in Figure 6. For larger pairs, log writing and the memtable sorting are more efficient, while memtable flushes are the bottleneck. Unlike LevelDB, WiscKey reaches the full device bandwidth for value sizes more than 4 KB. Since it does not write to the LSM-tree log and buffers appends to the vLog, it is 3× faster even for small values.

Figure 9 shows the random-load throughput of LevelDB and WiscKey for different value sizes. LevelDB's throughput ranges from only 2 MB/s (64-B value size) to 4.1 MB/s (256-KB value size), while

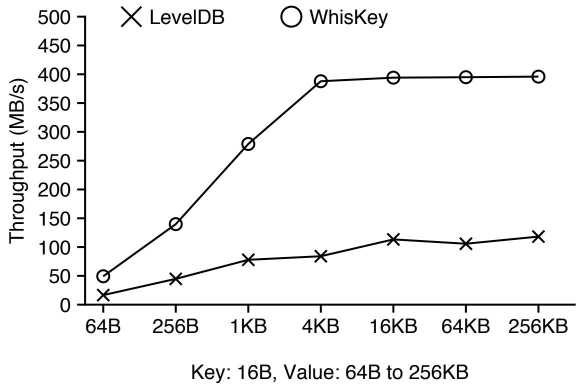


Figure 7: **Sequential-load Performance.** *This figure shows the sequential-load throughput of LevelDB and WiscKey for different value sizes for a 100-GB dataset. Key size is 16 B.*

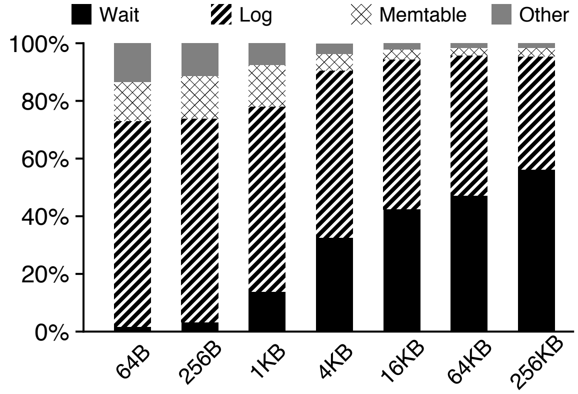


Figure 8: **Sequential-load Time Breakup of LevelDB.** *This figure shows the percentage of time incurred in different components during sequential load in LevelDB.*

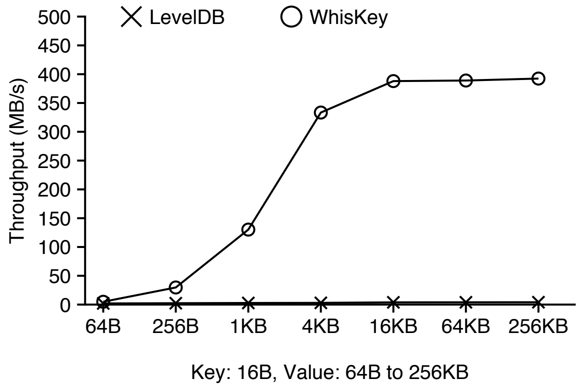


Figure 9: **Random-load Performance.** *This figure shows the random-load throughput of LevelDB and WiscKey for different value sizes for a 100-GB dataset. Key size is 16 B.*

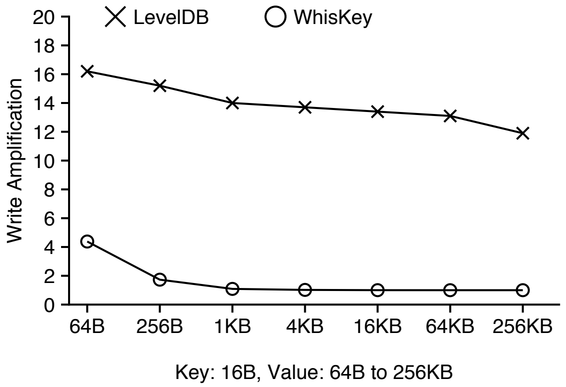


Figure 10: **Write Amplification of Random Load.** *This figure shows the write amplification of LevelDB and WiscKey for randomly loading a 100-GB database.*

WiscKey's throughput increases with the value size, reaching the peak device write throughput after the value size is bigger than 4 KB. WiscKey's throughput is 46× and 111× of LevelDB for the 1-KB and 4-KB value size respectively. LevelDB has low throughput because compaction both consumes a large percentage of the device bandwidth and also slows down foreground writes (to avoid overloading the L_0 of the LSM-tree, as described in Section 2.2). In WiscKey, compaction only introduces a small overhead, leading to the full device bandwidth being effectively utilized. To analyze this further, Figure 10 shows the write amplification of LevelDB and WiscKeẏThe write amplification of LevelDB is always more than 12, while that of WiscKey decreases quickly to nearly 1 when the value size reaches 1 KB, because the LSM-tree of WiscKey is significantly smaller.

4.1.2 Query Performance

We now compare the random lookup (point query) and range query performance of LevelDB and WiscKey. Figure 11 presents the random lookup results of 100,000 operations on a 100-GB random-loaded database. Even though a random lookup in WiscKey needs to check both the LSM-tree and the vLog, the throughput of WiscKey is still much better than LevelDB: for 1-KB value size, WiscKey's throughput is 12× of that of LevelDB. For large value sizes, the throughput of WiscKey is only limited by the random read throughput of the device, as shown in Figure 3. LevelDB has low throughput because of the high read amplification mentioned in Section 2.3. WiscKey performs significantly better because the read amplification is lower due to a smaller LSM-tree. Another reason for WiscKey's better performance is that the compaction process in WiscKey is less intense, thus avoiding many background reads and writes.

Figure 12 shows the range query (scan) performance of LevelDB and WiscKey. For a randomly-loaded database, LevelDB reads multiple files from different levels, while WiscKey requires random accesses to the vLog (but WiscKey leverages parallel random reads). As can be seen from Figure 12, the throughput of LevelDB initially increases with the value size for both databases.

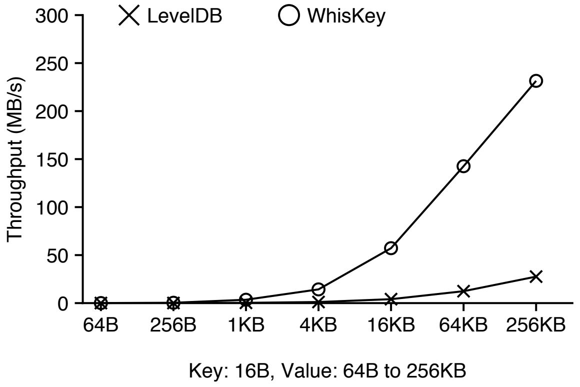


Figure 11: **Random Lookup Performance.** *This figure shows the random lookup performance for 100,000 operations on a 100-GB database that is randomly loaded.*

However, beyond a value size of 4 KB, since an SSTable file can store only a small number of key-value pairs, the overhead is dominated by opening many SSTable files and reading the index blocks and bloom filters in each file. For larger key-value pairs, WiscKey can deliver the device's sequential bandwidth, up to 8.4× of LevelDB. However, WiscKey performs 12× worse than LevelDB for 64-B key-value pairs due to the device's limited parallel random-read throughput for small request sizes; WiscKey's relative performance is better on high-end SSDs with higher parallel random-read throughput [3]. Furthermore, this workload represents a worst-case where the database is randomly-filled and the data is unsorted in the vLog.

Figure 12 also shows the performance of range queries when the data is sorted, which corresponds to a sequentially-loaded database; in this case, both LevelDB and WiscKey can sequentially scan through data. Performance for sequentially-loaded databases follows the same trend as randomly-loaded databases; for 64-B pairs, WiscKey is 25% slower because WiscKey reads both the keys and the values from the vLog (thus wasting bandwidth), but WiscKey is 2.8× faster for large key-value pairs. Thus, with small key-value pairs, log reorganization (sorting) for a random-loaded database can make WiscKey's range-query performance comparable to LevelDB's performance.

4.1.3 Garbage Collection

We now investigate WiscKey's performance while garbage collection is performed in the background. The performance can potentially vary depending on the percentage of free space found during garbage collection, since this affects the amount of data written and the amount of space freed by the garbage collection thread. We use random-load (the workload that is most affected by garbage collection) as the foreground work-

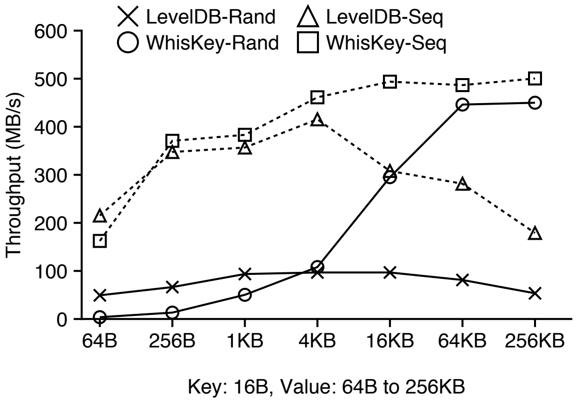


Figure 12: **Range Query Performance.** *This figure shows range query performance. 4 GB of data is queried from a 100-GB database that is randomly (Rand) and sequentially (Seq) loaded.*

load, and study its performance for various percentages of free space. Our experiment specifically involves three steps: we first create a database using random-load, then delete the required percentage of key-value pairs, and finally, we run the random-load workload and measure its throughput while garbage collection happens in the background. We use a key-value size of 4 KB and vary the percentage of free space from 25% to 100%.

Figure 13 shows the results: if 100% of data read by the garbage collector is invalid, the throughput is only 10% lower. Throughput is only marginally lower because garbage collection reads from the tail of the vLog and writes only valid key-value pairs to the head; if the data read is entirely invalid, no key-value pair needs to be written. For other percentages of free space, throughput drops about 35% since the garbage collection thread performs additional writes. Note that, in all cases, while garbage collection is happening, WiscKey is at least 70× faster than LevelDB.

4.1.4 Crash Consistency

Separating keys from values necessitates additional mechanisms to maintain crash consistency. We verify the crash consistency mechanisms of WiscKey by using the ALICE tool [45]; the tool chooses and simulates a comprehensive set of system crashes that have a high probability of exposing inconsistency. We use a test case which invokes a few asynchronous and synchronous `Put()` calls. When configured to run tests for ext4, xfs, and btrfs, ALICE checks more than 3000 selectively-chosen system crashes, and does not report any consistency vulnerability introduced by WiscKey.

The new consistency mechanism also affects WiscKey's recovery time after a crash, and we design an experiment to measure the worst-case recovery time of WiscKey and LevelDB. LevelDB's recovery time is proportional to the size of its log file after the crash;

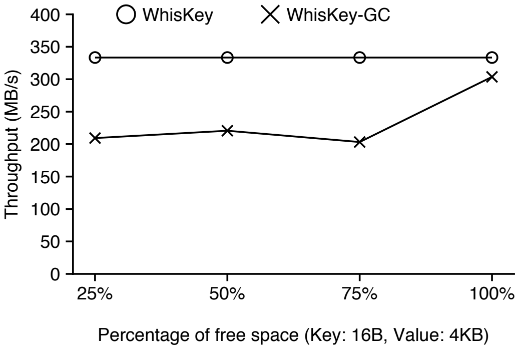


Figure 13: **Garbage Collection.** *This figure shows the performance of WiscKey under garbage collection for various free-space ratios.*

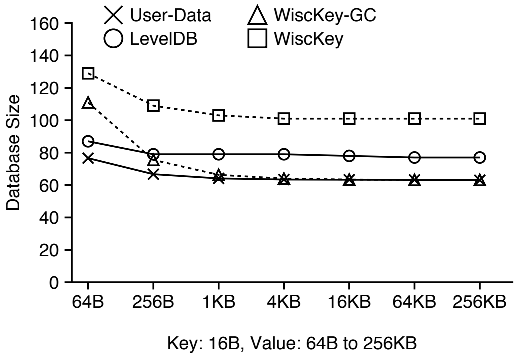


Figure 14: **Space Amplification.** *This figure shows the actual database size of LevelDB and WiscKey for a random-load workload of a 100-GB dataset. User-Data represents the logical database size.*

the log file exists at its maximum size just before the memtable is written to disk. WiscKey, during recovery, first retrieves the head pointer from the LSM-tree, and then scans the vLog file from the head pointer till the end of the file. Since the updated head pointer is persisted on disk when the memtable is written, WiscKey's worst-case recovery time also corresponds to a crash happening just before then. We measured the worst-case recovery time induced by the situation described so far; for 1-KB values, LevelDB takes 0.7 seconds to recover the database after the crash, while WiscKey takes 2.6 seconds. Note that WiscKey can be configured to persist the head pointer more frequently if necessary.

4.1.5 Space Amplification

When evaluating a key-value store, most previous work focused only on read and write amplification. However, space amplification is important for flash devices because of their expensive price-per-GB compared with hard drives. Space amplification is the ratio of the actual size of the database on disk to the logical size of the the database [5]. For example, if a 1-KB key-value pair takes 4 KB of space on disk, then the space amplification is 4. Compression decreases space amplification while extra data (garbage, fragmentation, or metadata) increases space amplification. Compression is disabled to make the discussion simple.

For a sequential-load workload, the space amplification can be near one, given that the extra metadata in LSM-trees is minimal. For a random-load or overwrite workload, space amplification is usually more than one when invalid pairs are not garbage collected fast enough.

Figure 14 shows the database size of LevelDB and WiscKey after randomly loading a 100-GB dataset (the same workload as Figure 9). The space overhead of LevelDB arises due to invalid key-value pairs that are not garbage collected when the workload is finished. The space overhead of WiscKey includes the invalid key-value pairs and the extra metadata (pointers in the LSM-tree and the tuple in the vLog as shown in Figure 5). After garbage collection, the database size of WiscKey is close to the logical database size when the extra metadata is small compared to the value size.

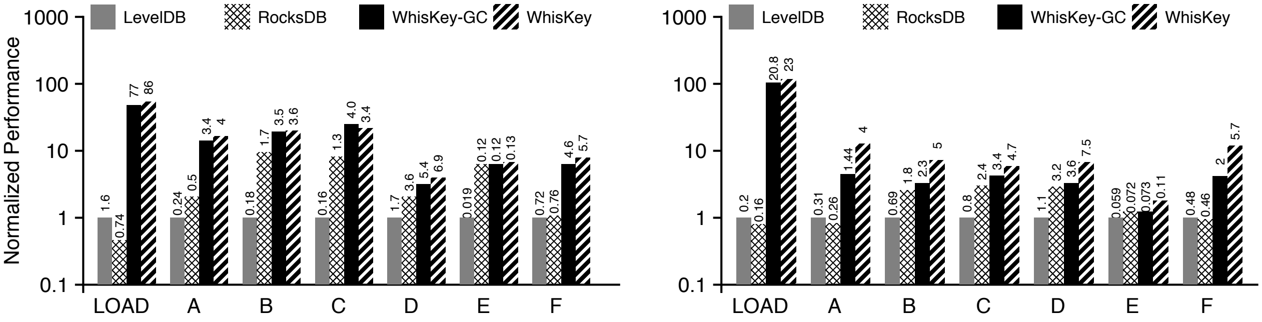
No key-value store can minimize read amplification, write amplification, and space amplification at the same time. Tradeoffs among these three factors are balanced differently in various systems. In LevelDB, the sorting and garbage collection are coupled together. LevelDB trades higher write amplification for lower space amplification; however, the workload performance can be significantly affected. WiscKey consumes more space to minimize I/O amplification when the workload is running; because sorting and garbage collection are decoupled in WiscKey, garbage collection can be done later, thus minimizing its impact on foreground performance.

4.1.6 CPU Usage

We now investigate the CPU usage of LevelDB and WiscKey for various workloads shown in previous sections. The CPU usage shown here includes both the application and operating system usage.

As shown in Table 1, LevelDB has higher CPU usage for sequential-load workload. As we explained in Figure 8, LevelDB spends a large amount of time writing key-value pairs to the log file. Writing to the log file involves encoding each key-value pair, which has high CPU cost. Since WiscKey removes the log file as an optimization, WiscKey has lower CPU usage than LevelDB. For the range query workload, WiscKey uses 32 background threads to do the prefetch; therefore, the CPU usage of WiscKey is much higher than LevelDB.

We find that CPU is not a bottleneck for both LevelDB and WiscKey in our setup. The architecture of LevelDB is based on single writer protocol. The background compaction also only uses one thread. Better concurrency design for multiple cores is explored in RocksDB [25].



(a) Value size: 1KB

(b) Value size: 16KB

Figure 15: **YCSB Macrobenchmark Performance.** *This figure shows the performance of LevelDB, RocksDB, and WiscKey for various YCSB workloads. The X-axis corresponds to different workloads, and the Y-axis shows the performance normalized to LevelDB's performance. The number on top of each bar shows the actual throughput achieved (K ops/s). (a) shows performance under 1-KB values and (b) shows performance under 16-KB values. The load workload corresponds to constructing a 100-GB database and is similar to the random-load microbenchmark. Workload-A has 50% reads and 50% updates, Workload-B has 95% reads and 5% updates, and Workload-C has 100% reads; keys are chosen from a Zipf, and the updates operate on already-existing keys. Workload-D involves 95% reads and 5% inserting new keys (temporally weighted distribution). Workload-E involves 95% range queries and 5% inserting new keys (Zipf), while Workload-F has 50% reads and 50% read-modify-writes (Zipf).*

Workloads	**Seq Load**	**Rand Load**	**Rand Lookup**	**Range Query**
LevelDB	10.6%	6.3%	7.9%	11.2%
WiscKey	8.2%	8.9%	11.3%	30.1%

Table 1: **CPU Usage of LevelDB and WiscKey.** *This table compares the CPU usage of four workloads on LevelDB and WiscKey. Key size is 16 B and value size is 1 KB. Seq-Load and Rand-Load sequentially and randomly load a 100-GB database respectively. Given a 100-GB random-filled database, Rand-Lookup issues 100 K random lookups, while Range-Query sequentially scans 4-GB data.*

4.2 YCSB Benchmarks

The YCSB benchmark [21] provides a framework and a standard set of six workloads for evaluating the performance of key-value stores. We use YCSB to compare LevelDB, RocksDB [25], and WiscKey, on a 100-GB database. In addition to measuring the usual-case performance of WiscKey, we also run WiscKey with garbage collection always happening in the background so as to measure its worst-case performance. RocksDB [25] is a SSD-optimized version of LevelDB with many optimizations, including multiple memtables and background threads for compaction. We use RocksDB with the default configuration parameters. We evaluated the key-value stores with two different value sizes, 1 KB and 16 KB (data compression is disabled).

WiscKey performs significantly better than LevelDB and RocksDB, as shown in Figure 15. For example, during load, for 1-KB values, WiscKey performs at least 50× faster than the other databases in the usual case, and at least 45× faster in the worst case (with garbage collection); with 16-KB values, WiscKey performs 104× better, even under the worst case.

For reads, the Zipf distribution used in most workloads allows popular items to be cached and retrieved without incurring disk access, thus reducing WiscKey's advantage over LevelDB and RocksDB. Hence, WiscKey's relative performance (compared to the LevelDB and RocksDB) is better in Workload-A (50% reads) than in Workload-B (95% reads) and Workload-C (100% reads). However, RocksDB and LevelDB still do not match WiscKey's performance in any of these workloads.

The worst-case performance of WiscKey (with garbage collection switched on always, even for read-only workloads) is better than LevelDB and RocksDB. However, the impact of garbage collection on performance is markedly different for 1-KB and 16-KB values. Garbage collection repeatedly selects and cleans a 4-MB chunk of the vLog; with small values, the chunk will include many key-value pairs, and thus garbage collection spends more time accessing the LSM-tree to verify the validity of each pair. For large values, garbage collection spends less time on the verification, and hence aggressively writes out the cleaned chunk, affecting foreground throughput more. Note that, if necessary, garbage collection can be throttled to reduce its foreground impact.

Unlike the microbenchmark considered previously, Workload-E has multiple small range queries, with each query retrieving between 1 and 100 key-value pairs. Since the workload involves multiple range queries, accessing the first key in each range resolves to a random lookup – a situation favorable for WiscKey. Hence, WiscKey performs better than RocksDB and LevelDB even for 1-KB values.

5 Related Work

Various key-value stores based on hash tables have been proposed for SSD devices. FAWN [8] keeps key-value pairs in a append-only log on the SSD, and uses an in-memory hash table index for fast lookups. FlashStore [22] and SkimpyStash [23] follow the same design, but optimize the in-memory hash table; FlashStore uses cuckoo hashing and compact key signatures, while SkimpyStash moves a part of the table to the SSD using linear chaining. BufferHash [7] uses multiple in-memory hash tables, with bloom filters to choose which hash table to use for a lookup. SILT [35] is highly optimized for memory, and uses a combination of log-structure, hash-table, and sorted-table layouts.WiscKey shares the log-structure data layout with these key-value stores. However, these stores use hash tables for indexing, and thus do not support modern features that have been built atop LSM-tree stores, such as range queries or snapshots. WiscKey instead targets a feature-rich key-value store which can be used in various situations.

Much work has gone into optimizing the original LSM-tree key-value store [43]. bLSM [49] presents a new merge scheduler to bound write latency, thus maintaining a steady write throughput, and also uses bloom filters to improve performance. VT-tree [50] avoids sorting any previously sorted key-value pairs during compaction, by using a layer of indirection. WiscKey instead directly separates values from keys, significantly reducing write amplification regardless of the key distribution in the workload. LOCS [53] exposes internal flash channels to the LSM-tree key-value store, which can exploit the abundant parallelism for a more efficient compaction. Atlas [32] is a distributed key-value store based on ARM processors and erasure coding, and stores keys and values on different hard drives. WiscKey is a standalone key-value store, where the separation between keys and values is highly optimized for SSD devices to achieve significant performance gains. LSM-trie [54] uses a trie structure to organize keys, and proposes a more efficient compaction based on the trie; however, this design sacrifices LSM-tree features such as efficient support for range queries. RocksDB, described previously, still exhibits high write amplification due to its design being fundamentally similar to LevelDB; RocksDB's optimizations are orthogonal to WiscKey's design.

Walnut [18] is a hybrid object store which stores small objects in a LSM-tree and writes large objects directly to the file system. IndexFS [47] stores its metadata in a LSM-tree with the column-style schema to speed up the throughput of insertion. Purity [19] also separates its index from data tuples by only sorting the index and storing tuples in time order. All three systems use similar techniques as WiscKey. However, we solve this problem in a more generic and complete manner, and optimize both load and lookup performance for SSD devices across a wide range of workloads.

Key-value stores based on other data structures have also been proposed. TokuDB [13, 14] is based on fractal-tree indexes, which buffer updates in internal nodes; the keys are not sorted, and a large index has to be maintained in memory for good performance. ForestDB [6] uses a HB+-trie to efficiently index long keys, improving the performance and reducing the space overhead of internal nodes. NVMKV [39] is a FTL-aware key-value store which uses native FTL capabilities, such as sparse addressing, and transactional supports. Vector interfaces that group multiple requests into a single operation are also proposed for key-value stores [52]. Since these key-value stores are based on different data structures, they each have different trade-offs relating to performance; instead, WiscKey proposes improving the widely used LSM-tree structure.

Many proposed techniques seek to overcome the scalability bottlenecks of in-memory key-value stores, such as Mastree [38], MemC3 [27], Memcache [41], MICA [36] and cLSM [28]. These techniques may be adapted for WiscKey to further improve its performance.

6 Conclusions

Key-value stores have become a fundamental building block in data-intensive applications. In this paper, we propose WiscKey, a novel LSM-tree-based key-value store that separates keys and values to minimize write and read amplification. The data layout and I/O patterns of WiscKey are highly optimized for SSD devices. Our results show that WiscKey can significantly improve performance for most workloads. Our hope is that key-value separation and various optimization techniques in WiscKey will inspire the future generation of high-performance key-value stores.

Acknowledgments

We thank the anonymous reviewers and Ethan Miller (our shepherd) for their feedback. We thank the members of the ADSL research group, the RocksDB team (FaceBook), Yinan Li (Microsoft Research) and Bin Fan (Tachyon Nexus) for their suggestions and comments on this work at various stages.

This material was supported by funding from NSF grants CNS-1419199, CNS-1421033, CNS-1319405, and CNS-1218405 as well as generous donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Seagate, Samsung, Veritas, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

[1] Apache HBase. http://hbase.apache.org/, 2007.

[2] Redis. http://redis.io/, 2009.

[3] Fusion-IO ioDrive2. http://www.fusionio.com/products/iodrive2, 2015.

[4] Riak. http://docs.basho.com/riak/, 2015.

[5] RocksDB Blog. http://rocksdb.org/blog/, 2015.

[6] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. *IEEE Transactions on Computers*, Preprint, May 2015.

[7] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and Large CAMs for High-performance Data-intensive Networked Systems. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI '10)*, San Jose, California, April 2010.

[8] David Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[9] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, New York, New York, June 2013.

[10] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.9 edition, 2014.

[11] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the USENIX Annual Technical Conference (USENIX '15)*, Santa Clara, California, July 2015.

[12] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[13] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan Fogel, Bradley Kuszmaul, and Jelani Nelson. Cache-Oblivious Streaming B-trees. In *Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '07)*, San Diego, California, June 2007.

[14] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. On External Memory Graph Traversal. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, San Francisco, California, January 2000.

[15] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*, Atlanta, Georgia, December 2010.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, Seattle, Washington, November 2006.

[17] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-speed Data Processing. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA-11)*, San Antonio, Texas, February 2011.

[18] Jianjun Chen, Chris Douglas, Michi Mutsuzaki, Patrick Quaid, Raghu Ramakrishnan, Sriram Rao, and Russell Sears. Walnut: A Unified Cloud Object Store. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, Scottsdale, Arizona, May 2012.

[19] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, Melbourne, Australia, May 2015.

[20] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!s Hosted Data Serving Platform. In *Proceedings of the VLDB Endowment (PVLDB 2008)*, Auckland, New Zealand, August 2008.

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, Indiana, June 2010.

[22] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. In *Proceedings of the 36th International Conference on Very Large Databases (VLDB 2010)*, Singapore, September 2010.

[23] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*, Athens, Greece, June 2011.

[24] Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Laksh-

man, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007.

[25] Facebook. RocksDB. http://rocksdb.org/, 2013.

[26] Facebook. RocksDB 2015 H2 Roadmap. http://rocksdb.org/blog/2015/rocksdb-2015-h2-roadmap/, 2015.

[27] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, Illinois, April 2013.

[28] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling Concurrent Log-Structured Data Stores. In *Proceedings of the EuroSys Conference (EuroSys '15)*, Bordeaux, France, April 2015.

[29] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis of HDFS Under HBase: A Facebook Messages Case Study. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, California, February 2014.

[30] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST '15)*, Santa Clara, California, February 2015.

[31] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[32] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidus Key-value Storage System for Cloud Data. In *Proceedings of the 31st International Conference on Massive Storage Systems and Technology (MSST '15)*, Santa Clara, California, May 2015.

[33] 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*, Big Sky Resort, Montana, Oct 2009.

[34] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST '15)*, Santa Clara, California, February 2015.

[35] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[36] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle, Washington, April 2014.

[37] Haohui Mai and Jing Zhao. Scaling HDFS to Manage Billions of Files with Key Value Stores. In *The 8th Annual Hadoop Summit*, San Jose, California, Jun 2015.

[38] Yandong Mao, Eddie Kohler, and Robert Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the EuroSys Conference (EuroSys '12)*, Bern, Switzerland, April 2012.

[39] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *Proceedings of the USENIX Annual Technical Conference (USENIX '15)*, Santa Clara, California, July 2015.

[40] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[41] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, Illinois, April 2013.

[42] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. AlphaSort: A RISC Machine Sort. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*, Minneapolis, Minnesota, May 1994.

[43] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[44] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, and Thomas Anderson. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, October 2014.

[45] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting

Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, October 2014.

[46] Kai Ren and Garth Gibson. TABLEFS: Enhancing Metadata Efciency in the Local File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '13)*, San Jose, California, June 2013.

[47] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, New Orleans, Louisana, November 2014.

[48] Sanjay Ghemawat and Jeff Dean. LevelDB. http://code.google.com/p/leveldb, 2011.

[49] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, Scottsdale, Arizona, May 2012.

[50] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building Workload-Independent Storage with VT-Trees. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[51] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving Large-scale Batch Computed Data with Project Voldemort. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[52] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. Using Vector Interfaces to Deliver Millions of IOPS from a Networked Key-value Storage Server. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '12)*, San Jose, California, October 2012.

[53] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *Proceedings of the EuroSys Conference (EuroSys '14)*, Amsterdam, Netherlands, April 2014.

[54] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *Proceedings of the USENIX Annual Technical Conference (USENIX '15)*, Santa Clara, California, July 2015.

Towards Accurate and Fast Evaluation of Multi-Stage Log-Structured Designs

Hyeontaek Lim,[1] David G. Andersen,[1] Michael Kaminsky[2]
[1]*Carnegie Mellon University,* [2]*Intel Labs*

Abstract

Multi-stage log-structured (MSLS) designs, such as LevelDB, RocksDB, HBase, and Cassandra, are a family of storage system designs that exploit the high sequential write speeds of hard disks and flash drives by using multiple append-only data structures. As a first step towards accurate and fast evaluation of MSLS, we propose new analytic primitives and MSLS design models that quickly give accurate performance estimates. Our model can almost perfectly estimate the cost of inserts in LevelDB, whereas the conventional worst-case analysis gives 1.8–3.5X higher estimates than the actual cost. A few minutes of offline analysis using our model can find optimized system parameters that decrease LevelDB's insert cost by up to 9.4–26.2%; our analytic primitives and model also suggest changes to RocksDB that reduce its insert cost by up to 32.0%, without reducing query performance or requiring extra memory.

1 Introduction

Log-structured store designs provide fast write and easy crash recovery for block-based storage devices that have considerably higher sequential write speed than random write speed [37]. In particular, multi-stage versions of log-structured designs, such as LSM-tree [36], COLA [2], and SAMT [42], strive to balance read speed, write speed, and storage space use by segregating fresh and old data in multiple append-only data structures. These designs have been widely adopted in modern datastores including LevelDB [19], RocksDB [12], Bigtable [8], HBase [45], and Cassandra [27].

Given the variety of multi-stage log-structured (MSLS) designs, a system designer is faced with a problem of plenty, raising questions such as: Which design is best for this workload? How should the systems' parameters be set? How sensitive is that choice to changes in workload? Our goal in this paper is to move toward answering these questions and more through an improved—both in quality and in speed—analytical method for understanding and comparing the performance of these systems. This analytical approach can help shed light on how different design choices affect the performance of today's systems, and it provides an opportunity to optimize (based on the analysis) parameter choices given a workload. For example, in Section 6, we show that a few minutes of offline analysis can find improved parameters for LevelDB that decrease the cost of inserts by up to 9.4–26.2%. As another example, in Section 7, we reduce the insert cost in RocksDB by up to 32.0% by changing its system design based upon what we have learned from our analytic approach.

Prior evaluations of MSLS designs largely reside at the two ends of the spectrum: (1) asymptotic analysis and (2) experimental measurement. *Asymptotic analysis* of an MSLS design typically gives a big-*O* term describing the cost of an operation type (e.g., query, insert), but previous asymptotic analyses do not reflect real-world performance because they assume the worst case. *Experimental measurement* of an implementation produces accurate performance numbers, which are often limited to a particular implementation and workload, with lengthy experiment time to explore various system configurations.

This paper proposes a new evaluation method for MSLS designs that provides accurate and fast evaluation without needing to run the full implementations. Our approach uses new analytic primitives that help model the dynamics of MSLS designs. We build upon this model by combining it with a nonlinear solver to help automatically optimize system parameters to maximize performance.

This paper makes four key contributions:

- New analytic primitives to model creating the log structure and merging logs with redundant data (§3);
- System models for LevelDB and RocksDB, representative MSLS designs, using the primitives (§4, §5);
- Optimization of system parameters with the LevelDB model, improving real system performance (§6); and
- Application of lessons from the LevelDB model to the RocksDB system to reduce its write cost (§7).

Section 2 describes representative MSLS designs and common evaluation metrics for MSLS designs. Section 8 broadens the applications of our analytic primitives. Section 9 discusses the implications and limitations of our method. Appendix A provides proofs. Appendices B and C include additional system models.

2 Background

This section introduces a family of multi-stage log-structured designs and their practical variants, and explains metrics commonly used to evaluate these designs.

2.1 Multi-Stage Log-Structured Designs

A multi-stage log-structured (MSLS) design is a storage system design that contains multiple append-only data structures, each of which is created by sequential writes; for instance, several designs use sorted arrays and tables that are often called *SSTables* [19, 42]. These data structures are organized as *stages*, either logically or physically, to segregate different classes of data—e.g., fresh data and old data, frequently modified data and static data, small items and large items, and so forth. *Components* in LSM-tree [36] and *levels* in many designs [2, 19, 42] are examples of stages.

MSLS designs exploit the fast sequential write speed of modern storage devices. On hard disk and flash drives, sequential writes are up to an order of magnitude faster than random writes. By restricting most write operations to incur only sequential I/O, MSLS can provide fast writes.

Using multiple stages reduces the I/O cost for data updates. Frequent changes are often contained within a few stages that either reside in memory and/or are cheap to rewrite—this approach shares the same insight as the generational garbage collection used for memory management [25, 28]. The downside is that the system may have to search in multiple stages to find a single item because the item can exist in any of these stages. This can potentially reduce query performance.

The system moves data between stages based upon certain criteria. Common conditions are the byte size of the data stored in a stage, the age of the stored data, etc. This data migration typically reduces the total data volume by merging multiple data structures and reducing the redundancy between them; therefore, it is referred to as "compaction" or "merge."

MSLS designs are mainly classified by how they organize log structures and how and when they perform compaction. The data structures and compaction strategy significantly affect the cost of various operations.

2.1.1 Log-Structured Merge-Tree

The log-structured merge-tree (LSM-tree) [36] is a write-optimized store design with two or more components, each of which is a tree-like data structure [35]. One component (C_0) resides in memory; the remaining components (C_1,C_2,...) are stored on disk. Each component can hold a set of items, and multiple components can contain multiple items of the same key. A lower-numbered component always stores a newer version of the item than any higher-numbered component does.

For query processing, LSM-tree searches in potentially multiple components. It starts from C_0 and stops as soon as the desired item is found.

Handling inserts involves updating the in-memory component and merging data between components. A new entry is inserted into C_0 (and is also logged to disk for crash recovery), and the new item is migrated over time from C_0 to C_1, from C_1 to C_2, and so on. Frequent updates of the same item are coalesced in C_0 without spilling them to the disk; cold data, in contrast, remains in C_1 and later components which reside on low-cost disk.

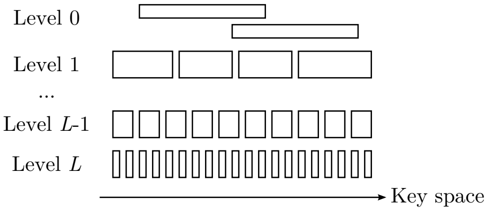


Figure 1: A simplified overview of LevelDB data structures. Each rectangle is an SSTable. Note that the x-axis is the key space; the rectangles are not to scale to indicate their byte size. The memtable and logs are omitted.

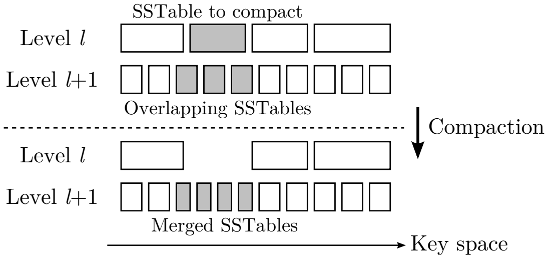


Figure 2: Compaction between two levels in LevelDB.

The data merge in LSM-tree is mostly a sequential I/O operation. The data from C_l is read and merged into C_{l+1}, using a "rolling merge" that creates and updates nodes in the C_{l+1} tree incrementally in the key space.

The authors of LSM-tree suggested maintaining component sizes to follow a geometric progression. The size of a component is r times larger than the previous component size, where r is commonly referred to as a "growth factor," typically between 10 and 20. With such size selection, the expected I/O cost per insert by the data migration is $O((r+1)\log_r N)$, where N is the size of the largest component, i.e., the total number of unique keys. The worst-case lookup incurs $O(\log_r N)$ random I/O by accessing all components, if finding an item in a component costs $O(1)$ random I/O.

2.1.2 LevelDB

LevelDB [19] is a well-known variant of LSM-tree. It uses an in-memory table called a memtable, on-disk log files, and on-disk SSTables. The memtable plays the same role as C_0 of LSM-tree, and write-ahead logging is used for recovery. LevelDB organizes multiple levels that correspond to the components of LSM-tree; however, as shown in Figure 1, LevelDB uses a set of SSTables instead of a single tree-like structure for each level, and LevelDB's first level (level-0) can contain duplicate items across multiple SSTables.

Handing data updates in LevelDB is mostly similar to LSM-tree with a few important differences. Newly inserted data is stored in the memtable and appended to a log file. When the log size exceeds a threshold (e.g., 4 MiB[1]), the content of the memtable is converted into an SSTable and inserted to level-0. When the table count in level-0 reaches a threshold (e.g., 4), LevelDB begins to migrate the data of level-0 SSTables into level-1. For level-1 and later levels, when the aggregate byte size of SSTables in a level reaches a certain threshold, LevelDB picks an SSTable from that level and merges it into the next level. Figure 2 shows the compaction process; it takes all next-level SSTables whose key range overlaps with the SSTable being compacted, replacing the next-level SSTables with new SSTables containing merged items.

The SSTables created by compaction follow several invariants. A new SSTable has a size limit (e.g., 2 MiB), which makes the compaction process incremental. An SSTable cannot have more than a certain amount of overlapping data (e.g., 20 MiB) in the next level, which limits the future cost of compacting the SSTable.

LevelDB compacts SSTables in a circular way within the key space for each level. Fine-grained SSTables and round-robin SSTable selection have interesting implications in characterizing LevelDB's write cost.

There are several variants of LevelDB. A popular version is RocksDB [12], which claims to improve write performance with better support for multithreading. Unlike LevelDB, RocksDB picks the largest SSTable available for concurrent compaction. We discuss the impact of this strategy in Section 7. RocksDB also supports "universal compaction," an alternative compaction strategy that trades read performance for faster writes.

We choose to apply our analytic primitives and modeling techniques to LevelDB in Section 4 because it creates interesting and nontrivial issues related to its use of SSTables and incremental compaction. We show how we can analyze complex compaction strategies such as RocksDB's universal compaction in Section 5.

2.2 Common Evaluation Metrics

This paper focuses on analytic metrics (e.g., per-insert cost factors) more than on experimental metrics (e.g., insert throughput represented in MB/s or kOPS).

Queries and inserts are two common operation types. A query asks for one or more data items, which can also return "not found." An insert stores new data or updates existing item data. While it is hard to define a cost metric for every type of query and insert operation, prior studies extensively used two metrics defined for the *amortized I/O cost per processed item*: read amplification and write amplification.

Read amplification (RA) is the expected number of random read I/O operations to serve a query, assuming that the total data size is much larger than the system memory size, which translates to the expected I/O overhead of query processing [7, 29]. RA is based on the fact that random I/O access on disk and flash is a critical resource for query performance.

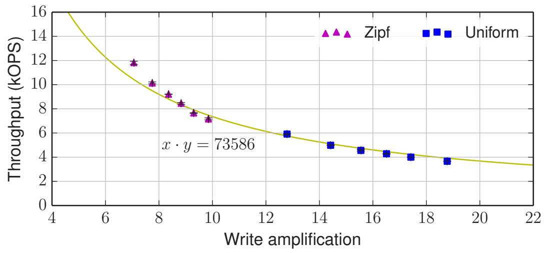


Figure 3: Write amplification is an important metric; increased write amplification decreases insert throughput on LevelDB.

Write amplification (WA) is the expected amount of data written to disk or flash per insert, which measures the I/O overhead of insert processing. Its concept originates from a metric to measure the efficiency of the flash translation layer (FTL), which stores blocks in a log structure-like manner; WA has been adopted later in key-value store studies to project insert throughput and estimate the life expectancy of underlying flash drives [12, 19, 29, 30, 43].

WA and insert throughput are inversely related. Figure 3 shows LevelDB's insert throughput for 1 kB items on a fast flash drive.[2] We vary the total data volume from 1 GB to 10 GB and examine two distributions for the key popularity, uniform and Zipf. Each configuration is run 3 times. Workloads that produce higher WA (e.g., larger data volume and/or uniform workloads) have lower throughput.

In this paper, our main focus is WA. Unlike RA, whose effect on actual system performance can be reduced by dedicating more memory for caching, the effect of WA cannot be mitigated easily without changing the core system design because the written data must be eventually flushed to disk/flash to ensure durability. For the same reason, we do not to use an extended definition of WA that includes the expected amount of data *read* from disk per insert. We discuss how to estimate RA in Section 8.

3 Analytic Primitives

Our goal in later sections is to create simple but accurate models of the write amplification of different MSLS designs. To reach this goal, we first present three new analytic primitives, Unique, Unique^{-1}, and Merge, that form the basis for those models. In Sections 4 and 5, we show how to express the insert and growth behavior of LevelDB and RocksDB using these primitives.

[1] Mi denotes 2^{20}. k, M, and G denote 10^3, 10^6, and 10^9, respectively.

[2] We use Intel® SSDSC2BB160G4T with `fsync` enabled for LevelDB.

3.1 Roles of Redundancy

Redundancy has an important effect on the behavior of an MSLS design. Any given table store (SSTable, etc.) contains at most one entry for a single key, no matter how many inserts were applied for that key. Similarly, when compaction merges tables, the resulting table will also contain only a single copy of the key, no matter how many times it appeared in the tables that were merged. Accurate models must thus consider redundancy.

Asymptotic analyses in prior studies ignore redundancy. Most analyses assume that compactions observe no duplicate keys from insert requests and input tables being merged [2, 19]. The asymptotic analyses therefore give the same answer regardless of skew in the key popularity; it ignores whether all keys are equally popular or some keys are more popular than others. It also estimates only an upper bound on the compaction cost—duplicate keys mean that less total data is written, lowering real-world write amplification.

We first clarify our assumptions and then explain how we quantify the effect of redundancy.

3.2 Notation and Assumptions

Let K be the key space. Without loss of generality, K is the set of all integers in $[0, N-1]$, where N is the total number of unique keys that the workload uses.

A discrete random variable X maps an insert request to the key referred to by the request. f_X is the probability mass function for X, i.e., $f_X(k)$ for $k \in K$ is the probability of having a specific key k for each insert request, assuming the keys in the requests are independent and identically distributed (i.i.d.) and have no spatial locality in popularity. As an example, a Zipf key popularity is defined as $f_X(h(i)) = (1/i^s)/(\sum_{n=1}^{N} 1/n^s)$, where s is the skew and h maps the rank of each key to the key.[3] Since there is no restriction on how f_X should look, it can be built from a key popularity distribution inferred by an empirical workload characterization [1, 38, 49].

Without loss of generality, $0 < f_X(k) < 1$. We can remove any key k satisfying $f_X(k) = 0$ from K because k will never appear in the workload. Similarly, $f_X(k) = 1$ degenerates to a workload with exactly one key, which is trivial to analyze.

A table is a set of the items that contains no duplicate keys. Tables are constructed from a sequence of insert requests or merges of other tables.

L refers to the total number of standard levels in an MSLS design. *Standard* levels include only the levels that follow the invariants of the design; for example, the level-0 in LevelDB does not count towards L because level-0 contains overlapping tables, while other levels do not, and has a different compaction trigger that is based on the table count in the level, not the aggregate size of tables in a level. L is closely related to read amplification; an MSLS design may require L random I/Os to retrieve an item that exists only in the last level (unless the design uses additional data structures such as Bloom filters [5]).

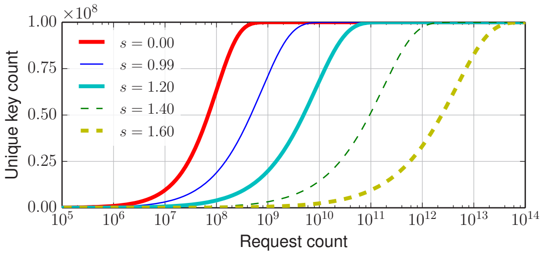


Figure 4: Unique key count as a function of request count for 100 million unique keys, with varying Zipf skew (s).

To avoid complicating the analysis, we assume that all items have equal size (e.g., 1000 bytes). This assumption is consistent with YCSB [11], a widely-used key-value store benchmark. We relax this assumption in Section 8.

3.3 Counting Unique Keys

A sequence of insert requests may contain duplicate keys. The requests with duplicate keys overwrite or modify the stored values. When storing the effect of the requests in a table, only the final (combined) results survive. Thus, a table can be seen as a set of distinct keys in the requests.

We first formulate Unique, a function describing the expected number of unique keys that appear in p requests:

Definition 1.

$$\text{Unique}(p) := N - \sum_{k \in K} (1 - f_X(k))^p \text{ for } p \geq 0.$$

Theorem 1. Unique(p) *is the expected number of unique keys that appear in p requests.*

Figure 4 plots the number of unique keys as a function of the number of insert requests for 100 million unique keys ($N = 10^8$). We use Zipf distributions with varying skew. The unique key count increases as the request count increases, but the increase slows down as the unique key count approaches the total unique key count. The unique key count with less skewed distributions increases more rapidly than with more skewed distributions until it is close to the maximum.

In the context of MSLS designs, Unique gives a hint about how many requests (or how much time) it takes for a level to reach a certain size from an empty state. With no or low skew, a level quickly approaches its full capacity and the system initiates compaction; with high skew, however, it can take a long time to accumulate enough keys to trigger compaction.

We examine another useful function, Unique^{-1}, which

[3] Note that $s = 0$ leads to a *uniform* key popularity, i.e., $f_X(k) = 1/N$. We use $s = 0.99$ frequently to describe a "skewed" or simply "Zipf" distribution for the key popularity, which is the default skew in YCSB [11].

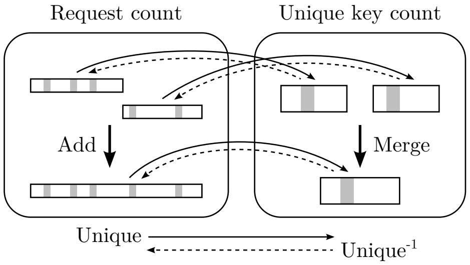


Figure 5: Isomorphism of Unique. Gray bars indicate a certain redundant key.

is the inverse function of Unique. $\text{Unique}^{-1}(u)$ estimates the expected number of requests to observe u unique keys in the requests.[4] By extending the domain of Unique to the real numbers, we can ensure the existence of Unique^{-1}:

Lemma 1. $\text{Unique}^{-1}(u)$ exists for $0 \le u < N$.

We further extend the domain of Unique^{-1} to include N by using limits. $\text{Unique}(\infty) := \lim_{p\to\infty} \text{Unique}(p) = N$; $\text{Unique}^{-1}(N) := \lim_{u\to N} \text{Unique}^{-1}(u) = \infty$.

It is straightforward to compute the value of $\text{Unique}^{-1}(u)$ by solving $\text{Unique}(p) = u$ for p numerically or by approximating Unique and Unique^{-1} with piecewise linear functions.

3.4 Merging Tables

Compaction takes multiple tables and creates a new set of tables that contain no duplicate keys. Nontrivial cases involve tables with overlapping key ranges. For such cases, we can estimate the size of merged tables using a combination of Unique and Unique^{-1}:

Definition 2. $\text{Merge}(u,v) := \text{Unique}(\text{Unique}^{-1}(u) + \text{Unique}^{-1}(v))$ for $0 \le u, v \le N$.

Theorem 2. $\text{Merge}(u,v)$ *is the expected size of a merged table that is created from two tables of sizes u and v.*

In worst-case analysis, merging tables of size u and v results in a new table of size $u+v$, assuming the input tables contain no duplicate keys. The error caused by this assumption grows as u and v approach N and as the key popularity has more skew. For example, with 100 million (10^8) total unique keys and Zipf skew of 0.99, $\text{Merge}(10^7, 9\times 10^7) \approx 9.03 \times 10^7$ keys, whereas the worst-case analysis expects 10^8 keys.

Finally, Unique is an isomorphism as shown in Figure 5. Unique maps the length of a sequence of requests to the number of unique keys in it, and Unique^{-1} does the opposite. Adding request counts corresponds to applying Merge to unique key counts; the addition calculates the length of concatenated request sequences, and Merge obtains the number of unique keys in the merged table. Translating the number of requests to the number of unique keys and vice versa makes it easy to build an MSLS model, as presented in Section 4.

3.5 Handling Workloads with Dependence

As stated in Section 3.2, our primitives assume i.i.d., that insertions are independent, yet real-world workloads can have dependence between keys. A common scenario is using composite keys to describe multiple attributes of a single entity [26, 34]: [`book100|title`], [`book100|author`], [`book100|date`]. Related composite keys are often inserted together, resulting in dependent inserts.

Fortunately, we can treat these dependent inserts as independent if each insert is independent of a large number of (but not necessarily all) other inserts handled by the system. The dependence between a few inserts causes little effect on the overall compaction process because compaction involves many keys; for example, the compaction cost difference between inserting keys independently and inserting 10 related keys sharing the same key prefix as a batch is only about 0.2% on LevelDB when the workload contains 1 million or more total unique keys (for dependent inserts, 100,000 or more independent key groups, each of which has 10 related keys). Therefore, our primitives give good estimates in many practical scenarios which lack strictly independent inserts.

4 Modeling LevelDB

This section applies our analytic primitives to model a practical MSLS design, LevelDB. We explain how the dynamics of LevelDB components can be incorporated into the LevelDB model. We compare the analytic estimate with the measured performance of both a LevelDB simulator and the original implementation.

We assume that the dictionary-based compression [10, 17, 21] is not used in logs and SSTables. Using compression can reduce the write amplification (WA) by a certain factor; its effectiveness depends on how compressible the stored data is. Section 8 discusses how we handle variable-length items created as a result of compression.

Algorithm 1 summarizes the WA estimation for LevelDB. **unique**() and **merge**() calculate Unique and Merge as defined in Section 3. **dinterval**() calculates DInterval, defined in this section.

4.1 Logging

LevelDB's write-ahead logging (WAL) writes roughly the same amount as the data volume of inserts. We do not need to account for key redundancy because logging does not perform redundancy removal. As a consequence,

[4] Unique^{-1} is similar to, but differs from, the generalized coupon collector's problem (CCP) [16]. Generalized CCP terminates *as soon as* a certain number of unique items has been collected, whereas Unique^{-1} is merely defined as the inverse of Unique. Numerically, solutions of the generalized CCP are typically smaller than those of Unique^{-1} due to CCP's eager termination.

```
 1 // @param  L     maximum level
 2 // @param  wal   write-ahead log file size
 3 // @param  c0    level-0 SSTable count
 4 // @param  size  level sizes
 5 // @return       write amplification
 6 function estimateWA_LevelDB(L, wal, c0, size[]) {
 7   local l, WA, interval[], write[];
 8
 9   // mem -> log
10   WA = 1;
11
12   // mem -> level-0
13   WA += unique(wal) / wal;
14
15   // level-0 -> level-1
16   interval[0] = wal * c0;
17   write[1] = merge(unique(interval[0]), size[1]);
18   WA += write[1] / interval[0];
19
20   // level-l -> level-(l+1)
21   for (l = 1; l < L; l++) {
22     interval[l] = interval[l-1] + dinterval(size, l);
23     write[l+1] = merge(unique(interval[l]),
        size[l+1]) + unique(interval[l]);
24     WA += write[l+1] / interval[l];
25   }
26
27   return WA;
28 }
```

Algorithm 1: Pseudocode of a model of WA of LevelDB.

logging contributes 1 unit of WA (line #10). An advanced WAL scheme [9] can lower the logging cost below 1 unit.

4.2 Constructing Level-0 SSTables

LevelDB stores the contents of the memtable as a new SSTable in level-0 whenever the current log size reaches a threshold *wal*, which is 4 MiB by default.[5] Because an SSTable contains no redundant keys, we use Unique to compute the expected size of the SSTable corresponding to the accumulated requests; for every *wal* requests, LevelDB creates an SSTable of $\text{Unique}(wal)$, which adds $\text{Unique}(wal)/wal$ to WA (line #13).

4.3 Compaction

LevelDB compacts one or more SSTables in a level into the next level when any of the following conditions is satisfied: (1) level-0 has at least $c0$ SSTables; (2) the aggregate size of SSTables in a level-l ($1 \le l \le L$) reaches $\text{Size}(l)$ bytes; or (3) an SSTable has observed a certain number of seeks from query processing. The original LevelDB defines $c0 = 4$ SSTables[6] and $\text{Size}(l) = 10^l$ MiB. The level to compact is chosen based on the ratio of the current SSTable count or level size to the triggering condition, which can be approximated as prioritizing levels in their order from 0 to L in the model. The seek trigger depends on the distribution of queries as well as of insert requests, which is beyond the scope of this paper.

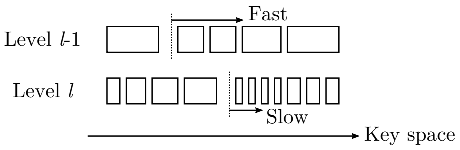


Figure 6: Non-uniformity of the key density caused by the different compaction speed of two adjacent levels in the key space. Each rectangle represents an SSTable. Vertical dotted lines indicate the last compacted key; the rectangles right next to the vertical lines will be chosen for compaction next time.

We examine two quantities to estimate the amortized compaction cost: a certain interval (a request count) that is large enough to capture the average compaction behavior of level-l, denoted as $\text{Interval}(l)$; and the expected amount of data written to level-$(l+1)$ during that interval, denoted as $\text{Write}(l+1)$. The contribution to WA by the compaction from level-l to level-$(l+1)$ is given by $\text{Write}(l+1)/\text{Interval}(l)$ by the definition of WA (line #18, #24).

4.3.1 Compacting Level-0 SSTables

LevelDB picks a level-0 SSTable and other level-0 SSTables that overlap with the first SSTable picked. It chooses overlapping level-1 SSTables as the other compaction input, and it can possibly choose more level-0 SSTables as long as the number of overlapping level-1 SSTables remains unchanged. Because level-0 contains overlapping SSTables with a wide key range, a single compaction commonly picks multiple level-0 SSTables; to build a concise model, we assume that all level-0 SSTables are chosen for compaction whenever the trigger for level-0 is met.

Let $\text{Interval}(0)$ be the interval of creating $c0$ SSTables, where $\text{Interval}(0) = wal \cdot c0$ (line #16). Compaction performed for that duration merges the SSTables created from $\text{Interval}(0)$ requests, which contain $\text{Unique}(\text{Interval}(0))$ unique keys, into level-1 with $\text{Size}(1)$ unique keys. Therefore, $\text{Write}(1) = \text{Merge}(\text{Unique}(\text{Interval}(0)), \text{Size}(1))$ (line #17).

4.3.2 Compacting Non-Level-0 SSTables

While the compaction from level-l to level-$(l+1)$ ($1 \le l < L$) follows similar rules as level-0 does, it is more complicated because of how LevelDB chooses the next SSTable to compact. LevelDB remembers $\text{LastKey}(l)$, the last key of the SSTable used in the last compaction for level-l and picks the first SSTable whose smallest key succeeds $\text{LastKey}(l)$; if there exists no such SSTable, LevelDB picks the SSTable with the smallest key in the level. This compaction strategy chooses SSTables in a circular way in the key space for each level.

Non-uniformity arises from round-robin compaction.

[5] We use the byte size and the item count interchangeably based on the assumption of fixed item size, as described in Section 3.2.

[6] LevelDB begins compaction with 4 level-0 SSTables, and new insert requests stall if the compaction of level-0 is not fast enough that the level-0 SSTable count reaches 12.

Compaction removes items from a level, but its effect is localized in the key space of that level. Compaction from a lower level into that level, however, tends to push items across the key space of the receiving level: the lower level makes faster progress compacting the entire key space because it contains fewer items, as depicted in Figure 6. As a result, the recently-compacted part of the key space has a lower chance of having items (low density), whereas the other part, which has not been compacted recently, is more likely to have items (high density). Because the maximum SSTable size is constrained, the low density area has SSTables covering a wide key range, and the high density area has SSTables with a narrow key range.

This non-uniformity makes compaction cheaper. Compaction occurs for an SSTable at the dense part of the key space. The narrow key range of the dense SSTable means a relatively small number of overlapping SSTables in the next level. Therefore, the compaction of the SSTable results in less data written to the next level.

Some LevelDB variants [22] explicitly pick an SSTable that maximizes the ratio of the size of that SSTable to the size of all overlapping SSTables in the next level, in hope of making the compaction cost smaller. Interestingly, due to the non-uniformity, LevelDB already *implicitly* realizes a similar compaction strategy. Our simulation results (not shown) indicate that the explicit SSTable selection brings a marginal performance gain over LevelDB's circular SSTable selection.

To quantify the effect of the non-uniformity to compaction, we model the density distribution of a level. Let $\mathrm{DInterval}(l)$ be the expected interval between compaction of the same key in level-l. This is also the interval to merge the level-l data into the entire key space of level-$(l+1)$. We use d to indicate the unidirectional distance from the most recently compacted key $\mathrm{LastKey}(l)$ to a key in the key space, where $0 \le d < N$. $d = 0$ represents the key just compacted, and $d = N-1$ is the key that will be compacted next time. Let $\mathrm{Density}(l,d)$ be the probability of having an item for the key with distance d in level-l. Because we assume no spatial key locality, we can formulate Density by approximating $\mathrm{LastKey}(l)$ to have a uniform distribution:

Theorem 3. Assuming $\mathrm{P}(\mathrm{LastKey}(l) = k) = 1/N$ for $1 \le l < L$, $k \in K$, then $\mathrm{Density}(l,d) = \mathrm{Unique}(\mathrm{DInterval}(l) \cdot d/N)/N$ for $1 \le l < L$, $0 \le d < N$.

We also use a general property of the density:

Lemma 2. $\sum_{d=0}^{N-1} \mathrm{Density}(l,d) = \mathrm{Size}(l)$ for $1 \le l < L$.

The value of $\mathrm{DInterval}(l)$ can be obtained by solving it numerically using Theorem 3 and Lemma 2.

We see that $\mathrm{DInterval}(l)$ is typically larger than $\mathrm{Unique}^{-1}(\mathrm{Size}(l))$ that represents the expected interval of compacting the same key *without* non-uniformity. For example, with $\mathrm{Size}(l) = 10$ Mi, $N = 100$ M (10^8), and a uniform key popularity distribution, $\mathrm{DInterval}(l)$ is at least twice as large as $\mathrm{Unique}^{-1}(\mathrm{Size}(l))$: 2.26×10^7 vs. 1.11×10^7. This confirms that non-uniformity does slow down the progression of $\mathrm{LastKey}(l)$, improving the efficiency of compaction.

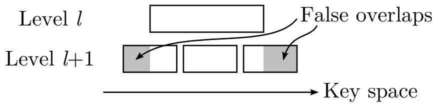


Figure 7: False overlaps that occur during the LevelDB compaction. Each rectangle indicates an SSTable; its width indicates the table's key range, not the byte size.

$\mathrm{Interval}(l)$, the actual interval we use to calculate the amortized WA, is cumulative and increases by DInterval, i.e., $\mathrm{Interval}(l) = \mathrm{Interval}(l-1) + \mathrm{DInterval}(l)$ (line #22). Because compacting lower levels is favored over compacting upper levels, an upper level may contain more data than its compaction trigger as an *overflow* from lower levels. We use a simple approximation to capture this behavior by adding the cumulative term $\mathrm{Interval}(l-1)$.

False overlaps are another effect caused by the incremental compaction using SSTables in LevelDB. Unlike non-uniformity, they increase the compaction cost slightly. For an SSTable being compacted, overlapping SSTables in the next level may contain items that lie outside the key range of the SSTable being compacted, as illustrated in Figure 7. Even though the LevelDB implementation attempts to reduce such false overlaps by choosing more SSTables in the lower level without creating new overlapping SSTables in the next level, false overlaps may add extra data writes whose size is close to that of the SSTables being compacted, i.e., $\mathrm{Unique}(\mathrm{Interval}(l))$ for $\mathrm{Interval}(l)$. Note that these extra data writes caused by false overlaps are more significant when Unique for the interval is large, i.e., under low skew, and they diminish as Unique becomes small, i.e., under high skew.

Several proposals [30, 41] strive to further reduce false overlaps by reusing a portion of input SSTables, essentially trading storage space and query performance for faster inserts. Such techniques can reduce WA by up to 1 per level, and even more if they address other types of false overlaps; the final cost savings, however, largely depend on the workload skew and the degree of the reuse.

By considering all of these factors, we can calculate the expected size of the written data. During $\mathrm{Interval}(l)$, level-l accepts $\mathrm{Unique}(\mathrm{Interval}(l))$ unique keys from the lower levels, which are merged into the next level containing $\mathrm{Size}(l+1)$ unique keys. False overlaps add extra writes roughly as much as the compacted level-l data. Thus, $\mathrm{Write}(l+1) = \mathrm{Merge}(\mathrm{Unique}(\mathrm{Interval}(l)), \mathrm{Size}(l+1)) + \mathrm{Unique}(\mathrm{Interval}(l))$ (line #23).

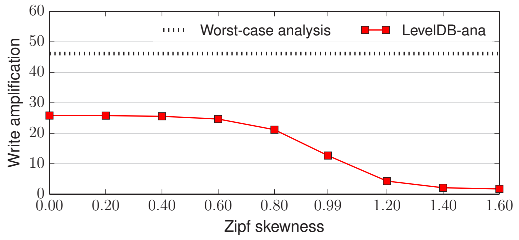


Figure 8: Effects of the workload skew on WA. Using 100 million unique keys, 1 kB item size.

4.4 Sensitivity to the Workload Skew

To examine how our LevelDB model reacts to the workload skew, we compare our WA estimates with worst-case analysis results. Our worst-case scenarios make the same assumption as prior asymptotic analyses [2, 36, 39], that the workload has no redundancy; therefore, merging two SSTables yields an SSTable whose size is exactly the same as the sum of the input SSTable sizes. In other words, compacting levels of size u and v results in $u+v$ items in the worst case.

Figure 8 plots the estimated WA for different Zipf skew parameters. Because our analytic model ("LevelDB-ana") considers the key popularity distribution of the workload in estimating WA, it clearly shows how WA decreases as LevelDB handles more skewed workloads; in contrast, the worst-case analysis ("Worst-case analysis") gives the same result regardless of the skew.

4.5 Comparisons with the Worst-Case Analysis, Simulation, and Experiment

We compare analytic estimates of WA given by our LevelDB model with the estimates given by the worst-case analysis, and the measured cost by running experiments on a LevelDB simulator and the original implementation.

We built a fast LevelDB simulator in C++ that follows the LevelDB design specification [20] to perform an item-level simulation and uses system parameters extracted from the LevelDB source code. This simulator does not intend to capture every detail of LevelDB implementation behaviors; instead, it realizes the high-level design components as explained in the LevelDB design document. The major differences are (1) our simulator runs in memory; (2) it performs compaction synchronously without concurrent request processing; and (3) it does not implement several opportunistic optimizations: (a) reducing false overlaps by choosing more SSTables in the lower level, (b) bypassing level-0 and level-1 for a newly created SSTable from the memtable if there are no overlapping SSTables in these levels, and (c) dynamically allowing more than 4 level-0 SSTables under high load.

For the measurement with the LevelDB implementation, we instrumented the LevelDB code (v1.18) to report the number of bytes written to disk via system calls. We use an item size that is 18 bytes smaller than we do in the analysis and simulation, to compensate for the increased data writes due to LevelDB's own storage space overhead. For fast experiments, we disable `fsync` and checksumming,[7] which showed no effects on WA in our experiments. We also avoid inserting items at an excessive rate that can overload level-0 with many SSTables and cause a high lookup cost.

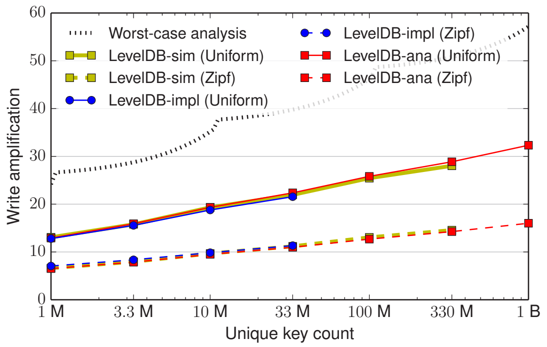


Figure 9: Comparison of WA between the estimation from our LevelDB model, the worst-case analysis, LevelDB simulation, and implementation results, with a varying number of total unique keys. Using 1 kB item size. Simulation and implementation results with a large number of unique keys are unavailable due to excessive runtime.

Both LevelDB simulator and implementation use a YCSB [11]-like workload generator written in C++. Each experiment initializes the system by inserting all keys once and then measures the average WA of executing random insert requests whose count is 10 times the total unique key count.

Figure 9 shows WA estimation and measurement with a varying number of total unique keys. Due to excessive experiment time, the graph excludes some data points for simulation ("LevelDB-sim") and implementation ("LevelDB-impl") with a large number of unique keys. The graph shows that our LevelDB model successfully estimates WA that agrees almost perfectly with the simulation and implementation results. The most significant difference occurs at 330 M unique keys with the uniform popularity distribution, where the estimated WA is only 3.0% higher than the measured WA. The standard worst-case analysis, however, significantly overestimates WA by 1.8–3.5X compared to the actual cost, which highlights the accuracy of our LevelDB model.

Figure 10 compares results with different *write buffer* size (i.e., the memtable size), which determines how much data in memory LevelDB accumulates to create a level-0 SSTable (and also affects how long crash recovery may

[7] MSLS implementations can use special CPU instructions to accelerate checksumming and avoid making it a performance bottleneck [23].

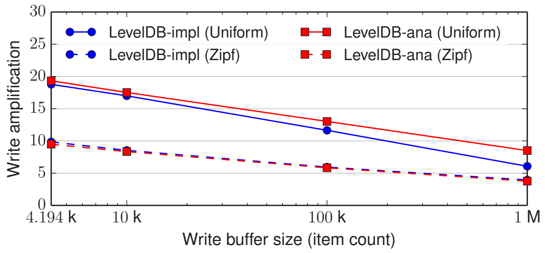


Figure 10: Comparison of WA between the estimation from our LevelDB model and implementation results, with varying write buffer sizes. Using 10 million unique keys, 1 kB item size.

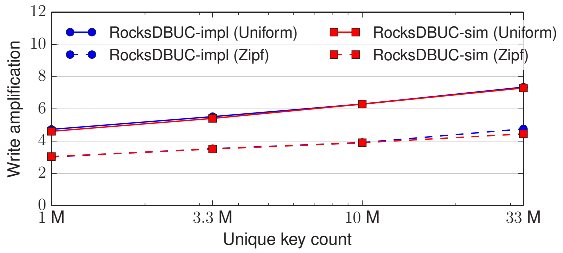


Figure 11: Comparison of WA between the estimation from our table-level simulation and implementation results for RocksDB's universal compaction, with a varying number of total unique keys. Using 1 kB item size.

take). In our LevelDB model, *wal* reflects the write buffer size. We use write buffer sizes between LevelDB's default size of 4 MiB and 10% of the last level size. The result indicates that our model estimates WA with good accuracy, but the error increases as the write buffer size increases for uniform key popularity distributions. We suspect that the error comes from the approximation in the model to take into account temporal overflows of levels beyond their maximum size; the error diminishes when level sizes are set to be at least as large as the write buffer size. In fact, avoiding too small level-1 and later levels has been suggested by RocksDB developers [14], and our optimization performed in Section 6 typically results in moderately large sizes for lower levels under uniform distributions, which makes this type of error insignificant for practical system parameter choices.

5 Modeling Universal Compaction

This section focuses on how we can model complex compaction strategies such as "universal compaction" implemented in RocksDB [15]. Section 7 revisits RocksDB to compare its "level style compaction" with LevelDB.

Universal compaction combines three small compaction strategies. RocksDB keeps a list of SSTables ordered by the age of their data, and compaction is restricted to adjacent tables. Compaction begins when the SSTable count exceeds a certain threshold (*Precondition*). First, RocksDB merges all SSTables whose total size minus the last one's size exceeds the last one's size by a certain factor (*Condition 1*); second, it merges consecutive SSTables that do not include a sudden increase in size beyond a certain factor (*Condition 2*); third, it merges the newest SSTables such that the total SSTable count drops below a certain threshold (*Condition 3*). Condition 1 avoids excessive duplicate data across SSTables, and Conditions 2 and 3 prevent high read amplification.

In such a multi-strategy system, it is difficult to determine how frequently each condition will cause compaction and what SSTables will be chosen for compaction.

We take this challenge as an opportunity to demonstrate how our analytic primitives are applicable to analyzing a complex system by using a *table-level simulation*. Unlike full simulators that keep track of individual items, a table-level simulator calculates only the SSTable size. It implements compaction conditions as the system design specifies, and it estimates the size of new or merged SSTables by using our analytic primitives. Dividing the total size of created SSTables by the total number of inserts gives the estimated WA. Unlike our LevelDB model that understands incremental compaction, a model for universal compaction does not need to consider non-uniformity and false overlaps. Interested readers may refer to Appendix C for the full pseudocode of the simulator.

Figure 11 compares WA obtained by our table-level simulation and the full RocksDB implementation. We use the default configuration, except for the SSTables count for compaction triggers set to 12. The simulation result ("RocksDBUC-sim") is close to the measured WA ("RocksDBUC-impl"). The estimated WA differs from the measured WA by up to 6.5% (the highest error with 33 M unique keys and skewed key inserts) though the overall accuracy remains as high as our LevelDB model presented in Section 4.

6 Optimizing System Parameters

Compared to full simulators and implementations, an analytic model offers fast estimation of cost metrics for a given set of system parameters. To demonstrate fast evaluation of the analytic model, we use an example of optimizing LevelDB system parameters to reduce WA using our LevelDB model.

Note that the same optimization effort could be made with the full LevelDB implementation by substituting our LevelDB model with the implementation and a synthetic workload generator. However, it would take prohibitively long to explore the large parameter space, as examined in Section 6.4.

6.1 Parameter Set to Optimize

The level sizes, Size(l), are important system parameters in LevelDB. They determine when LevelDB should initi-

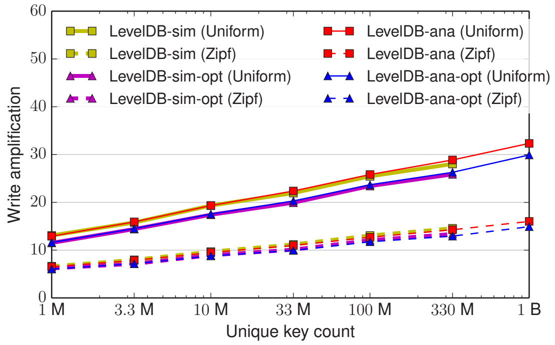


Figure 12: Improved WA using optimized level sizes on our analytic model and simulator for LevelDB.

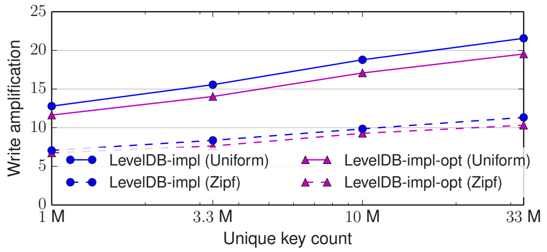


Figure 13: Improved WA using optimized level sizes on the LevelDB implementation.

ate compaction for standard levels and affect the overall compaction cost of the system. The original LevelDB design uses a geometric progression of $\text{Size}(l) = 10^l$ MiB. Interesting questions are (1) what level sizes different workloads favor; and (2) whether the geometric progression of level sizes is the optimal for all workloads.

Using different level sizes does not necessarily trade query performance or memory use. The log size, level-0 SSTable count, and total level count—the main determinants of query performance—are all unaffected by this system parameter.

6.2 Optimizer

We implemented a system parameter optimizer based on our analytic model. The objective function to minimize is the estimated WA. Input variables are $\text{Size}(l)$, excluding $\text{Size}(L)$, which will be equal to the total unique key count. After finishing the optimization, we use the new level sizes to obtain new WA estimates and measurement results on our analytic model and simulator. We also force the LevelDB implementation to use the new level sizes and measure WA. Our optimizer is written in Julia [4] and uses Ipopt [47] for nonlinear optimization. To speed up Unique, we use a compressed key popularity distribution which groups keys with similar probabilities and stores their average probability.[8]

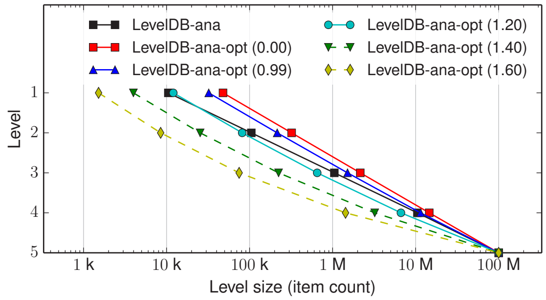


Figure 14: Original and optimized level sizes with varying Zipf skew. Using 100 million unique keys, 1 kB item size.

Source	Analysis		Simulation	
	No opt	Opt	No opt	Opt
mem→log	1.00	1.00	1.00	1.00
mem→level-0	1.00	1.00	1.00	1.00
level-0→1	1.62	3.85	1.60	3.75
level-1→2	4.77	4.85	4.38	4.49
level-2→3	6.22	4.82	6.04	4.66
level-3→4	6.32	4.65	6.12	4.58
level-4→5	4.89	3.50	5.31	3.93
Total	25.82	23.67	25.45	23.41

Table 1: Breakdown of WA sources on the analysis and simulation without and with the level size optimization. Using 100 million unique keys, 1 kB item size, and a uniform key popularity distribution.

6.3 Optimization Results

Our level size optimization successfully reduces the insert cost of LevelDB. Figures 12 and 13 plot WA with optimized level sizes. Both graphs show that the optimization ("LevelDB-ana-opt," "LevelDB-sim-opt," and "LevelDB-impl-opt") improves WA by up to 9.4%. The analytic estimates and simulation results agree with each other as before, and the LevelDB implementation exhibits lower WA across all unique key counts.

The optimization is effective because level sizes differ by the workload skew, as shown in Figure 14. Having larger lower levels is beneficial for relatively low skew as it reduces the size ratio of adjacent levels. On the other hand, high skew favors smaller lower levels and level sizes that grow faster than the standard geometric progression. With high skew, compaction happens more frequently in the lower levels to remove redundant keys; keeping these levels small reduces the cost of compaction. This result suggests that it is suboptimal to use fixed level sizes for different workloads and that using a geometric progression of level sizes is not always the best design to minimize WA.

[8] For robustness, we optimize using both the primal and a dual form of the LevelDB model presented in Section 4. The primal optimizes over $\text{Size}(l)$ and the dual optimizes over $\text{Unique}^{-1}(\text{Size}(l))$. We pick the result of whichever model produces the smaller WA.

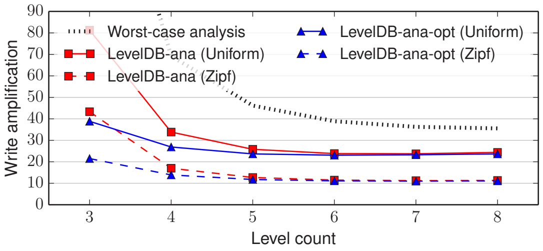


Figure 15: WA using varying numbers of levels. The level count excludes level-0. Using 100 million unique keys, 1 kB item size.

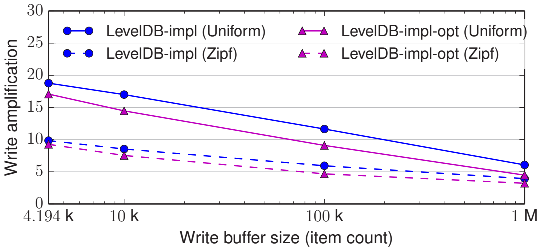


Figure 16: Improved WA using optimized level sizes on the LevelDB implementation, with a large write buffer. Using 10 million unique keys, 1 kB item size.

Table 1 further examines how the optimization affects per-level insert costs, using the LevelDB model and simulation. Per-level WA tends to be more variable using the original level sizes, while the optimization makes them relatively even across levels except the last level. This result suggests that it may be worth performing a runtime optimization that dynamically adjusts level sizes to achieve the lower overall WA by reducing the variance of the per-level WA.

By lowering WA, the system can use fewer levels to achieve faster lookup speed without significant impact on insert costs. Figure 15 reveals how much extra room for query processing the optimization can create. This analysis changes the level count by altering the growth factor of LevelDB, i.e., using a higher growth factor for a lower level count. The result shows that the optimization is particularly effective with a fewer number of levels, and it can save almost a whole level's worth of WA compared to using a fixed growth factor. For example, with the optimized level sizes, a system can use 3 levels instead of 4 levels without incurring excessively high insert costs.

A LevelDB system with large memory can further benefit from our level size optimization. Figure 16 shows the result of applying the optimization to the LevelDB implementation, with a large write buffer. The improvement becomes more significant as the write buffer size increases, reaching 26.2% of WA reduction at the buffer size of 1 million items.

6.4 Optimizer Performance

The level size optimization requires little time due to the fast evaluation of our analytic model. For 100 million unique keys with a uniform key popularity distribution, the entire optimization took 2.63 seconds, evaluating 17,391 different parameter sets (6,613 evaluations per second) on a server-class machine equipped with Intel® Xeon® E5-2680 v2 processors. For the same-sized workload, but with Zipf skew of 0.99, the optimization time increased to 79 seconds, which is far more than the uniform case, but is less than 2 minutes; for this optimization, the model was evaluated 16,680 times before convergence (211 evaluations per second).

Evaluating this many system parameters using a full implementation—or even item-level simulation—is prohibitively expensive. Using the same hardware as above, our in-memory LevelDB simulator takes 45 minutes to measure WA for a *single* set of system parameters with 100 million unique keys. The full LevelDB implementation takes 101 minutes (without `fsync`) to 490 minutes (with `fsync`), for a smaller dataset with 10 million unique keys.

7 Improving RocksDB

In this section, we turn our attention to RocksDB [12], a well-known variant of LevelDB. RocksDB offers improved capabilities and multithreaded performance, and provides an extensive set of system configurations to temporarily accelerate bulk loading by sacrificing query performance or relaxing durability guarantees [13, 14]; nevertheless, there have been few studies of how RocksDB's *design* affects its performance. We use RocksDB v4.0 and apply the same set of instrumentation, configuration, and workload generation as we do to LevelDB.

RocksDB supports "level style compaction" that is similar to LevelDB's data layout, but differs in how it picks the next SSTable to compact. RocksDB picks the largest SSTable in a level for compaction,[9] rather than keeping LevelDB's round-robin SSTable selection. We learned in Section 4, however, that LevelDB's compaction strategy is effective in reducing WA because it tends to pick SSTables that overlap a relatively small number of SSTables in the next level.

To compare the compaction strategies used by LevelDB and RocksDB, we measure the insert cost of both systems in Figure 17. Unfortunately, the current RocksDB strategy produces higher WA ("RocksDB-impl") than LevelDB does ("LevelDB-impl"). In theory, the RocksDB approach may help multithreaded compaction because large tables may be spread over the entire key space so that they facilitate parallel compaction; this effect, how-

[9] Recent versions of RocksDB support additional strategies for SSTable selection.

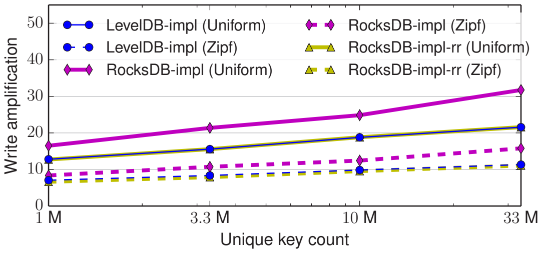


Figure 17: Comparison of WA between LevelDB, RocksDB, and a modified RocksDB with LevelDB-like SSTable selection.

ever, was not evident in our experiments using multiple threads. The high insert cost of RocksDB is entirely caused by RocksDB's compaction strategy; implementing LevelDB's SSTable selection in RocksDB ("RocksDB-impl-rr") reduces RocksDB's WA by up to 32.0%, making it comparable to LevelDB's WA. This result confirms that LevelDB's strategy is good at reducing WA as our analytic model predicts.

We have not found a scenario where RocksDB's current strategy excels, though some combinations of workloads and situations may favor it. LevelDB and RocksDB developers may or may have not intended any of the effects on WA when designing their systems. Either way, our analytic model provides quantitative evidence that LevelDB's table selection will perform well under a wide range of workloads despite being the "conventional" solution.

8 Estimating Read Amplification

This section presents read amplification (RA) estimation.

We introduce a *weighted* variant of our analytic primitives. A per-key weight w, which is nontrivial (i.e., $w(k) \neq 0$ for some k) and nonnegative, specifies how much contribution each key makes to the result:

Definition 3.
$\text{Unique}(p,w) := \sum_{k \in K} [1-(1-f_X(k))^p]\, w(k)$ for $p \geq 0$.

We construct $w(k)$ to indicate the probability of having key k for each query. For level-l, let $s(l)$ and $e(l)$ be the expected age of the newest and oldest item in level-l in terms of the number of inserts, obtained by using the system model presented in Section 4. We find $c(l)$, the expected I/O cost to perform a query at level-l. The expected I/O cost to perform queries that finish at level-l is $[\text{Unique}(e(l),w) - \text{Unique}(s(l),w)] \cdot c(l)$. Adding the expected I/O cost of each level gives the overall RA.

As another use case, the weighted variant can add support for *variable-length* items to the system models presented in Sections 4 and 5. By setting $w(k)$ to the size of the item for key k, Unique returns the expected size of unique items instead of their expected count. Because weighted Unique is still strictly monotonic, weighted Unique^{-1} and Merge exist.

9 Discussion

Analyzing an MSLS design with an accurate model can provide useful insights on how one should design a new MSLS to exploit opportunities provided by workloads. For example, our analytic model reveals that LevelDB's byte size-based compaction trigger makes compaction much less frequent and less costly under skew; such a design choice should be suitable for many real-world workloads with skew [1].

A design process complemented with accurate analysis can help avoid false conclusions about a design's performance. LevelDB's per-level WA is less (only up to 4–6) than assumed in the worst case (11–12 for a growth factor of 10), even for uniform workloads. Our analytical model justifies LevelDB's high growth factor, which turns out to be less harmful for insert performance than standard worst-case analysis implies.

Our analytic primitives and modeling are not without limitations. Assumptions such as independence and no spatial locality in requested keys may not hold if there are dependent keys that share the same prefix though a small amount of such dependence does not change the overall system behavior and thus can be ignored as discussed in Section 3.5. Our modeling in Section 4 does not account for time-varying workload characteristics (e.g., flash crowds) or special item types such as tombstones that represent item deletion, while the simulation-oriented modeling in Section 5 can handle such cases. We leave extending our primitives further to accommodate remaining cases as future work.

Both design and implementation influence the final system performance. Our primitives and modeling are useful for understanding the *design* of MSLS systems. Although we use precise metrics such as WA to describe the system performance throughout this work, these metrics are ultimately not identical to implementation-level metrics such as operations per second. Translating a good system design into an efficient implementation is critical to achieving good performance, and remains a challenging and important goal for system developers and researchers.

10 Related Work

Over the past decade, numerous studies have proposed new multi-stage log-structured (MSLS) designs and evaluated their performance. In almost every case, the authors present implementation-level performance [2, 12, 18, 19, 22, 29, 30, 32, 39, 40, 41, 42, 43, 46, 48]. Some employ analytic metrics such as write amplification to explain the design rationale, facilitate design comparisons, and generalize experiment results [12, 22, 29, 30, 32, 39, 43], and most of the others also use the concept of per-operation costs. However, they eventually rely on the experimental measurement because their analysis fails to offer sufficiently high accuracy to make meaningful performance comparisons. LSM-tree [36], LHAM [33], COLA [2], bLSM [39], and B-tree variants [6, 24] provide extensive analysis on their design, but their analyses are limited to asymptotic complexities or always assume the worst case.

Despite such a large number of MSLS design proposals, there is little active research to devise improved evaluation methods for these proposals to fill the gap between asymptotic analysis and experimental measurement. The sole existing effort is limited to a specific system design [31], but does not provide general-purpose primitives. We are unaware of prior studies that successfully capture workload skew and the dynamics of compaction to the degree that the estimates are close to simulation and implementation results, as we present in this paper.

11 Conclusion

We present new analytic primitives for modeling multistage log-structured (MSLS) designs, which can quickly and accurately estimate their performance. We have presented a model for the popular LevelDB system, which estimates write amplification very close to experimentally determined actual costs; using this model, we were able to find more favorable system parameters that reduce the overall cost of writes. Based upon lessons learned from the model, we propose changes to RocksDB to lower its insert costs. We believe that our analytic primitives and modeling method are applicable to a wide range of MSLS designs and performance metrics. The insights derived from the models facilitate comparisons of MSLS designs and ultimately help develop new designs that better exploit workload characteristics to improve performance.

Acknowledgments

This work was supported by funding from National Science Foundation under awards IIS-1409802 and CNS-1345305, and Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). We would like to thank anonymous FAST reviewers for their feedback and James Mickens for shepherding this paper. We appreciate Eddie Kohler, Mark Callaghan, Kai Ren, and anonymous SOSP reviewers for their comments on early versions of this work.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, June 2012.

[2] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 2007.

[3] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformations. *Journal of Algorithms*, 1(4):301–358, Dec. 1980.

[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. Dec. 2014. http://arxiv.org/abs/1411.1607.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[7] M. Callaghan. Read Amplification Factor. http://mysqlha.blogspot.com/2011/08/read-amplification-factor.html, 2011.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. 7th USENIX OSDI*, Nov. 2006.

[9] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[10] Y. Collet. LZ4. https://github.com/Cyan4973/lz4, 2015.

[11] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, June 2010.

[12] Facebook. RocksDB. http://rocksdb.org/, 2015.

[13] Facebook. Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks, 2014.

[14] Facebook. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide, 2015.

[15] Facebook. RocksDB Universal Compaction. https://github.com/facebook/rocksdb/wiki/Universal-Compaction, 2015.

[16] P. Flajolet, D. Gardy, and L. Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3), Nov. 1992.

[17] J.-L. Gailly and M. Adler. zlib. http://www.zlib.net/, 2013.

[18] M. Ghosh, I. Gupta, S. Gupta, and N. Kumar. Fast compaction algorithms for NoSQL databases. In *Proc. the 35th IEEE International Conference on Distributed Computing Systems (ICDCS)*, June 2015.

[19] Google. LevelDB. https://github.com/google/leveldb, 2014.

[20] Google. LevelDB file layout and compactions. https://github.com/google/leveldb/blob/master/doc/impl.html, 2014.

[21] Google. Snappy. https://github.com/google/snappy, 2015.

[22] HyperDex. HyperLevelDB. http://hyperdex.org/performance/leveldb/, 2013.

[23] Intel SSE4 programming reference. https://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf, 2007.

[24] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. BetrFS: A right-optimized write-optimized file system. In *Proc. 13th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2015.

[25] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 2011.

[26] B. Kate, E. Kohler, M. S. Kester, N. Narula, Y. Mao, and R. Morris. Easy freshness with Pequod cache joins. In *Proc. 11th USENIX NSDI*, Apr. 2014.

[27] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating System Review*, 44:35–40, Apr. 2010.

[28] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6), June 1983.

[29] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[30] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami. NVMKV: A scalable, lightweight, FTL-aware key-value store. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, July 2015.

[31] P. Menon, T. Rabl, M. Sadoghi, and H.-A. Jacobsen. Optimizing key-value stores for hybrid storage architectures. In *Proceedings of CASCON*, 2014.

[32] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom. Lightweight application-level crash consistency on transactional flash storage. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, July 2015.

[33] P. Muth, P. O'Neil, A. Pick, and G. Weikum. The LHAM log-structured history data access method. *The VLDB Journal*, 8(3-4):199–221, Feb. 2000.

[34] NoSQL data modeling techniques. https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/, 2012.

[35] P. E. O'Neil. The SB-tree: An index-sequential structure for high-performance sequential access. *Acta Inf.*, 29(3):241–265, 1992.

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

[37] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[38] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proc. 5th ACM Symposium on Cloud Computing (SOCC)*, Nov. 2014.

[39] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[40] R. Sears, M. Callaghan, and E. Brewer. Rose: Compressed, log-structured replication. *Proc. VLDB Endowment*, 1(1), Aug. 2008.

[41] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building workload-independent storage with VT-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[42] R. P. Spillane, P. J. Shetty, E. Zadok, S. Dixit, and S. Archak. An efficient multi-tier tablet server storage architecture. In *Proc. 2nd ACM Symposium on Cloud Computing (SOCC)*, Oct. 2011.

[43] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *Proc. 13th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2015.

[44] The Apache Software Foundation. Apache Cassandra. https://cassandra.apache.org/, 2015.

[45] The Apache Software Foundation. Apache HBase. https://hbase.apache.org/, 2015.

[46] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. LogBase: A scalable log-structured database system in the cloud. *Proc. VLDB Endowment*, 5(10), June 2012.

[47] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm

for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

[48] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[49] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. Characterizing storage workloads with counter stacks. In *Proc. 11th USENIX OSDI*, Oct. 2014.

A Proofs

This section provides proofs for theorems presented in this paper.

Theorem 1. *(in Section 3.3)* $\mathrm{Unique}(p)$ *is the expected number of unique keys that appear in p requests.*

Proof. Key k counts towards the unique key count if k appears at least once in a sequence of p requests, whose probability is $1-(1-f_X(k))^p$. Therefore, the expected unique key count is $\sum_{k \in K}(1-(1-f_X(k))^p) = N - \sum_{k \in K}(1-f_X(k))^p = \mathrm{Unique}(p)$. □

Lemma 1. *(in Section 3.3)* $\mathrm{Unique}^{-1}(u)$ exists for $0 \leq u < N$.

Proof. Suppose $0 \leq p < q$. $(1-f_X(k))^q < (1-f_X(k))^p$ because $0 < 1-f_X(k) < 1$. $\mathrm{Unique}(q) - \mathrm{Unique}(p) = -\sum_{k \in K}(1-f_X(k))^q + \sum_{k \in K}(1-f_X(k))^p > 0$. Thus, Unique is a strictly monotonic function that is defined over $[0, N)$. □

Theorem 2. *(in Section 3.4)* $\mathrm{Merge}(u, v)$ *is the expected size of a merged table that is created from two tables whose size is u and v.*

Proof. Let p and q be the expected numbers of insert requests that would produce tables of size u and v, respectively. The merged table is expected to contain all $k \in K$ except those missing in both request sequences. Therefore, the expected merged table size is $N - \sum_{k \in K}(1-f_X(k))^p(1-f_X(k))^q = \mathrm{Unique}(p+q)$. Because $p = \mathrm{Unique}^{-1}(u)$ and $q = \mathrm{Unique}^{-1}(v)$, $\mathrm{Unique}(\mathrm{Unique}^{-1}(u) + \mathrm{Unique}^{-1}(v)) = \mathrm{Merge}(u, v)$. □

Theorem 3. *(in Section 4.3)* Assuming $\mathrm{P}(\mathrm{LastKey}(l) = k) = 1/N$ for $1 \leq l < L$, $k \in K$, then $\mathrm{Density}(l, d) = \mathrm{Unique}(\mathrm{DInterval}(l) \cdot d/N)/N$ for $1 \leq l < L$, $0 \leq d < N$.

Proof. Suppose $\mathrm{LastKey}(l) = k \in K$. Let k' be $(k - d + N) \bmod N$. Let r be $\mathrm{DInterval}(l) \cdot d/N$. There are r requests since the last compaction of k'. Level-l has k' if any of r requests contains k', whose probability is $1-(1-f_X(k'))^r$.

By considering all possible k and thus all possible k', $\mathrm{Density}(l, d) = \sum_{k \in K}(1/N)(1-(1-f_X(k))^r) = \mathrm{Unique}(\mathrm{DInterval}(l) \cdot d/N)/N$. □

Lemma 2. *(in Section 4.3)* $\sum_{d=0}^{N-1} \mathrm{Density}(l, d) = \mathrm{Size}(l)$ for $1 \leq l < L$.

Proof. The sum over the density equals to the expected unique key count, which is the number of keys level-l maintains, i.e., $\mathrm{Size}(l)$. □

Theorem 4. *(in Section 8) The expected I/O cost to perform queries that finishes at level-l is given by* $[\mathrm{Unique}(e(l), w) - \mathrm{Unique}(s(l), w)] \cdot c(l)$*, where w describes the query distribution and* $c(l)$ *is the expected I/O cost to perform a query at level-l.*

Proof. For key k to exist in level-l and be used for query processing (without being served in an earlier level), it must appear in at least one of $e(l) - s(l)$ requests and in none of other $s(l)$ requests. The first condition ensures the existence of the key in level-l, and the second condition rejects the existence of the key in an earlier level (otherwise, queries for key k will be served in that level). Thus, the probability of such a case is $(1-(1-f_X(k))^{e(l)-s(l)}) \cdot (1-f_X(k))^{s(l)} = (1-f_X(k))^{s(l)} - (1-f_X(k))^{e(l)}$.

The expected I/O cost to perform a query for key k that finishes at level-l is $\left[(1-f_X(k))^{s(l)} - (1-f_X(k))^{e(l)}\right] \cdot c(l)$.

Because the fraction of the queries for key k among all queries is given by $w(k)$, the expected I/O cost to perform queries that finishes at level-l is $\sum\limits_{k \in K}\left[(1-f_X(k))^{s(l)} - (1-f_X(k))^{e(l)}\right] c(l) w(k) = \sum\limits_{k \in K}\left[\Big((1-(1-f_X(k))^{e(l)}\Big) - \Big(1-(1-f_X(k))^{s(l)}\Big)\right] w(k)$ $c(l) = [\mathrm{Unique}(e(l), w) - \mathrm{Unique}(s(l), w)] \cdot c(l)$. □

B Modeling COLA and SAMT

The cache-oblivious lookahead array (COLA) [2] is a generalized and improved binomial list [3]. Like LSM-tree, COLA has multiple levels whose count is $\lceil \log_r N \rceil$, where r is the growth factor. Each level contains zero or one SSTable. Unlike LSM-tree, however, COLA uses the merge count as the main compaction criterion; a level in COLA accepts $r-1$ merges with the lower level before the level is merged into the next level.

COLA has roughly similar asymptotic complexities to LSM-tree's. A query in COLA may cost $O(\log_r N)$ random I/O per lookup if looking up a level costs $O(1)$ random I/O. COLA's data migration costs $O((r-1)\log_r N)$ I/O per insert. r is usually chosen between 2 and 4.

The Sorted Array Merge Tree (SAMT) [42] is similar to COLA but performs compaction differently. Instead of eagerly merging data to have a single log structure per level, SAMT keeps up to r SSTables before merging them and moving the merged data into the next level. Therefore, a lookup costs $O(r \log_r N)$ random I/O, whereas the per-update I/O cost decreases to $O(\log_r N)$.

A few notable systems implementing a version of COLA and SAMT are HBase [45] and Cassandra [27, 44].

Algorithm 2 presents models for COLA and SAMT. Both models assume that the system uses write-ahead log files whose count is capped by the growth factor r.

```
 1 // @param  L    maximum level
 2 // @param  wal  write-ahead log file size
 3 // @param  r    growth factor
 4 // @return      write amplification
 5 function estimateWA_COLA(L, wal, r) {
 6   local l, j, WA, interval[], write[];
 7   // mem -> log
 8   WA = 1;
 9   // mem -> level-1; level-l -> level-(l+1)
10   interval[0] = wal;
11   for (l = 0; l < L - 1; l++) {
12     interval[l + 1] = interval[l] * r;
13     write[l + 1] = 0;
14     for (j = 0; j < r - 1; j++)
15       write[l + 1] += merge(unique(interval[l]),
        unique(interval[l] * j));
16     WA += write[l + 1] / interval[l + 1];
17   }
18   // level-(L-1) -> level-L
19   WA += unique(∞) / interval[L - 1];
20   return WA;
21 }
22
23 function estimateWA_SAMT(L, wal, r) {
24   local l, WA, interval[], write[];
25   // mem -> log
26   WA = 1;
27   // mem -> level-1; level-l -> level-(l+1)
28   interval[0] = wal;
29   for (l = 0; l < L - 1; l++) {
30     interval[l + 1] = interval[l] * r;
31     write[l + 1] = r * unique(interval[l]);
32     WA += write[l + 1] / interval[l + 1];
33   }
34   // level-(L-1) -> level-L
35   WA += unique(∞) / interval[L - 1];
36   return WA;
37 }
```

Algorithm 2: Pseudocode of models of WA of COLA and SAMT.

In COLA, line #15 calculates the amount of writes for a level that has already accepted j merges ($0 \le j < r-1$). Compaction of the second-to-last level is treated specially because the last level must be large enough to hold all unique keys and has no subsequent level (line #19). The SAMT model is simpler because it defers merging the data in the same level.

C Modeling Universal Compaction

Algorithm 3 models RocksDB's universal compaction using a table-level simulation presented in Section 5. Line #15 estimates the size of a new SSTable created from insert requests. Line #26, #43, and #55 predict the outcome of SSTable merges caused of different compaction triggers.

merge_all() takes a list of (multiple) SSTable sizes and returns the expected size of the merge result (i.e., $\mathrm{Unique}(\sum_i \mathrm{Unique}^{-1}(\mathrm{sizes}[i]))$).

```
// @param  wal                                 write-ahead log file size
// @param  level0_file_num_compaction_trigger  number of files to trigger compaction
// @param  level0_stop_writes_trigger          maximum number of files
// @param  max_size_amplification_percent      parameter for Condition 1
// @param  size_ratio                          parameter for Condition 2
// @param  tables                              list of initial SSTable sizes
// @param  num_inserts                         number of inserts to simulate
// @return                                     write amplification
function estimateWA_UC(wal, level0_file_num_compaction_trigger, level0_stop_writes_trigger,
     max_size_amplification_percent, size_ratio, tables, num_inserts) {
  local inserts, writes, done, last, start_i, last_i, i, candidate_count, candidate_size, table_size;
  inserts = writes = 0;
  while (inserts < num_inserts) {
    if (len(tables) < level0_stop_writes_trigger) {
      // a new SSTable
      table_size = unique(wal);
      writes += wal;                   // mem -> log
      writes += table_size;            // mem -> level-0
      inserts += wal;
      tables = [table_size] + tables;
    }
    // Precondition
    if (len(tables) >= level0_file_num_compaction_trigger) {
      last = len(tables) - 1;
      // Condition 1
      if (sum(tables[0...last-1]) / tables[last] > max_size_amplification_percent / 100) {
        table_size = merge_all(tables);
        tables = [table_size];
        writes += table_size;          // level-0 -> level-0
      } else {
        done = false;
        // Condition 2
        for (start_i = 0; start_i < len(tables); start_i++) {
          candidate_count = 1;
          candidate_size = tables[start_i];
          for (i = start_i + 1; i < len(tables); i++) {
            if (candidate_size * (100 + size_ratio) / 100 < tables[i])
              break;
            candidate_size += tables[i];
            candidate_count++;
          }
          if (candidate_count >= 2) {
            last_i = start_i + candidate_count - 1;
            table_size = merge_all(tables[start_i...last_i]);
            tables = tables[0...start_i-1] + [table_size] + tables[last_i+1...last]);
            writes += table_size;      // level-0 -> level-0
            done = true;
            break;
          }
        }
        // Condition 3
        if (done == false) {
          candidate_count = len(tables) - level0_file_num_compaction_trigger;
          if (candidate_count >= 2) {
            last_i = candidate_count - 1;
            table_size = merge_all(tables[0...last_i]);
            tables = [table_size] + tables[last_i+1...last];
            writes += table_size;      // level-0 -> level-0
          }
        }
      }
    }
  }
  return writes / inserts;
}
```

Algorithm 3: Pseudocode of a table-level simulation of RocksDB's universal compaction.

Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication

Heng Zhang, Mingkai Dong, Haibo Chen*
Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

ABSTRACT

In-memory key/value store (KV-store) is a key building block for many systems like databases and large websites. Two key requirements for such systems are efficiency and availability, which demand a KV-store to continuously handle millions of requests per second. A common approach to availability is using replication such as primary-backup (PBR), which, however, requires $M+1$ times memory to tolerate M failures. This renders scarce memory unable to handle useful user jobs.

This paper makes the first case of building highly available in-memory KV-store by integrating erasure coding to achieve memory efficiency, while not notably degrading performance. A main challenge is that an in-memory KV-store has much scattered metadata. A single KV *put* may cause excessive coding operations and parity updates due to numerous small updates to metadata. Our approach, namely Cocytus, addresses this challenge by using a hybrid scheme that leverages PBR for small-sized and scattered data (e.g., metadata and key), while only applying erasure coding to relatively large data (e.g., value). To mitigate well-known issues like lengthy recovery of erasure coding, Cocytus uses an online recovery scheme by leveraging the replicated metadata information to continuously serving KV requests. We have applied Cocytus to Memcached. Evaluation using YCSB with different KV configurations shows that Cocytus incurs low overhead for latency and throughput, can tolerate node failures with fast online recovery, yet saves 33% to 46% memory compared to PBR when tolerating two failures.

1 INTRODUCTION

The increasing demand of large-scale Web applications has stimulated the paradigm of placing large datasets within memory to satisfy millions of operations per second with sub-millisecond latency. This new computing model, namely in-memory computing, has been emerging recently. For example, large-scale in-memory key/value systems like Memcached [13] and Redis [47] have been widely used in Facebook [24], Twitter [38] and LinkedIn. There have also been considerable interests of applying in-memory databases (IMDBs) to performance-hungry scenarios (e.g., SAP HANA [12], Oracle TimesTen [18] and Microsoft Hekaton [9]).

Even if many systems have a persistent backing store to preserve data durability after a crash, it is still important to retain data in memory for instantaneously taking over the job of a failed node, as rebuilding terabytes of data into memory is time-consuming. For example, it was reported that recovering around 120 GB data from disk to memory for an in-memory database in Facebook took 2.5-3 hours [14]. Traditional ways of providing high availability are through replication such as standard primary-backup (PBR) [5] and chain-replication [39], by which a dataset is replicated $M+1$ times to tolerate M failures. However, this also means dedicating M copies of CPU/memory without producing user work, requiring more standby machines and thus multiplying energy consumption.

This paper describes Cocytus, an efficient and available in-memory replication scheme that is strongly consistent. Cocytus aims at reducing the memory consumption for replicas while keeping similar performance and availability of PBR-like solutions, though at additional CPU cost for update-intensive workloads. The key of Cocytus is efficiently combining the space-efficient erasure coding scheme with the PBR.

Erasure coding is a space-efficient solution for data replication, which is widely applied in distributed storage systems, including Windows Azure Store [15] and Facebook storage [23]. However, though space-efficient, erasure coding is well-known for its lengthy recovery and transient data unavailability [15, 34].

In this paper, we investigate the feasibility of applying erasure coding to in-memory key/value stores (KV-stores). Our main observation is that the abundant and speedy CPU cores make it possible to perform coding online. For example, a single Intel Xeon E3-1230v3 CPU core can encode data at 5.26GB/s for Reed-Solomon(3,5) codes, which is faster than even current high-end NIC with 40Gb/s bandwidth. However, the block-oriented nature of erasure coding and the unique feature of KV-stores raise several challenges to Cocytus to meet the goals of efficiency and availability.

[*] Corresponding author

The first challenge is that the scattered metadata like a hashtable and the memory allocation information of a KV-store will incur a large number of coding operations and updates even for a single KV put. This incurs not only much CPU overhead but also high network traffic. Cocytus addresses this issue by leveraging the idea of separating metadata from data [42] and uses a hybrid replication scheme. In particular, Cocytus uses erasure coding for application data while using PBR for small-sized metadata.

The second challenge is how to consistently recover lost data blocks online with the distributed data blocks and parity blocks[1]. Cocytus introduces a distributed online recovery protocol that consistently collects all data blocks and parity blocks to recover lost data, yet without blocking services on live data blocks and with predictable memory.

We have implemented Cocytus in Memcached 1.4.21 with the synchronous model, in which a server sends responses to clients after receiving the acknowledgments from backup nodes to avoid data loss. We also implemented a pure primary-backup replication in Memcached 1.4.21 for comparison. By using YCSB [8] to issue requests with different key/value distribution, we show that Cocytus incurs little degradation on throughput and latency during normal processing and can gracefully recover data quickly. Overall, Cocytus has high memory efficiency while incurring small overhead compared with PBR, yet at little CPU cost for read-mostly workloads and modest CPU cost for update-intensive workloads.

In summary, the main contribution of this paper includes:

- The first case of exploiting erasure coding for in-memory KV-store.
- Two key designs, including a hybrid replication scheme and distributed online recovery that achieve efficiency, availability and consistency.
- An implementation of Cocytus on Memcached [13] and a thorough evaluation that confirms Cocytus's efficiency and availability.

The rest of this paper is organized as follows. The next section describes necessary background information about primary-backup replication and erasure coding on a modern computing environment. Section 3 describes the design of Cocytus, followed up by the recovery process in section 4. Section 5 describes the implementation details. Section 6 presents the experimental data of Cocytus. Finally, section 7 discusses related work, and section 8 concludes this paper.

2 Background and Challenges

This section first briefly reviews primary-backup replication (PBR) and erasure coding, and then identifies opportunities and challenges of applying erasure coding to in-memory KV-stores.

2.1 Background

Primary-backup replication: Primary-backup replication (PBR) [3] is a widely-used approach to providing high availability. As shown in Figure 1(a), each primary node has M backup nodes to store its data replicas to tolerate M failures. One of the backup nodes would act as the new primary node if the primary node failed, resulting in a *view change* (e.g., using Paxos [19]). As a result, the system can still provide continuous services upon node failures. This, however, is at the cost of high data redundancy, e.g., M additional storage nodes and the corresponding CPUs to tolerate M failures. For example, to tolerate two node failures, the storage efficiency of a KV-store can only reach 33%.

Erasure coding: Erasure coding is an efficient way to provide data durability. As shown in Figure 1(b), with erasure coding, an N-node cluster can use K of N nodes for data and M nodes for parity ($K + M = N$). A commonly used coding scheme is Reed-Solomon codes (RS-code) [30], which computes parities according to its data over a finite field by the following formula (the matrix is called a *Vandermonde matrix*):

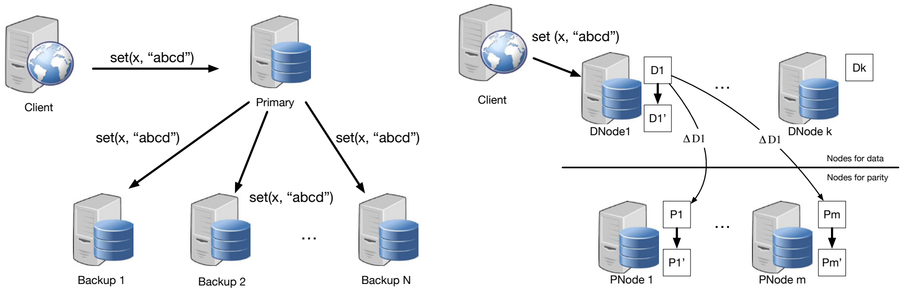
$$\begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_M \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & a_1^1 & \cdots & a_1^{K-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & a_{M-1}^1 & \cdots & a_{M-1}^{K-1} \end{bmatrix} * \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_K \end{bmatrix} \qquad (1)$$

An update on a DNode (a node for data) can be achieved by broadcasting its delta to all PNodes (nodes for parity) and asking them to add the delta to parity with a predefined coefficient. This approach works similarly for updating any parity blocks; its correctness can be proven by the following equation, where A represents the *Vandermonde matrix* mentioned in formula (1).

$$\begin{bmatrix} P_1^{'} \\ \vdots \\ P_i^{'} \\ \vdots \\ P_M^{'} \end{bmatrix} = A * \begin{bmatrix} D_1 \\ \vdots \\ D_i^{'} \\ \vdots \\ D_K \end{bmatrix} = A * \begin{bmatrix} D_1 \\ \vdots \\ D_i + \Delta D_i \\ \vdots \\ D_K \end{bmatrix} = \begin{bmatrix} P_1 \\ \vdots \\ P_i \\ \vdots \\ P_M \end{bmatrix} + \begin{bmatrix} 1 \\ \vdots \\ a_1^{i-1} \\ \vdots \\ a_{M-1}^{i-1} \end{bmatrix} * \Delta D_i$$

In the example above, we denote the corresponding RS-code scheme as RS(K,N). Upon node failures, any K nodes of the cluster can recover data or parity lost in the failed nodes, and thus RS(K,N) can handle M node

[1] Both data blocks and parity blocks are called code words in coding theory. We term "parity blocks" as those code words generated from the original data and "data blocks" as the original data.



(a) Key/value store with primary-backup replication (b) Key/value store with erasure coding

Figure 1: Data storage with two different replication schemes.

failures at most. During recovery, the system recalculates the lost data or parity by solving the equations generated by the above equation.

As only M of N nodes are used for storing parities, the memory efficiency can reach K/N. For example, an RS(3,5) coding scheme has storage efficiency of 60% while tolerating up to two node failures.

2.2 Opportunities and Challenges

The emergence of in-memory computing significantly boosts the performance of many systems. However, this also means that a large amount of data needs to be placed in memory. As memory is currently volatile, a node failure would cause data loss for a large chunk of memory. Even if the data has its backup in persistent storage, it would require non-trivial time to recover the data for a single node [14].

However, simply using PBR may cause significant memory inefficiency. Despite an increase of the volume, memory is still a scarce resource, especially when processing the "big-data" applications. It was frequently reported that memory bloat either significantly degraded the performance or simply caused server crashes [4]. This is especially true for workload-sharing clusters, where the budget for storing specific application data is not large.

Opportunities: The need for both availability and memory efficiency makes erasure coding a new attractive design point. The increase of CPU speed and the CPU core counts make erasure coding suitable to be used even in the critical path of data processing. Table 1 presents the encoding and decoding speed for different Reed-Solomon coding scheme on a 5-node cluster with an average CPU core (2.3 GHz Xeon E5, detailed configurations in section 6.1). Both encoding and decoding can be done at 4.24-5.52GB/s, which is several hundreds of times compared to 20 years ago (e.g., 10MB/s [31]). This means that an average-speed core is enough to handle data transmitted through even a network link with 40Gb/s. This reveals a new opportunity to trade CPU resources for better memory efficiency to provide high availability.

scheme	encoding speed	decoding speed
RS(4,5)	5.52GB/s	5.20GB/s
RS(3,5)	5.26GB/s	4.83GB/s
RS(2,5)	4.56GB/s	4.24GB/s

Table 1: The speed of coding data with different schemes for a 5-node cluster

Challenges: However, trivially applying erasure coding to in-memory KV-stores may result in significant performance degradation and consistency issues.

The first challenge is that coding is done efficiently only in a bulk-oriented nature. However, an update operation in a KV-store may result in a number of small updates, which would introduce notable coding operations and network traffic. For example, in Memcached, both the hashtable and the allocation metadata need to be modified for a *set* operation. For the first case, a KV pair being inserted into a bucket will change the four pointers of the double linked list. Some statistics like that for LRU replacement need to be changed as well. In the case of a hashtable expansion or shrinking, all key/value pairs may need to be relocated, causing a huge amount of updates. For the allocation metadata, as Memcached uses a slab allocator, an allocation operation commonly changes four variables and a free operation changes six to seven variables.

The second challenge is that a data update involves updates to multiple parity blocks across machines. During data recovery, there are also multiple data blocks and parity blocks involved. If there are concurrent updates in progress, this may easily cause inconsistent recovery of data.

3 Design

3.1 Interface and Assumption

Cocytus is an in-memory replication scheme for key/value stores (KV-stores) to provide high memory efficiency and high availability with low overhead. It assumes that a KV-store has two basic operations: $Value \leftarrow get(Key)$ and $set(Key, Value)$, where Key

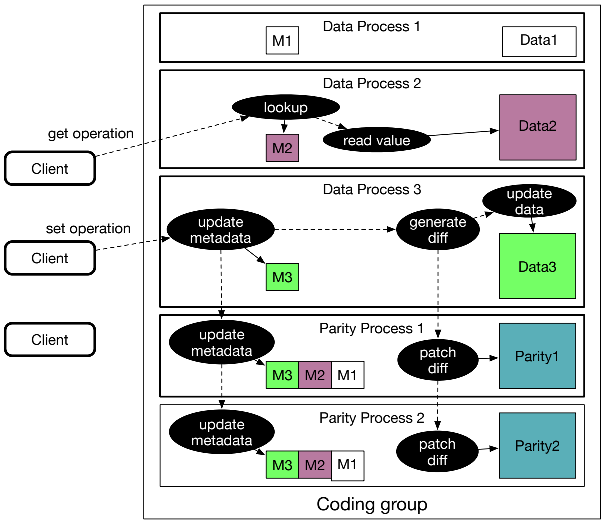


Figure 2: Requests handled by an coding group in Cocytus, where K=3, M=2.

and Value are arbitrary strings. According to prior large-scale analysis on key/value stores in commercial workloads [1, 24], Cocytus assumes that the value size is usually much larger than the key size.

Cocytus handles only omission node failures where a node is fail-stop and won't taint other nodes; commission or Byzantine failures are not considered. It also does not consider a complete power outage that crashes the entire cluster. In such cases, it assumes that there is another storage layer that constantly stores data to preserve durability [24]. Alternatively, one may leverage battery-backed RAM like NVDIMM [37, 35] to preserve durability.

Cocytus is designed to be synchronous, i.e., a response of a *set* request returned to the client guarantees that the data has been replicated/coded and can survive node failures.

Cocytus works efficiently for read-mostly workloads, which are typical for many commercial KV-stores [1]. For update-intensive workloads, Cocytus would use more CPU resources due to the additional calculations caused by the erasure coding, and achieve a similar latency and throughput compared to a simple primary-backup replication.

3.2 Architecture

Cocytus separates data from metadata and leverages a hybrid scheme: metadata and key are replicated using primary-backup while values are erasure coded.

One basic component of Cocytus is the coding group, as shown in Figure 2. Each group comprises K data processes handling requests to data blocks and M parity processes receiving update requests from the data processes. A *get* operation only involves one data node, while a *set* operation updates metadata in both primary and its backup node, and generates diffs to be patched to the parity codes.

Cocytus uses sharding to partition key/value tuples into different groups. A coding group handles a key *shard*, which is further divided into P *partitions* in the group. Each partition is handled by a particular data process, which performs coding at the level of virtual address spaces. This makes the coding operation neutral to the changes of value sizes of a KV pair as long as the address space of a data process does not change. There is no data communication among the data processes, which ensures fault isolation among data processes. When a data process crashes, one parity process immediately handles the requests for the partition that belongs to crashed nodes and recovers the lost data, while other data processes continuously provide services without disruption.

Cocytus is designed to be strongly consistent, which never loses data or recovers inconsistent data. However, strict ordering on parity processes is not necessary for Cocytus. For example, two data processes update their memory at the same time, which involves two updates on the parity processes. However, the parity processes can execute the updates in any order as long as they are notified that the updates have been received by all of the parity processes. Thus, in spite of the update ordering, the data recovered later are guaranteed to be consistent. Section 4.1.2 will show how Cocytus achieves consistent recovery when a failure occurs.

3.3 Separating Metadata from Data

For a typical KV-store, there are two types of important metadata to handle requests. The first is the mapping information, such as a (distributed) hashtable that maps keys to their value addresses. The second one is the allocation information. As discussed before, if the metadata is erasure coded, there will be a larger number of small updates and lengthy unavailable duration upon crashes.

Cocytus uses primary-backup replication to handle the mapping information. In particular, the parity processes save the metadata for all data processes in the same coding group. For the allocation information, Cocytus applies a slab-based allocation for metadata allocation. It further relies on an additional deterministic allocator for data such that each data process will result in the same memory layout for values after every operation.

Interleaved layout: One issue caused by this design is that parity processes save more metadata than those in the data processes, which may cause memory imbalance. Further, as parity processes only need to participate in *set* operations, they may become idle for read-mostly workloads. In contrast, for read-write workloads, the parity processes may become busy and may become a bottleneck of the KV-store.

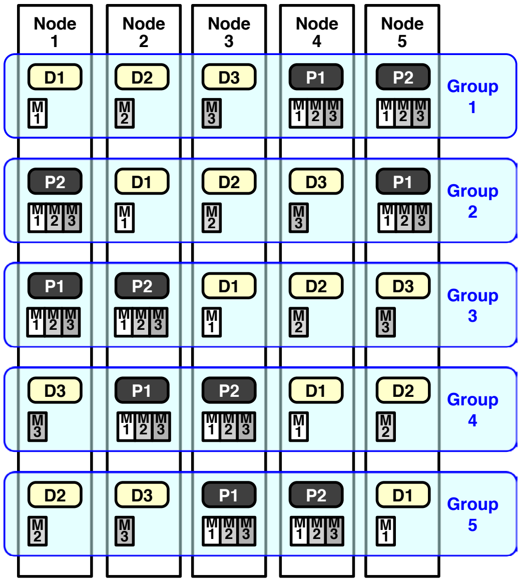


Figure 3: Interleaved layout of coding groups in Cocytus. The blocks in the same row belong to one coding group.

To address these issues, Cocytus interleaves coding groups in a cluster to balance workload and memory on each node, as shown in Figure 3. Each node in Cocytus runs both parity processes and data processes; a node will be busy on parity processes or data processes for update-intensive or read-mostly workload accordingly.

The interleaved layout can also benefit the recovery process by exploiting the cluster resources instead of one node. Because the shards on one node belong to different groups, a single node failure leads a process failure on each group. However, the first parity nodes of these groups are distributed across the cluster, all nodes will work together to do recovery.

To extend Cocytus in a large scale cluster, there are three dimensions to consider, including the number of data processes (K) and the number of parity processes (M) in a coding group, as well as the number of coding groups. A larger K increases memory efficiency but makes the parity process suffer from higher CPU pressure for read-write workloads. A larger M leads to more failures to be tolerated but decreases memory efficiency and degrades the performance of *set* operations. A neutral way to extend Cocytus is deploying more coding groups.

3.4 Consistent Parity Updating with Piggybacking

Because an erasure-coding group has multiple parity processes, sending the update messages to such processes needs an *atomic broadcast*. Otherwise, a KV-store may result in inconsistency. For example, when a data process has received a *set* request and is sending updates to two parity processes, a failure occurs and only one parity process has received the update message. The following recovery might recover incorrect data due to the inconsistency between parities.

A natural solution to this problem is using two-phase commit (2PC) to implement atomic broadcast. This, however, requires two rounds of messages and doubles the I/O operations for *set* requests. Cocytus addresses this problem with a piggybacking approach. Each request is assigned with an *xid*, which monotonously increases at each data process like a logical clock. Upon receiving parity updates, a parity process first records the operation in a buffer corresponding with the *xid* and then immediately send acknowledgements to its data process. After the data process receives acknowledgements from all parity processes, the operation is considered stable in the KV-store. The data process then updates the *latest stable xid* as well as data and metadata, and sends a response to the client. When the data process sends the next parity update, this request piggybacks on the *latest stable xid*. When receiving a piggybacked request, the parity processes mark all operations that have smaller *xid* in the corresponding buffer as *READY* and install the updates in place sequentially. Once a failure occurs, the corresponding requests that are not received by all parity processes will be discarded.

4 Recovery

When a node crashes, Cocytus needs to reconstruct lost data online while serving client requests. Cocytus assumes that the KV-store will eventually keep its fault tolerance level by assigning new nodes to host the recovered data. Alternatively, Cocytus can degenerate its fault tolerance level to tolerate fewer failures. In this section, we first describe how Cocytus recovers data in-place to the parity node and then illustrate how Cocytus migrates the data to recover the parity and data processes when a crashed node reboots or a new standby node is added.

4.1 Data Recovery

Because data blocks are only updated at the last step of handling *set* requests which is executed sequentially with *xid*. We can regard the *xid* of the latest completed request as the logical timestamp (T) of the data block. Similarly, there are K logical timestamps ($VT[1..K]$) for a parity block, where K is the number of the data processes in the same coding group. Each of the K logical timestamps is the *xid* of the latest completed request from the corresponding data process.

Suppose data processes 1 to F crash at the same time. Cocytus chooses all alive data blocks and F parity blocks to reconstruct the lost data blocks. Suppose the logical timestamps of data blocks are T_{F+1}, T_{F+2}, ..., T_K and the logical timestamps of parity blocks are VT_1,

VT_2, ..., VT_{F}. If $VT_1 = VT_2 = ... = VT_{\text{F}}$ and $VT_1[F+1..K] = \langle T_{\text{F+1}}, T_{\text{F+2}}, ..., T_{\text{K}} \rangle$, then theses data blocks and parity blocks agree with formula (1). Hence, they are consistent.

The recovery comprises two phases: preparation and online recovery. During the preparation phase, the parity processes synchronize their request buffers that correspond to the failed processes. Once the preparation phase completes, all parity blocks are consistent on the failed processes. During online recovery, alive data process send their data blocks with its logical timestamp, so the parity processes can easily provide the consistent parity blocks.

4.1.1 Preparation

Once a data process failure is detected, a corresponding parity process is selected as the recovery process to do the recovery and to provide services on behalf of the crashed data process. The recovery process first collects latest *xid*s which correspond to failed data processes from all parity processes. Hence, a parity process has a latest *xid* for each data process because it maintains an individual request buffer for each data process. The minimal latest *xid* is then chosen as the stable *xid*. Requests with greater *xid* received by the failed data process haven't been successfully received by all parity processes and thus should be discarded. Then, the stable *xid* is sent to all parity processes. The parity processes apply the update requests in place of which the *xid* equal to or less than the stable *xid* in the corresponding buffer. After that, all parity processes are consistent in the failed data process because their corresponding logical timestamps are all the same with the stable *xid*.

The preparation phase blocks key/value requests for a very short time. According to our evaluation, the blocking time is only 7ms to 13 ms even under a high workload.

4.1.2 Online recovery

The separation of metadata and data enables online recovery of key/value pairs. During recovery, the recovery process can leverage the replicated metadata to reconstruct lost data online to serve client requests, while using idle CPU cycles to proactively reconstruct other data.

During the online recovery, data blocks are recovered in a granularity of 4KB, which is called a recovery unit. According to the address, each recovery unit is assigned an ID for the convenience of communication among processes.

As shown in Figure 4, there are five steps in our online recovery protocol:

- 1. To reconstruct a recovery unit, a recovery process becomes the recovery initiator and sends messages consisting of the recovery unit ID and a list of involved recovery processes to alive data processes.

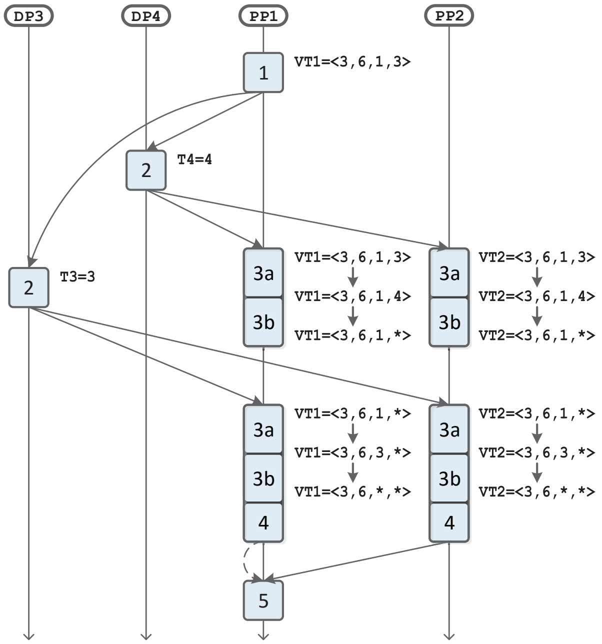


Figure 4: Online recovery when DP1 and DP2 crash in an RS(4, 6) coding group

- 2. When the ith data process receives the message, it sends the corresponding data unit to all recovery processes along with its logical timestamp T_i.

- 3(a). When a recovery process receives the data unit and the logical timestamp T_i, it first applies the requests whose *xid* equals to or less than T_i in the corresponding buffer. At this time, the ith logical timestamp on this recovery process equals to T_i.

- 3(b). The recovery process subtracts the corresponding parity unit by the received data unit with the predefined coefficient. After the subtraction completes, the parity unit is no longer associated with the ith data process. It stops being updated by the ith data process. Hence, the rest of parity units on this recovery process are still associated with the ith data process.

- 4. When a recovery process has received and handled all data units from alive data processes, it sends the final corresponding parity unit to the recovery initiator, which is only associated with the failed data processes.

- 5. When the recovery initiator has received all parity units from recovery processes, it decodes them by solving the following equation, in which the $fn_1, fn_2, ..., fn_F$ indicate the numbers of F failure data processes and the rn_1, rn_2, ..., rn_F indicate the numbers of F parity processes chosen to be the recovery processes.

$$\begin{bmatrix} P_{rn_1} \\ P_{rn_2} \\ \vdots \\ P_{rn_F} \end{bmatrix} = \begin{bmatrix} a_{rn_1-1}^{fn_1-1} & a_{rn_1-1}^{fn_2-1} & \cdots & a_{rn_1-1}^{fn_F-1} \\ a_{rn_2-1}^{fn_1-1} & a_{rn_2-1}^{fn_2-1} & \cdots & a_{rn_2-1}^{fn_F-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{rn_F-1}^{fn_1-1} & a_{rn_F-1}^{fn_2-1} & \cdots & a_{rn_F-1}^{fn_F-1} \end{bmatrix} * \begin{bmatrix} D_{fn_1} \\ D_{fn_2} \\ \vdots \\ D_{fn_F} \end{bmatrix} \tag{2}$$

Correctness argument: Here we briefly argue the correctness of the protocol. Because when a data block is updated, all parity processes should have received the corresponding update requests. Hence, in step 3(a), the parity process must have received all required update requests and can synchronize its corresponding logical timestamp with the received logical timestamp. Since the received data block and parity block have the same logical timestamps, the received data block should be the same as the data block which is used to construct the parity block. Because a parity block is a *sum* of data blocks with the individual predefined coefficients in the *Vandermonde* matrix, after the subtraction in step 3(b), the parity block is only constructed by the rest of data blocks. At the beginning of step 4, the parity block is only constructed by the data blocks of failed data processes because the parity process has done step 3 for each alive data process. Finally, with the help of stable *xid* synchronization in the preparation phase, the parity blocks received in step 5 are all consistent and should agree with equation 2.

4.1.3 Request Handling on Recovery Process

Cocytus allows a recovery process to handle requests during recovery. For a *get* request, it tries to find the key/value pair through the backup hashtable. If it finds the pair, the recovery process checks whether the data blocks needed for the value have been recovered. If the data blocks have not been recovered, the recovery process initiates data block recovery for each data block. After the data blocks are recovered, the recovery process sends the response to the client with the requested value.

For a *set* request, the recovery process allocates a new space for the new value with the help of the allocation metadata in the backup. If the allocated data blocks are not recovered, the recovery process calls the recovery function for them. After recovery, the recovery process handles the operation like a normal data process.

4.2 Data Migration

Data process recovery: During the data process recovery, Cocytus can migrate the data from the recovery process to a new data process. The recovery process first sends the keys as well as the metadata of values (i.e., sizes and addresses) in the hashtable to the new data process. While receiving key/value pairs, the new data process rebuilds the hashtable and the allocation metadata. After all key/value pairs are sent to the new data process, the recovery process stops providing services to clients.

When metadata migration completes, the data (i.e., value) migration starts. At that moment, the data process can handle the requests as done in the recovery process. The only difference between them is that the data process does not recover the data blocks by itself. When data process needs to recover a data block, it sends a request to the recovery process. If the recovery process has already recovered the data block, it sends the recovered data block to the data process directly. Otherwise, it starts a recovery procedure. After all data blocks are migrated to the data process, the migration completes.

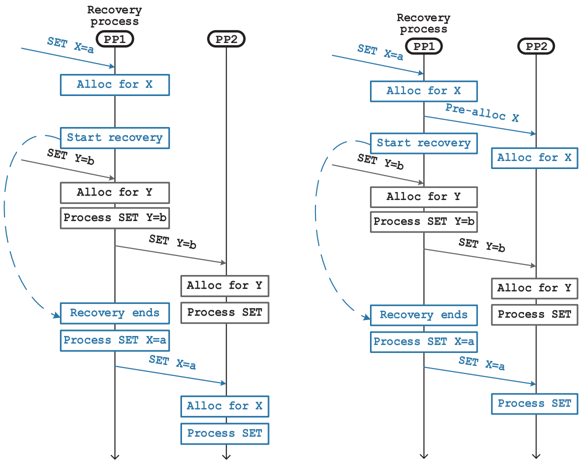
If either the new data process or the corresponding recovery process fails during data migration, both of them should be killed. This is because having only one of them will lead to insufficient information to provide continuous services. Cocytus can treat this failure as a data process failure.

Parity process recovery: The parity process recovery is straightforward. After a parity process crashes, the data process marks all data blocks with a *miss* bit for that parity process. The data processes first send the metadata to the recovering parity process. Once the transfer of metadata completes, the logical timestamps of new parity processes are the same with the metadata it has received. After the transfer of metadata, the data processes migrate the data that may overlap with *parity update* requests. Before sending a *parity update* request which involves data blocks marked with a *miss* bit, the data process needs to send the involved data blocks to the new parity process. In this way, data blocks sent to the new parity process have the same logical timestamps with the metadata sent before. After the new parity process receives all data blocks, the recovery completes. If either of the data processes fails during the recovery of the parity process, the recovery fails and Cocytus starts to recover the failed data process.

5 Implementation

We first built a KV-store with Cocytus from scratch. To understand its performance implication on real KV-stores, we also implemented Cocytus on top of Memcached 1.4.21 with the synchronous model, by adding about 3700 SLoC to Memcached. Currently, Cocytus only works for single-thread model and the data migration is not fully supported. To exploit multicore, Cocytus can be deployed with sharding and multi-process instead of multi-threading. In fact, using multi-threading has no significant improvement for data processes which may suffer from unnecessary resource contention and break data isolation. The parity processes could be implemented in a multi-threaded way to distribute the high CPU pressure under write-intensive workloads, which we leave as future work. We use Jerasure [27] and GF-complete [26] for the Galois-Field operations in RS-code. Note that Cocytus is largely orthogonal with the coding schemes; it will be our future work to apply other network or space-efficient coding schemes [33, 28]. This section describes some implementation issues.

Deterministic allocator: In Cocytus, the allocation metadata is separated from data. Each data process maintains a memory region for data with the *mmap* syscall. Each parity process also maintains an equivalent memory region for parity. To manage the data region, Cocytus uses two AVL trees, of which one records the free space and the other records the allocated space. The tree node consists of the start address of a memory piece and its length. The length is ensured to be multiples of 16 and is used as the index of the trees. Each memory location is stored in either of the trees. An *alloc* operation will find an appropriate memory piece in the free-tree and move it to the allocated-tree and the *free* operations do the opposite. The trees manage the memory pieces in a way similar to the buddy memory allocation: large blocks might be split into small ones during *alloc* operations and consecutive pieces are merged into a larger one during *free* operations. To make the splitting and merging fast, all memory blocks are linked by a list according to the address. Note that only the metadata is stored in the tree, which is stored separately from the actual memory managed by the allocator.



(a) without pre-alloc

(b) with pre-alloc

Figure 5: In (a), the memory allocation ordering for X and Y is different on PP1 and PP2. In (b), thanks to the pre-alloc, the memory allocation ordering remains the same on different processes.

Pre-alloc: Cocytus uses the deterministic allocator and hashtables to ensure all metadata in each node is consistent. Hence, Cocytus only needs to guarantee that each process will handle the related requests in the same order. The piggybacked two-phase commit (section 3.4) can mostly provide such a guarantee.

One exception is shown in Figure 5(a). When a recovery process receives a *set* request with X=a, it needs to allocate memory for the value. If the memory for the value needs to be recovered, the recovery process first starts the recovery for X and puts this *set* request into a waiting queue. In Cocytus, the recovery is asynchronous. Thus, the recovery process is able to handle other requests before the recovery is finished. During this time frame, another *set* request with Y=b comes to the recovery process. The recovery process allocates memory for it and fortunately the memory allocated has already been recovered. Hence, the recovery process directly handles the *set* request with Y=b without any recovery and sends requests to other parity processes for fault-tolerance. As soon as they receive the request, other processes (for example, PP2 in the figure) allocate memory for Y and finish their work as usual. Finally, when the recovery for X is finished, the recovery process continues to handle the *set* request with X=a. It also sends fault-tolerance requests to other parity processes, on which the memory is allocated for X. Up to now, the recovery process has allocated memory for X and Y successively. However, on other parity processes, the memory allocation for Y happens before that for X. This different allocation ordering between recovery processes and parity processes will cause inconsistency.

Cocytus solves this problem by sending a pre-allocation request (shown in Figure 5(b)) before each *set* operation is queued due to recovery. In this way, the parity processes can pre-allocate space for the queued set requests and the ordering of memory allocation is guaranteed.

Recovery leader: Because when multiple recovery processes want to recover the two equivalent blocks simultaneously, both of them want to start an online recovery protocol, which is unnecessary. To avoid this situation, Cocytus assigns a recovery leader in each group. A recovery leader is a parity process responsible for initiating and finishing the recovery in the group. All other parity processes in the group will send recovery requests to the recovery leader if they need to recover data, and the recovery leader will broadcast the result after the recovery is finished. A recovery leader is not absolutely necessary but such a centralized management of recovery can prevent the same data from being recovered multiple times and thus reduce the network traffic. Considering the interleaved layout of the system, the recovery leaders are uniformly distributed on different nodes and won't become the bottleneck.

Short-cut Recovery for Consecutive Failures: When there are more than one data process failures and the data of some failed processes are already recovered by the recovery process, the further recovered data might be wrong if we do not take the recovery process into consideration.

In the example given in Figure 4, suppose DP1 (data process 1) fails first and PP1 (parity process 1) becomes a recovery process for it. After PP1 recovered a part of data blocks, DP2 fails and PP2 becomes a recovery process for DP2. At that moment, some data blocks on PP1 have been recovered and others haven't. To recover a data block on DP2, if its corresponding data block on DP1 has been recovered, it should be recovered in the way that involves 3 data blocks and 1 parity block, otherwise it should be recovered in the way that involves 2 data blocks and 2 parity blocks. The procedures of the two kinds of recovery are definitely different.

Primary-backup replication: To evaluate Cocytus, we also implemented a primary-backup (PBR) replication version based on Memcached-1.4.21 with almost the same design as Cocytus, like synchronous write, piggyback, except that Cocytus puts the data in a coded space and needs to decode data after a failure occurs. We did not directly use Repcached [17] for two reasons. One is that Repcached only supports one slave worker. The other one is that *set* operation in Repcached is asynchronous and thus does not guarantee crash consistency.

6 Evaluation

We evaluate the performance of Cocytus by comparing it to primary-backup replication (PBR) and the vanilla Memcached. The highlights of our evaluation results are the followings:

- Cocytus achieves high memory efficiency: It reduces memory consumption by 33% to 46% for value sizes from 1KB to 16KB when tolerating two node failures.

- Cocytus incurs low overhead: It has similar throughput with PBR and vanilla KV-store (i.e., Memcached) and incurs small increase in latency compared to vanilla KV-store.

- Cocytus can tolerate failures as designed and recover fast and gracefully: Even under two node crashes, Cocytus can gracefully recover lost data and handle client requests with close performance with PBR.

6.1 Experimental Setup

Hardware and configuration: Due to our hardware limit, we conduct all experiments on a 6-node cluster of machines. Each machine has two 10-core 2.3GHz Intel Xeon E5-2650, 64GB of RAM and is connected with 10Gb network. We use 5 out of the 6 nodes to run as servers and the remaining one as client processes.

To gain a better memory efficiency, Cocytus could use more data processes in a coding group. However, deploying too many data processes in one group increases the burden on parity processes, which could be a bottleneck of the system. Because of the limitation of our cluster, we deploy Cocytus with five interleaved EC groups which are configured as RS(3,5) so that the system can tolerate two failures while maximizing the data processes. Each group consists of three data processes and two parity processes. With this deployment, each node contains three data processes and two parity processes of different groups.

Targets of comparison: We compare Cocytus with PBR and vanilla Memcached. To evaluate PBR, we distribute 15 data processes among the five nodes. For each data process, we launch 2 backup processes so that the system can also tolerate two node failures. This deployment launches more processes (45 processes) compared to Cocytus (25 processes), which could use more CPU resource in some cases. We deploy the vanilla Memcached by evenly distributing 15 instances among the five nodes. In this way, the number of processes of Memcached is the same as the data processes of Cocytus.

Workload: We use the YCSB [8] benchmark to generate our workloads. We generate each key by concatenating the a table name and an identifier, and a value is a compressed HashMap object, which consists of multiple fields. The distribution of the key probability is Zipfian [10], with which some keys are hot and some keys are cold. The length of the key is usually smaller than 16B. We also evaluate the systems with different read/write ratios, including equal-shares (50%:50%), read-mostly(95%:5%) and read-only (100%:0%).

Since the median of the value sizes from Facebook [24] are 4.34KB for *Region* and 10.7KB for *Cluster*, we test these caching systems with similar value sizes. As in YCSB, a value consists of multiple fields, to evaluate our system with various value sizes, we keep the field number as 10 while changing the field size to make the total value sizes be 1KB/4KB/16KB, i.e., the field sizes are 0.1KB/0.4KB/1.6KB accordingly. To limit the total data size to be 64GB, the item numbers for 1/4/16 KB are 64/16/1 million respectively. However, due to the object compression, we cannot predict the real value size received by the KV-store and the values may not be aligned as well; Cocytus aligns the compressed values to 16 bytes to perform coding.

6.2 Memory Consumption

As shown in Figure 6, Cocytus achieves notable memory saving compared to PBR, due to the use of erasure coding. With a 16KB value size, Cocytus achieves 46%

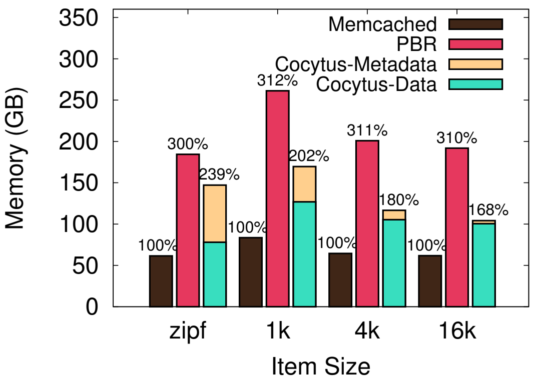


Figure 6: Memory consumption of three systems with different value sizes. Due to the compression in YCSB, the total memory cost for different value sizes differs a little bit.

memory saving compared to PBR. With RS(3,5), the expected memory overhead of Cocytus should be 1.66X while the actual memory overhead ranges from 1.7X to 2X. This is because replicating metadata and keys introduces more memory cost, e.g., 25%, 9.5% and 4% of all consumed memory for value sizes of 1KB, 4KB and 16KB. We believe such a cost is worthwhile for the benefit of fast and online recovery.

To investigate the effect of small- and variable-sized values, we conduct a test in which the value size follows the Zipfian distribution over the range from 10B to 1KB. Since it is harder to predict the total memory consumption, we simply insert 100 million such items. The result is shown as *zipf* in Figure 6. As expected, more items bring more metadata (including keys) which diminishes the benefit of Cocytus. Even so, Cocytus still achieves 20% memory saving compared to PBR.

6.3 Performance

As shown in Figure 7, Cocytus incurs little performance overhead for read-only and read-mostly workloads and incurs small overhead for write-intensive workload compared to vanilla Memcached. Cocytus has similar latency and throughput with PBR. The followings use some profiling data to explain the data.

Small overhead of Cocytus and PBR: As the three configurations handle *get* request with similar operations, the performance is similarly in this case. However, when handling *set* requests, Cocytus and PBR introduce more operations and network traffic and thus modestly higher latency and small degradation of throughput. From the profiled CPU utilization (Table 2) and network traffic (Memcached:540Mb/s, PBR: 2.35Gb/s, Cocytus:2.3Gb/s, profiled during 120 clients insert data), we found that even though PBR and Cocytus have more CPU operations and network traffic, both of them were not the bottleneck. Hence, multiple requests from clients can be overlapped and pipelined. Hence, the throughput is similar with the vanilla Memcached. Hence, both Cocytus and PBR can trade some CPU and network resources for high availability, while incurring small user-perceived performance overhead.

Higher write latency of PBR and Cocytus: The latency is higher when the read-write ratio is 95%:5%, which is a quite strange phenomenon. The reason is that *set* operations are preempted by *get* operations. In Cocytus and PBR, *set* operations are FIFO, while *set* operations and *get* operations are interleaved. Especially in the read-mostly workload, the *set* operations tend to be preempted, as *set* operations have longer path in PBR and Cocytus.

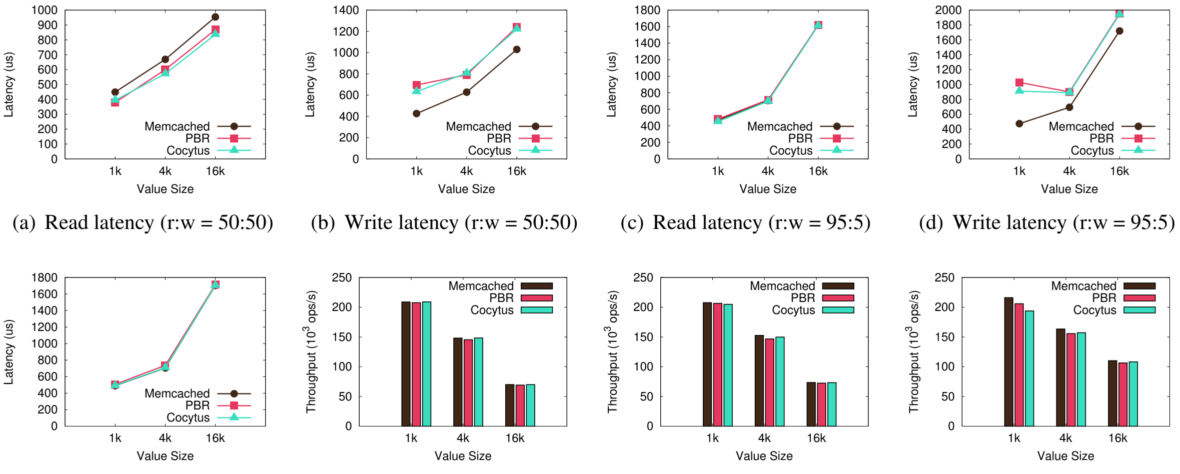
Lower read latency of PBR and Cocytus: There is an interesting phenomenon is that higher write latency causes lower read latency for PBR and Cocytus under update-intensive case (i.e., r:w = 50:50). This may be because when the write latency is higher, more client threads are waiting for the *set* operations at a time. However, the waiting on *set* operation does not block the *get* operation from other client threads. Hence, the client threads waiting on *get* operation could be done faster because there would be fewer client threads that could block this operation. As a result, the latency of *get* is lower.

6.4 Recovery Efficiency

We evaluate the recovery efficiency using 1KB value size for read-only, read-mostly and read-write workloads. We emulate two node failures by manually killing all processes on the node. The first node failure occurs at 60s after the benchmark starts. And the other node failure occurs at 100s, before the recovery of the first failure finishes. The two throughput collapses in each of the subfigures of Figure 8 are caused by the TCP connection mechanism and can be used coincidentally to indicate the time a node fails. The vertical lines indicate the time that all the data has been recovered.

Our evaluation shows that after the first node failure, Cocytus can repair the data at 550MB/s without client requests. The speed could be much faster if we use more processes. However, to achieve high availability, Cocytus first does recovery for requested units and recovers cold data when the system is idle.

As shown in Figure 8(a), Cocytus performs similarly as PBR when the workload is read-only, which confirms that data recovery could be done in parallel with read requests without notable overhead. The latencies for 50%, 90%, 99% requests are 408us, 753us and 1117us in Cocytus during recovery. Similar performance can be achieved when the read-write ratio is 95%, as shown in Figure 8(b). In the case with frequent *set* requests, as shown in Figure 8(c), the recovery affects the throughput of normal request handling modestly. The reason



(a) Read latency (r:w = 50:50) (b) Write latency (r:w = 50:50) (c) Read latency (r:w = 95:5) (d) Write latency (r:w = 95:5)

(e) Read latency (r:w = 100:0) (f) Throughput (r:w = 100:0) (g) Throughput (r:w = 95:5) (h) Throughput (r:w = 50:50)

Figure 7: Comparison of latency and throughput of the three configurations.

Read : Write	Memcached	PBR		Cocytus	
	15 processes	15 primary processes	30 backup processes	15 data processes	10 parity processes
50%:50%	231%CPUs	439%CPUs	189%CPUs	802%CPUs	255%CPUs
95%:5%	228%CPUs	234%CPUs	60%CPUs	256%CPUs	54%CPUs
100%:0%	222%CPUs	230%CPUs	21%CPUs	223%CPUs	15%CPUs

Table 2: CPU utilization for 1KB value size

is that to handle *set* operations Cocytus needs to allocate new blocks, which usually triggers data recovery on those blocks. Waiting for such data recovery to complete degrades the performance. In fact, after the first node crashes, the performance is still acceptable, since the recovery is relatively simpler and not all processes are involved in the recovery. However, when two node failures occur simultaneously, the performance can be affected more notably. Fortunately, this is a very rare case and even if it happens, Cocytus can still provide services with reasonable performance and complete the data recovery quickly.

To confirm the benefit of our online recovery protocol, we also implement a blocked version of Cocytus for comparison. In the blocked version of Cocytus, the *set* operations are delayed if there is any recovery in progress and the *get* operations are not affected. From Figure 8, we can observe that the throughput of the blocked version collapses even when there is only one node failure and 5% of *set* operations.

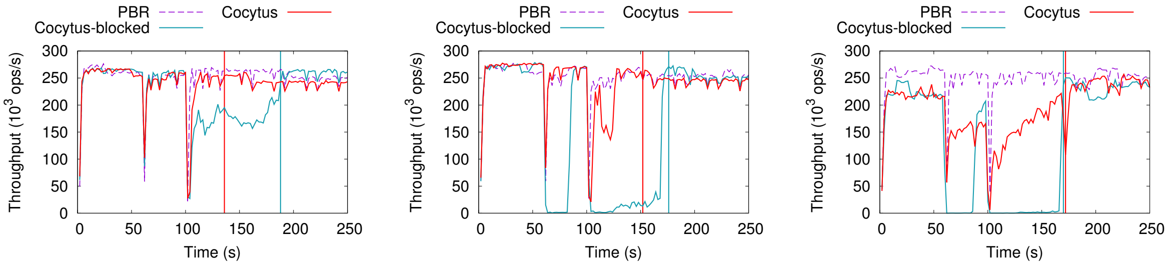
6.5 Different Coding Schemes

To understand the effect under different coding schemes, we evaluate the Cocytus with RS(4,5), RS(3,5) and RS(2,5). As shown in Figure 9, the memory consumption of RS(2,5) is the largest and the one of RS(4,5) is the least. All the three coding schemes benefit more from larger value sizes. Their throughput is similar because there are no bottlenecks on servers. However, the write latency of RS(2,5) is a little bit longer since it sends more messages to parity processes. The reason why RS(2,5) has lower read latency should be a longer write latency causes lower read latency (similar as the case described previously).

7 Related Work

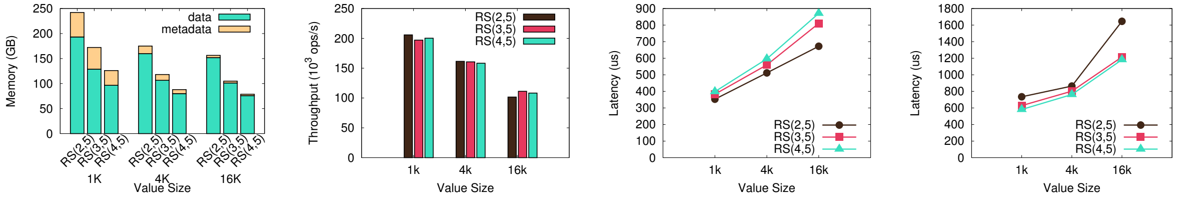
Separation of work: The separation of metadata/key and values is inspired by prior efforts on separation of work. For example, Wang et al. [42] separate data from metadata to achieve efficient Paxos-style asynchronous replication of storage. Yin et al. [46] separate execution from agreement to reduce execution nodes when tolerating Byzantine faults. Clement et al. [6] distinguish omission and Byzantine failures and leverage redundancy between them to reduce required replicas. In contrast, Cocytus separates metadata/key from values to achieve space-efficient and highly-available key/value stores.

Erasure coding: Erasure coding has been widely adopted in storage systems in both academia and industry to achieve both durability and space efficiency [15, 34, 29, 32, 23]. Generally, they provide a number of optimizations that optimize the coding efficiency and recovery bandwidth, like local reconstruction codes [15], Xorbas [32], piggyback codes [29] and lazy recovery [34]. PanFS [44] is a parallel file system that uses per-file erasure coding to protect files greater than 64KB, but repli-



(a) Throughput on recovery (r:w = 100:0) (b) Throughput on recovery (r:w = 95:5) (c) Throughput on recovery (r:w = 50:50)

Figure 8: Performance of PBR and Cocytus when nodes fail. The vertical lines indicate all data blocks are recovered completely.



(a) Memory cost (b) Throughput (r:w = 50:50) (c) Read latency (r:w = 50:50) (d) Write latency (r:w = 50:50)

Figure 9: Performance under different coding schemes

cates metadata and small files to minimize the cost of metadata updates.

Replication: Replication is a standard approach to fault tolerance, which may be categorized into synchronous [5, 3, 39] and asynchronous [19, 2]. Mojim [48] combines NVRAM and a two-tier primary-backup replication scheme to optimize database replication. Cocytus currently leverages standard primary-backup replication to provide availability to metadata and key in the face of omission failures. It will be our future work to apply other replications schemes or handle commission failures.

RAMCloud [25] exploits scale of clusters to achieve fast data recovery. Imitator [41] leverages existing vertices in partitioned graphs to provide fault-tolerant graph computation, which also leverages multiple replicas to recover failed data in one node. However, they do not provide online recovery such that the data being recovered cannot be accessed simultaneously. In contrast, Cocytus does not require scale of clusters for fast recovery but instead provide always-on data accesses, thanks to replicating metadata and keys.

Key/value stores: There have been a considerable number of interests in optimizing key/value stores, leveraging advanced hardware like RDMA [22, 36, 16, 43] or increasing concurrency [11, 20, 21]. Cocytus is largely orthogonal with such improvements and we believe that Cocytus can be similarly applied to such key/value stores to provide high availability.

8 Conclusion and Future Work

Efficiency and availability are two key demanding features for in-memory key/value stores. We have demonstrated such a design that achieves both efficiency and availability by building Cocytus and integrating it into Memcached. Cocytus uses a hybrid replication scheme by using PBR for metadata and keys while using erasure-coding for values with large sizes. Cocytus is able to achieve similarly normal performance with PBR and little performance impact during recovery while achieving much higher memory efficiency.

We plan to extend our work in several ways. First, we plan to explore a larger cluster setting and study the impact of other optimized coding schemes on the performance of Cocytus. Second, we plan to investigate how Cocytus can be applied to other in-memory stores using NVRAM [40, 7, 45]. Finally, we plan to investigate how to apply Cocytus to replication of in-memory databases.

Acknowledgment

We thank our shepherd Brent Welch and the anonymous reviewers for their constructive comments. This work is supported in part by China National Natural Science Foundation (61572314), the Top-notch Youth Talents Program of China, Shanghai Science and Technology Development Fund (No. 14511100902), Zhangjiang Hi-Tech program (No. 201501-YP-B108-012), and Singapore NRF (CREATE E2S2). The source code of Cocytus is available via `http://ipads.se.sjtu.edu.cn/pub/projects/cocytus`.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, pages 53–64. ACM, 2012.

[2] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *NSDI*, 2011.

[3] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[4] Y. Bu, V. Borkar, G. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *ACM SIGPLAN International Symposium on Memory Management*, pages 119–130. ACM, 2013.

[5] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[6] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290. ACM, 2009.

[7] 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 *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–118. ACM, 2011.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[9] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 international conference on Management of data*, pages 1243–1254. ACM, 2013.

[10] L. Egghe. Zipfian and lotkaian continuous concentration theory. *Journal of the American Society for Information Science and Technology*, 56(9):935–945, 2005.

[11] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 385–398, 2013.

[12] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[13] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[14] A. Goel, B. Chopra, C. Gerea, D. Mátáni, J. Metzler, F. Ul Haq, and J. Wiener. Fast database restarts at facebook. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 541–549. ACM, 2014.

[15] 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 *USENIX Annual Technical Conference*, pages 15–26, 2012.

[16] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 295–306. ACM, 2014.

[17] KLab Inc. http://repcached.lab.klab.org, 2011.

[18] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[19] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[20] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, page 27. ACM, 2014.

[21] R. Liu, H. Zhang, and H. Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC*, volume 14, pages 219–230, 2014.

[22] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[23] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebooks warm blob storage system. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 383–398. USENIX Association, 2014.

[24] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *NSDI*, pages 385–398, 2013.

[25] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[26] J. S. Plank, E. L. Miller, and W. B. Houston. GF-Complete: A comprehensive open source library for Galois Field arithmetic. Technical Report UT-CS-13-703, University of Tennessee, January 2013.

[27] 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, August 2008.

[28] 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 *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 81–94. USENIX Association, 2015.

[29] 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. *Proc. USENIX HotStorage*, 2013.

[30] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.

[31] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM computer communication review*, 27(2):24–36, 1997.

[32] 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 *Proceedings of the VLDB Endowment*, pages 325–336. VLDB Endowment, 2013.

[33] N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchandran. Distributed storage codes with repair-by-transfer and nonachievability of interior points on the storage-bandwidth tradeoff. *Information Theory, IEEE Transactions on*, 58(3):1837–1852, 2012.

[34] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proceedings of International Conference on Systems and Storage*, pages 1–7. ACM, 2014.

[35] SNIA. Nvdimm special interest group. http://www.snia.org/forums/sssi/NVDIMM, 2015.

[36] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached. In *USENIX Annual Technical Conference*, pages 347–353, 2012.

[37] V. Technology. Arxcis-nv (tm): Non-volatile dimm. `http://www.vikingtechnology.com/arxcis-nv`, 2014.

[38] Twitter Inc. Twemcache is the twitter memcached. https://github.com/twitter/twemcache, 2012.

[39] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[40] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, pages 61–75, 2011.

[41] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan. Replication-based fault-tolerance for large-scale graph processing. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 562–573. IEEE, 2014.

[42] Y. Wang, L. Alvisi, and M. Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *USENIX Annual Technical Conference*, pages 413–424, 2012.

[43] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104. ACM, 2015.

[44] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *FAST*, volume 8, pages 1–17, 2008.

[45] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 167–181. USENIX Association, 2015.

[46] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP*, pages 253–267. ACM, 2003.

[47] J. Zawodny. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine*, 79, 2009.

[48] Y. Zhang, J. Yang, A. Memaripour, and S. 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*, pages 3–18. ACM, 2015.

Slacker: Fast Distribution with Lazy Docker Containers

Tyler Harter, Brandon Salmon[†], Rose Liu[†],
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin, Madison [†] Tintri

Abstract

Containerized applications are becoming increasingly popular, but unfortunately, current container-deployment methods are very slow. We develop a new container benchmark, HelloBench, to evaluate the startup times of 57 different containerized applications. We use HelloBench to analyze workloads in detail, studying the block I/O patterns exhibited during startup and compressibility of container images. Our analysis shows that pulling packages accounts for 76% of container start time, but only 6.4% of that data is read. We use this and other findings to guide the design of Slacker, a new Docker storage driver optimized for fast container startup. Slacker is based on centralized storage that is shared between all Docker workers and registries. Workers quickly provision container storage using backend clones and minimize startup latency by lazily fetching container data. Slacker speeds up the median container development cycle by 20× and deployment cycle by 5×.

1 Introduction

Isolation is a highly desirable property in cloud computing and other multi-tenant platforms [8, 14, 27, 22, 24, 34, 38, 40, 42, 49]. Without isolation, users (who are often paying customers) must tolerate unpredictable performance, crashes, and privacy violations.

Hypervisors, or virtual machine monitors (VMMs), have traditionally been used to provide isolation for applications [12, 14, 43]. Each application is deployed in its own virtual machine, with its own environment and resources. Unfortunately, hypervisors need to interpose on various privileged operations (*e.g.*, page-table lookups [7, 12]) and use roundabout techniques to infer resource usage (*e.g.*, ballooning [43]). The result is that hypervisors are heavyweight, with slow boot times [50] as well as run-time overheads [7, 12].

Containers, as driven by the popularity of Docker [25], have recently emerged as a lightweight alternative to hypervisor-based virtualization. Within a container, all process resources are virtualized by the operating system, including network ports and file-system mount points. Containers are essentially just processes that enjoy virtualization of all resources, not just CPU and memory; as such, there is no intrinsic reason container startup should be slower than normal process startup.

Unfortunately, as we will show, starting containers is much slower in practice due to file-system provisioning bottlenecks. Whereas initialization of network, compute, and memory resources is relatively fast and simple (*e.g.*, zeroing memory pages), a containerized application requires a fully initialized file system, containing application binaries, a complete Linux distribution, and package dependencies. Deploying a container in a Docker or Google Borg [41] cluster typically involves significant copying and installation overheads. A recent study of Google Borg revealed: *"[task startup latency] is highly variable, with the median typically about 25 s. Package installation takes about 80% of the total: one of the known bottlenecks is contention for the local disk where packages are written"* [41].

If startup time can be improved, a number of opportunities arise: applications can scale instantly to handle flash-crowd events [13], cluster schedulers can frequently rebalance nodes at low cost [17, 41], software upgrades can be rapidly deployed when a security flaw or critical bug is fixed [30], and developers can interactively build and test distributed applications [31].

We take a two-pronged approach to solving the container-startup problem. First, we develop a new open-source Docker benchmark, HelloBench, that carefully exercises container startup. HelloBench is based on 57 different container workloads and measures the time from when deployment begins until a container is ready to start doing useful work (*e.g.*, servicing web requests). We use HelloBench and static analysis to characterize Docker images and I/O patterns. Among other findings, our analysis shows that (1) copying package data accounts for 76% of container startup time, (2) only 6.4% of the copied data is actually needed for containers to begin useful work, and (3) simple block-deduplication across images achieves better compression rates than gzip compression of individual images.

Second, we use our findings to build Slacker, a new Docker storage driver that achieves fast container distribution by utilizing specialized storage-system support at multiple layers of the stack. Specifically, Slacker uses the snapshot and clone capabilities of our backend storage server (a Tintri VMstore [6]) to dramatically reduce the cost of common Docker operations. Rather than prepropagate whole container images, Slacker lazily pulls image data as necessary, drastically reducing network I/O. Slacker also utilizes modifications we make to the Linux kernel in order to improve cache sharing.

The result of using these techniques is a massive improvement in the performance of common Docker operations; image pushes become 153× faster and pulls become 72× faster. Common Docker use cases involving these operations greatly benefit. For example, Slacker achieves a 5× median speedup for container deployment cycles and a 20× speedup for development cycles.

We also build MultiMake, a new container-based build tool that showcases the benefits of Slacker's fast startup. MultiMake produces 16 different binaries from the same source code, using different containerized GCC releases. With Slacker, MultiMake experiences a 10× speedup.

The rest of this paper is organized as follows. First, we describe the existing Docker framework (§2). Next, we introduce HelloBench (§3), which we use to analyze Docker workload characteristics (§4). We use these findings to guide our design of Slacker (§5). Finally, we evaluate Slacker (§6), present MultiMake (§7), discuss related work (§8), and conclude (§9).

2 Docker Background

We now describe Docker's framework (§2.1), storage interface (§2.2), and default storage driver (§2.3).

2.1 Version Control for Containers

While Linux has always used virtualization to isolate memory, cgroups [37] (Linux's container implementation) virtualizes a broader range of resources by providing six new namespaces, for file-system mount points, IPC queues, networking, host names, process IDs, and user IDs [19]. Linux cgroups were first released in 2007, but widespread container use is a more recent phenomenon, coinciding with the availability of new container management tools such as Docker (released in 2013). With Docker, a single command such as "`docker run -it ubuntu bash`" will pull Ubuntu packages from the Internet, initialize a file system with a fresh Ubuntu installation, perform the necessary cgroup setup, and return an interactive bash session in the environment.

This example command has several parts. First, "ubuntu" is the name of an *image*. Images are read-only copies of file-system data, and typically contain application binaries, a Linux distribution, and other packages needed by the application. Bundling applications in Docker images is convenient because the distributor can select a specific set of packages (and their versions) that will be used wherever the application is run. Second, "run" is an operation to perform on an image; the run operation creates an initialized root file system based on the image to use for a new *container*. Other operations include "push" (for publishing new images) and "pull" (for fetching published images from a central location); an image is automatically pulled if the user attempts to run a non-local image. Third, "bash" is the program to start within the container; the user may specify any executable in the given image.

Docker manages image data much the same way traditional version-control systems manage code. This model is suitable for two reasons. First, there may be different branches of the same image (*e.g.*, "ubuntu:latest" or "ubuntu:12.04"). Second, images naturally build upon one another. For example, the Ruby-on-Rails image builds on the Rails image, which in turn builds on the Debian image. Each of these images represent a new *commit* over a previous commit; there may be additional commits that are not tagged as runnable images. When a container executes, it starts from a committed image, but files may be modified; in version-control parlance, these modifications are referred to as unstaged changes. The Docker "commit" operation turns a container and its modifications into a new read-only image. In Docker, a *layer* refers to either the data of a commit or to the unstaged changes of a container.

Docker worker machines run a local Docker *daemon*. New containers and images may be created on a specific worker by sending commands to its local daemon. Image sharing is accomplished via centralized *registries* that typically run on machines in the same cluster as the Docker workers. Images may be published with a push from a daemon to a registry, and images may be deployed by executing pulls on a number of daemons in the cluster. Only the layers not already available on the receiving end are transferred. Layers are represented as gzip-compressed tar files over the network and on the registry machines. Representation on daemon machines is determined by a pluggable storage driver.

2.2 Storage Driver Interface

Docker containers access storage in two ways. First, users may mount directories on the host within a container. For example, a user running a containerized compiler may mount her source directory within the container so that the compiler can read the code files and produce binaries in the host directory. Second, containers need access to the Docker layers used to represent the application binaries and libraries. Docker presents a view of this application data via a mount point that the container uses as its root file system. Container storage and mounting is managed by a Docker storage driver; different drivers may choose to represent layer data in different ways. The methods a driver must implement are shown in Table 1 (some uninteresting functions and arguments are not shown). All the functions take a string "id" argument that identifies the layer being manipulated.

The `Get` function requests that the driver mount the layer and return a path to the mount point. The mount point returned should contain a view of not only the "id"

Method	Description
Get(id)=dir	mount “id” layer file system, return mount point
Put(id)	unmount “id” layer file system
Create(parent, id)	logically copy “parent” layer to “id” layer
Diff(parent, id)=tar	return compressed tar of changes in “id” layer
ApplyDiff(id, tar)	apply changes in tar to “id” layer

Table 1: **Docker Driver API.**

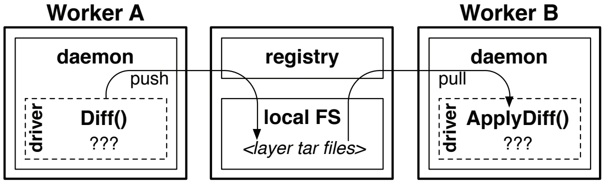


Figure 1: **Diff and ApplyDiff.** *Worker A is using `Diff` to package local layers as compressed tars for a push. B is using `ApplyDiff` to convert the tars back to the local format. Local representation varies depending on the driver, as indicated by the question marks.*

layer, but of all its ancestors (*e.g.*, files in the parent layer of the “id” layer should be seen during a directory walk of the mount point). `Put` unmounts a layer. `Create` copies from a parent layer to create a new layer. If the parent is `NULL`, the new layer should be empty. Docker calls `Create` to (1) provision file systems for new containers, and (2) allocate layers to store data from a pull.

`Diff` and `ApplyDiff` are used during Docker push and pull operations respectively, as shown in Figure 1. When Docker is pushing a layer, `Diff` converts the layer from the local representation to a compressed tar file containing the files of the layer. `ApplyDiff` does the opposite: given a tar file and a local layer it decompresses the tar file over the existing layer.

Figure 2 shows the driver calls that are made when a four-layer image (*e.g.*, ubuntu) is run for the first time. Four layers are created during the image pull; two more are created for the container itself. Layers A-D represent the image. The `Create` for A takes a `NULL` parent, so A is initially empty. The subsequent `ApplyDiff` call, however, tells the driver to add the files from the pulled tar to A. Layers B-D are each populated with two steps: a copy from the parent (via `Create`), and the addition of files from the tar (via `ApplyDiff`). After step 8, the pull is complete, and Docker is ready to create a container. It first creates a read-only layer E-init, to which it adds a few small initialization files, and then it creates E, the file system the container will use as its root.

2.3 AUFS Driver Implementation

The AUFS storage driver is a common default for Docker distributions. This driver is based on the AUFS file system (Another Union File System). Union file systems do not store data directly on disk, but rather use another file system (*e.g.*, ext4) as underlying storage.

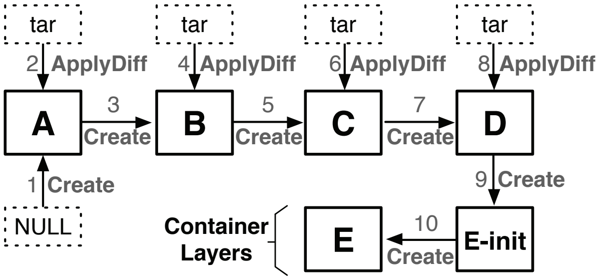


Figure 2: **Cold Run Example.** *The driver calls that are made when a four-layer image is pulled and run are shown. Each arrow represents a call (`Create` or `ApplyDiff`), and the nodes to which an arrow connects indicate arguments to the call. Thick-bordered boxes represent layers. Integers indicate the order in which functions are called.*

A union mount point provides a view of multiple directories in the underlying file system. AUFS is mounted with a list of directory paths in the underlying file system. During path resolution, AUFS iterates through the list of directories; the first directory to contain the path being resolved is chosen, and the inode from that directory is used. AUFS supports special *whiteout* files to make it appear that certain files in lower layers have been deleted; this technique is analogous to deletion markers in other layered systems (*e.g.*, LSM databases [29]). AUFS also supports COW (copy-on-write) at file granularity; upon write, files in lower layers are copied to the top layer before the write is allowed to proceed.

The AUFS driver takes advantage the AUFS file system’s layering and copy-on-write capabilities while also accessing the file system underlying AUFS directly. The driver creates a new directory in the underlying file system for each layer it stores. An `ApplyDiff` simple untars the archived files into the layer’s directory. Upon a `Get` call, the driver uses AUFS to create a unioned view of a layer and its ancestors. The driver uses AUFS’s COW to efficiently copy layer data when `Create` is called. Unfortunately, as we will see, COW at file granularity has some performance problems (§4.3).

3 HelloBench

We present HelloBench, a new benchmark designed to exercise container startup. HelloBench directly executes Docker commands, so pushes, pulls, and runs can be measured independently. The benchmark consists of two parts: (1) a collection of container images and (2) a test harness for executing simple tasks in said containers. The images were the latest available from the Docker Hub library [3] as of June 1, 2015. HelloBench consists of 57 images of the 72 available at the time. We selected images that were runnable with minimal configuration and do not depend on other containers. For example, WordPress is not included because a WordPress container depends on a separate MySQL container.

Linux Distro:	alpine, busybox, centos, cirros, crux, debian, fedora, mageia, opensuse, oraclelinux, ubuntu, ubuntu-debootstrap, ubuntu-upstart
Database:	cassandra, crate, elasticsearch, mariadb, mongo, mysql, percona, postgres, redis, rethinkdb
Language:	clojure, gcc, golang, haskell, hylang, java, jruby, julia, mono, perl, php, pypy, python, r-base, rakudo-star, ruby, thrift
Web Server:	glassfish, httpd, jetty, nginx, php-zendserver, tomcat
Web Framework:	django, iojs, node, rails
Other:	drupal, ghost, hello-world, jenkins, rabbitmq, registry, sonarqube

Table 2: **HelloBench Workloads.** *HelloBench runs 57 different container images pulled from the Docker Hub.*

Table 2 lists the images used by HelloBench. We divide the images into six broad categories as shown. Some classifications are somewhat subjective; for example, the Django image contains a web server, but most would probably consider it a web framework.

The HelloBench harness measures startup time by either running the simplest possible task in the container or waiting until the container reports readiness. For the language containers, the task typically involves compiling or interpreting a simple "hello world" program in the applicable language. The Linux distro images execute a very simple shell command, typically "`echo hello`". For long-running servers (particularly databases and web servers), HelloBench measures the time until the container writes an "up and ready" (or similar) message to standard out. For particularly quiet servers, an exposed port is polled until there is a response.

HelloBench images each consist of many layers, some of which are shared between containers. Figure 3 shows the relationships between layers. Across the 57 images, there are 550 nodes and 19 roots. In some cases, a tagged image serves as a base for other tagged images (*e.g.*, "ruby" is a base for "rails"). Only one image consists of a single layer: "alpine", a particularly lightweight Linux distribution. Application images are often based on non-latest Linux distribution images (*e.g.*, older versions of Debian); that is why multiple images will often share a common base that is not a solid black circle.

In order to evaluate how representative HelloBench is of commonly used images, we counted the number of pulls to every Docker Hub library image [3] on January 15, 2015 (7 months after the original HelloBench images were pulled). During this time, the library grew from 72 to 94 images. Figure 4 shows pulls to the 94 images, broken down by HelloBench category. HelloBench is representative of popular images, accounting for 86% of all pulls. Most pulls are to Linux distribution bases (*e.g.*, BusyBox and Ubuntu). Databases (*e.g.*, Redis and MySQL) and web servers (*e.g.*, nginx) are also popular.

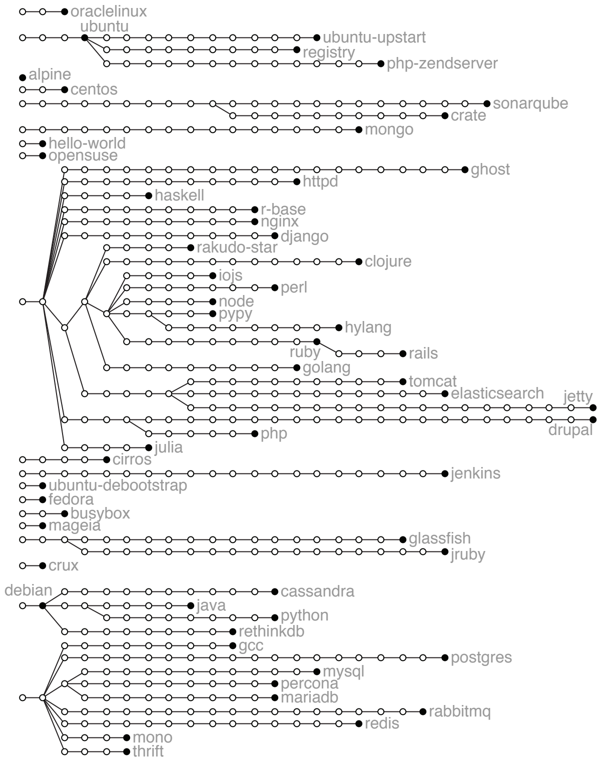


Figure 3: **HelloBench Hierarchy.** *Each circle represents a layer. Filled circles represent layers tagged as runnable images. Deeper layers are to the left.*

4 Workload Analysis

In this section, we analyze the behavior and performance of the HelloBench workloads, asking four questions: how large are the container images, and how much of that data is necessary for execution (§4.1)? How long does it take to push, pull, and run the images (§4.2)? How is image data distributed across layers, and what are the performance implications (§4.3)? And how similar are access patterns across different runs (§4.4)?

All performance measurements are taken from a virtual machine running on an PowerEdge R720 host with 2 GHz Xeon CPUs (E5-2620). The VM is provided 8 GB of RAM, 4 CPU cores, and a virtual disk backed by a Tintri T620 [1]. The server and VMstore had no other load during the experiments.

4.1 Image Data

We begin our analysis by studying the HelloBench images pulled from the Docker Hub. For each image, we take three measurements: its compressed size, uncompressed size, and the number of bytes read from the image when HelloBench executes. We measure reads by running the workloads over a block device traced with `blktrace` [11]. Figure 5 shows a CDF of these three numbers. We observe that only 20 MB of data is read on median, but the median image is 117 MB compressed and 329 MB uncompressed.

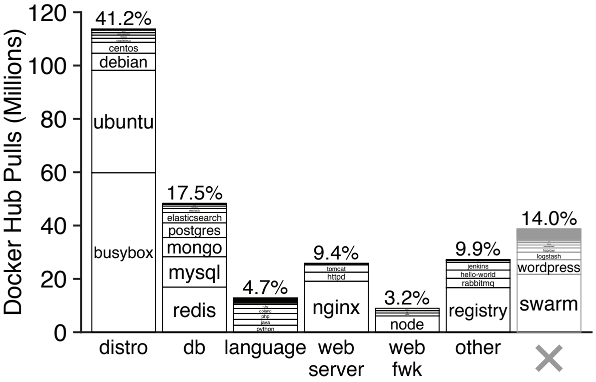


Figure 4: **Docker Hub Pulls.** *Each bar represents the number of pulls to the Docker Hub library, broken down by category and image. The far-right gray bar represents pulls to images in the library that are not run by HelloBench.*

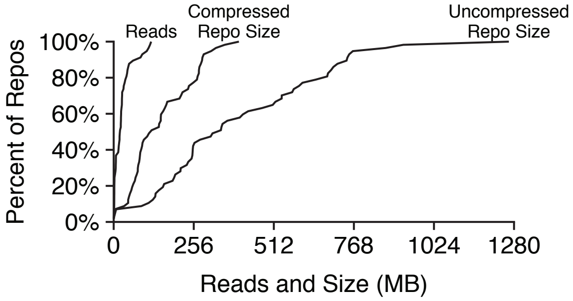


Figure 5: **Data Sizes (CDF).** *Distributions are shown for the number of reads in the HelloBench workloads and for the uncompressed and compressed sizes of the HelloBench images.*

We break down the read and size numbers by category in Figure 6. The largest relative waste is for distro workloads (30× and 85× for compressed and uncompressed respectively), but the absolute waste is also smallest for this category. Absolute waste is highest for the language and web framework categories. Across all images, only 27 MB is read on average; the average uncompressed image is 15× larger, indicating only 6.4% of image data is needed for container startup.

Although Docker images are much smaller when compressed as gzip archives, this format is not suitable for running containers that need to modify data. Thus, workers typically store data uncompressed, which means that compression reduces network I/O but not disk I/O. Deduplication is a simple alternative to compression that is suitable for updates. We scan HelloBench images for redundancy between blocks of files to compute the effectiveness of deduplication. Figure 7 compares gzip compression rates to deduplication, at both file and block (4 KB) granularity. Bars represent rates over single images. Whereas gzip achieves rates between 2.3 and 2.7, deduplication does poorly on a per-image basis. Deduplication across all images, however, yields rates of 2.6 (file granularity) and 2.8 (block granularity).

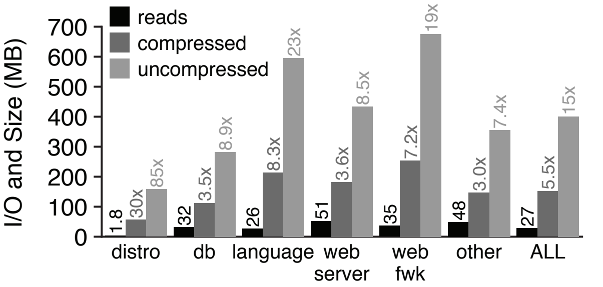


Figure 6: **Data Sizes (By Category).** *Averages are shown for each category. The size bars are labeled with amplification factors, indicating the amount of transferred data relative to the amount of useful data (i.e., the data read).*

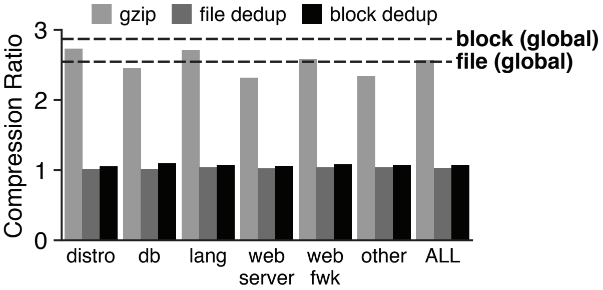


Figure 7: **Compression and Deduplication Rates.** *The y-axis represents the ratio of the size of the raw data to the size of the compressed or deduplicated data. The bars represent per-image rates. The lines represent rates of global deduplication across the set of all images.*

Implications: the amount of data read during execution is much smaller than the total image size, either compressed or uncompressed. Image data is sent over the network compressed, then read and written to local storage uncompressed, so overheads are high for both network and disk. One way to decrease overheads would be to build leaner images with fewer installed packages. Alternatively, image data could be lazily pulled as a container needs it. We also saw that global block-based deduplication is an efficient way to represent image data, even compared to gzip compression.

4.2 Operation Performance

Once built, containerized applications are often deployed as follows: the developer *pushes* the application image once to a central registry, a number of workers *pull* the image, and each worker *runs* the application. We measure the latency of these operations with HelloBench, reporting CDFs in Figure 8. Median times for push, pull, and run are 61, 16, and 0.97 seconds respectively.

Figure 9 breaks down operation times by workload category. The pattern holds in general: runs are fast while pushes and pulls are slow. Runs are fastest for the distro and language categories (0.36 and 1.9 seconds re-

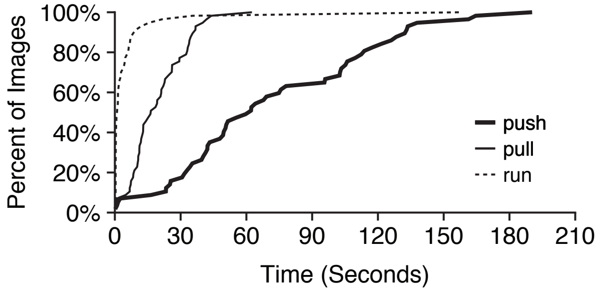


Figure 8: **Operation Performance (CDF).** *A distribution of push, pull, and run times for HelloBench are shown for Docker with the AUFS storage driver.*

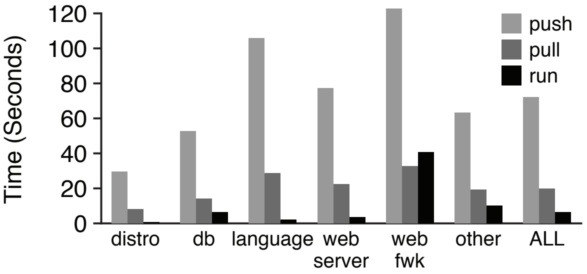


Figure 9: **Operation Performance (By Category).** *Averages are shown for each category.*

spectively). The average times for push, pull, and run are 72, 20, and 6.1 seconds respectively. Thus, 76% of startup time will be spent on pull when starting a new image hosted on a remote registry.

As pushes and pulls are slowest, we want to know whether these operations are merely high latency, or whether they are also costly in a way that limits throughput even if multiple operations run concurrently. To study scalability, we concurrently push and pull varying numbers of artificial images of varying sizes. Each image contains a single randomly generated file. We use artificial images rather than HelloBench images in order to create different equally-sized images. Figure 10 shows that the total time scales roughly linearly with the number of images and image size. Thus, pushes and pulls are not only high-latency, they consume network and disk resources, limiting scalability.

Implications: container startup time is dominated by pulls; 76% of the time spent on a new deployment will be spent on the pull. Publishing images with push will be painfully slow for programmers who are iteratively developing their application, though this is likely a less frequent case than multi-deployment of an already published image. Most push work is done by the storage driver's `Diff` function, and most pull work is done by the `ApplyDiff` function (§2.2). Optimizing these driver functions would improve distribution performance.

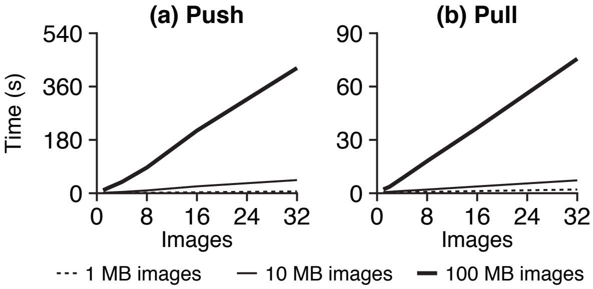


Figure 10: **Operation Scalability.** *A varying number of artificial images (x-axis), each containing a random file of a given size, are pushed or pulled simultaneously. The time until all operations are complete is reported (y-axis).*

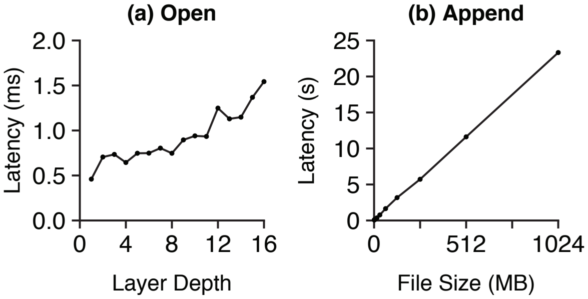


Figure 11: **AUFS Performance.** *Left: the latency of the `open` system call is shown as a function of the layer depth of the file. Right: the latency of a one-byte append is shown as a function of the size of the file that receives the write.*

4.3 Layers

Image data is typically split across a number of layers. The AUFS driver composes the layers of an image at runtime to provide a container a complete view of the file system. In this section, we study the performance implications of layering and the distribution of data across layers. We start by looking at two performance problems (Figure 11) to which layered file systems are prone: lookups to deep layers and small writes to non-top layers.

First, we create (and compose with AUFS) 16 layers, each containing 1K empty files. Then, with a cold cache, we randomly open 10 files from each layer, measuring the open latency. Figure 11a shows the result (an average over 100 runs): there is a strong correlation between layer depth and latency. Second, we create two layers, the bottom of which contains large files of varying sizes. We measure the latency of appending one byte to a file stored in the bottom layer. As shown by Figure 11b, the latency of small writes correspond to the file size (not the write size), as AUFS does COW at file granularity. Before a file is modified, it is copied to the topmost layer, so writing one byte can take over 20 seconds. Fortunately, small writes to lower layers induce a one-time cost per container; subsequent writes will be faster because the large file will have been copied to the top layer.

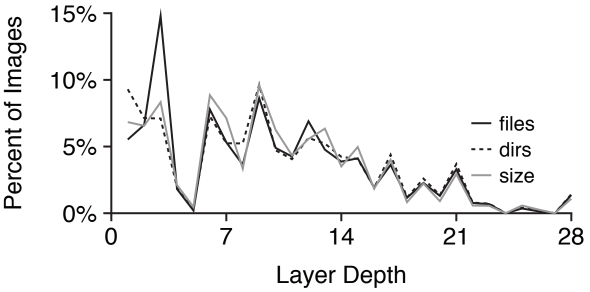


Figure 12: **Data Depth.** *The lines show mass distribution of data across image layers in terms of number of files, number of directories, and bytes of data in files.*

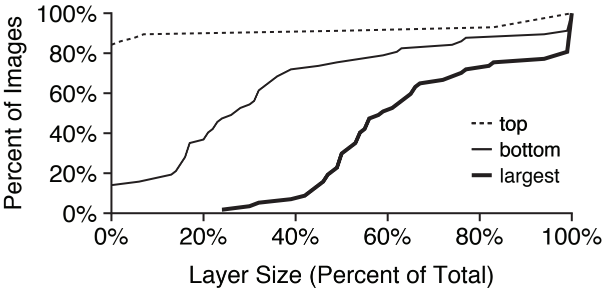


Figure 13: **Layer Size (CDF).** *The size of a given layer is measured relative to the total image size (x-axis), and the distribution of relative sizes is shown. The plot considers the topmost layer, bottommost layer, and whichever layer happens to be largest. All measurements are in terms of file bytes.*

Having considered how layer depth corresponds with performance, we now ask, *how deep is data typically stored for the HelloBench images?* Figure 12 shows the percentage of total data (in terms of number of files, number of directories, and size in bytes) at each depth level. The three metrics roughly correspond. Some data is as deep as level 28, but mass is more concentrated to the left. Over half the bytes are at depth of at least nine.

We now consider the variance in how data is distributed across layers, measuring, for each image, what portion (in terms of bytes) is stored in the topmost layer, bottommost layer, and whatever layer is largest. Figure 13 shows the distribution: for 79% of images, the topmost layer contains 0% of the image data. In contrast, 27% of the data resides in the bottommost layer in the median case. A majority of the data typically resides in a single layer.

Implications: for layered file systems, data stored in deeper layers is slower to access. Unfortunately, Docker images tend to be deep, with at least half of file data at depth nine or greater. Flattening layers is one technique to avoid these performance problems; however, flattening could potentially require additional copying and void the other COW benefits that layered file systems provide.

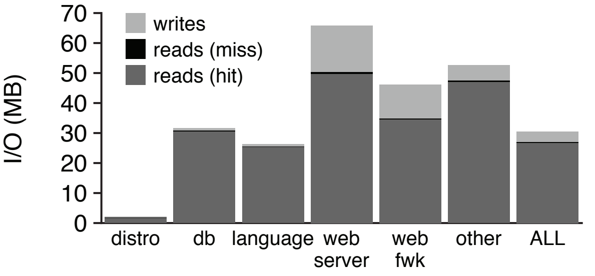


Figure 14: **Repeated I/O.** *The bars represent total I/O done for the average container workload in each category. Bar sections indicate read/write ratios. Reads that could have potentially been serviced by a cache populated by previous container execution are dark gray.*

4.4 Caching

We now consider the case where the same worker runs the same image more than once. In particular, we want to know whether I/O from the first execution can be used to prepopulate a cache to avoid I/O on subsequent runs. Towards this end, we run every HelloBench workload twice consecutively, collecting block traces each time. We compute the portion of reads during the second run that could potentially benefit from cache state populated by reads during the first run.

Figure 14 shows the reads and writes for the second run. Reads are broken into hits and misses. For a given block, only the first read is counted (we want to study the workload itself, not the characteristics of the specific cache beneath which we collected the traces). Across all workloads, the read/write ratio is 88/12. For distro, database, and language workloads, the workload consists almost completely of reads. Of the reads, 99% could potentially be serviced by cached data from previous runs.

Implications: The same data is often read during different runs of the same image, suggesting cache sharing will be useful when the same image is executed on the same machine many times. In large clusters with many containerized applications, repeated executions will be unlikely unless container placement is highly restricted. Also, other goals (*e.g.*, load balancing and fault isolation) may make colocation uncommon. However, repeated executions are likely common for containerized utility programs (*e.g.*, python or gcc) and for applications running in small clusters. Our results suggest these latter scenarios would benefit from cache sharing.

5 Slacker

In this section, we describe Slacker, a new Docker storage driver. Our design is based on our analysis of container workloads and five goals: (1) make pushes and pulls very fast, (2) introduce no slowdown for long-running containers, (3) reuse existing storage systems whenever possible, (4) utilize the powerful primitives

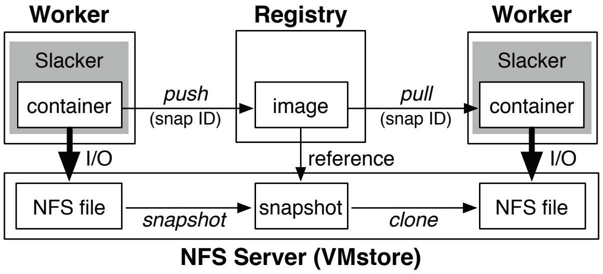


Figure 15: **Slacker Architecture.** *Most of our work was in the gray boxes, the Slacker storage plugin. Workers and registries represent containers and images as files and snapshots respectively on a shared Tintri VMstore server.*

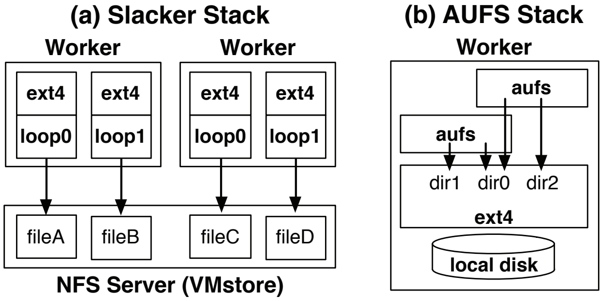


Figure 16: **Driver Stacks.** *Slacker uses one ext4 file system per container. AUFS containers share one ext4 instance.*

provided by a modern storage server, and (5) make no changes to the Docker registry or daemon except in the storage-driver plugin (§2.2).

Figure 15 illustrates the architecture of a Docker cluster running Slacker. The design is based on centralized NFS storage, shared between all Docker daemons and registries. Most of the data in a container is not needed to execute the container, so Docker workers only fetch data lazily from shared storage as needed. For NFS storage, we use a Tintri VMstore server [6]. Docker images are represented by VMstore's read-only snapshots. Registries are no longer used as hosts for layer data, and are instead used only as name servers that associate image metadata with corresponding snapshots. Pushes and pulls no longer involve large network transfers; instead, these operations simply share snapshot IDs. Slacker uses VMstore `snapshot` to convert a container into a shareable image and `clone` to provision container storage based on a snapshot ID pulled from the registry. Internally, VMstore uses block-level COW to implement `snapshot` and `clone` efficiently.

Slacker's design is based on our analysis of container workloads; in particular, the following four design subsections (§5.1 to §5.4) correspond to the previous four analysis subsections (§4.1 to §4.4). We conclude by discussing possible modifications to the Docker framework itself that would provide better support for non-traditional storage drivers such as Slacker (§5.5).

5.1 Storage Layers

Our analysis revealed that only 6.4% of the data transferred by a pull is actually needed before a container can begin useful work (§4.1). In order to avoid wasting I/O on unused data, Slacker stores all container data on an NFS server (a Tintri VMstore) shared by all workers; workers lazily fetch only the data that is needed. Figure 16a illustrates the design: storage for each container is represented as a single NFS file. Linux loopbacks (§5.4) are used to treat each NFS file as a virtual block device, which can be mounted and unmounted as a root file system for a running container. Slacker formats each NFS file as an ext4 file system.

Figure 16b compares the Slacker stack with the AUFS stack. Although both use ext4 (or some other local file system) as a key layer, there are three important differences. First, ext4 is backed by a network disk in Slacker, but by a local disk with AUFS. Thus, Slacker can lazily fetch data over the network, while AUFS must copy all data to the local disk before container startup.

Second, AUFS does COW above ext4 at the file level and is thus susceptible to the performance problems faced by layered file systems (§4.3). In contrast, Slacker layers are effectively flattened at the file level. However, Slacker still benefits from COW by utilizing block-level COW implemented within VMstore (§5.2). Furthermore, VMstore deduplicates identical blocks internally, providing further space savings between containers running on different Docker workers.

Third, AUFS uses different directories of a single ext4 instance as storage for containers, whereas Slacker backs each container by a different ext4 instance. This difference presents an interesting tradeoff because each ext4 instance has its own journal. With AUFS, all containers will share the same journal, providing greater efficiency. However, journal sharing is known to cause priority inversion that undermines QoS guarantees [48], an important feature of multi-tenant platforms such as Docker. Internal fragmentation [10, Ch. 17] is another potential problem when NFS storage is divided into many small, non-full ext4 instances. Fortunately, VMstore files are sparse, so Slacker does not suffer from this issue.

5.2 VMstore Integration

Earlier, we found that Docker pushes and pulls are quite slow compared to runs (§4.2). Runs are fast because storage for a new container is initialized from an image using the COW functionality provided by AUFS. In contrast, push and pull are slow with traditional drivers because they require copying large layers between different machines, so AUFS's COW functionality is not usable. Unlike other Docker drivers, Slacker is built on shared storage, so it is conceptually possible to do COW sharing between daemons and registries.

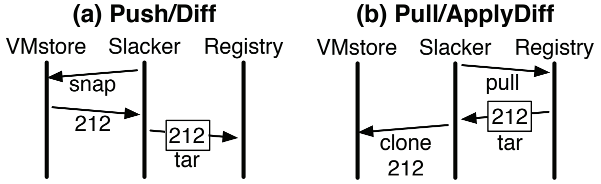


Figure 17: **Push/Pull Timelines.** *Slacker implements* `Diff` *and* `ApplyDiff` *with* `snapshot` *and* `clone` *operations.*

Fortunately, VMstore extends its basic NFS interface with an auxiliary REST-based API that, among other things, includes two related COW functions, `snapshot` and `clone`. The `snapshot` call creates a read-only snapshot of an NFS file, and `clone` creates an NFS file from a snapshot. Snapshots do not appear in the NFS namespace, but do have unique IDs. File-level snapshot and clone are powerful primitives that have been used to build more efficient journaling, deduplication, and other common storage operations [46]. In Slacker, we use `snapshot` and `clone` to implement `Diff` and `ApplyDiff` respectively. These driver functions are respectively called by Docker push and pull operations (§2.2).

Figure 17a shows how a daemon running Slacker interacts with a VMstore and Docker registry upon push. Slacker asks VMstore to create a snapshot of the NFS file that represents the layer. VMstore takes the snapshot, and returns a snapshot ID (about 50 bytes), in this case “212”. Slacker embeds the ID in a compressed tar file and sends it to the registry. Slacker embeds the ID in a tar for backwards compatibility: an unmodified registry expects to receive a tar file. A pull, shown in Figure 17b, is essentially the inverse. Slacker receives a snapshot ID from the registry, from which it can clone NFS files for container storage. Slacker’s implementation is fast because (a) layer data is never compressed or uncompressed, and (b) layer data never leaves the VMstore, so only metadata is sent over the network.

The names “`Diff`” and “`ApplyDiff`” are slight misnomers given Slacker’s implementation. In particular, `Diff(A, B)` is supposed to return a delta from which another daemon, which already has A, could reconstruct B. With Slacker, layers are effectively flattened at the namespace level. Thus, instead of returning a delta, `Diff(A, B)` returns a reference from which another worker could obtain a clone of B, with or without A.

Slacker is partially compatible with other daemons running non-Slacker drivers. When Slacker pulls a tar, it peeks at the first few bytes of the streamed tar before processing it. If the tar contains layer files (instead of an embedded snapshot), Slacker falls back to simply decompressing instead cloning. Thus, Slacker can pull images that were pushed by other drivers, albeit slowly. Other drivers, however, will not be able to pull Slacker images, because they will not know how to process the snapshot ID embedded in the tar file.

5.3 Optimizing Snapshot and Clone

Images often consist of many layers, with over half the HelloBench data being at a depth of at least nine (§4.3). Block-level COW has inherent performance advantages over file-level COW for such data, as traversing block-mapping indices (which may be flattened) is simpler than iterating over the directories of an underlying file system.

However, deeply-layered images still pose a challenge for Slacker. As discussed (§5.2), Slacker layers are flattened, so mounting any one layer will provide a complete view of a file system that could be used by a container. Unfortunately, the Docker framework has no notion of flattened layers. When Docker pulls an image, it fetches all the layers, passing each to the driver with `ApplyDiff`. For Slacker, the topmost layer alone is sufficient. For 28-layer images (*e.g.*, jetty), the extra clones are costly.

One of our goals was to work within the existing Docker framework, so instead of modifying the framework to eliminate the unnecessary driver calls, we optimize them with *lazy cloning*. We found that the primary cost of a pull is not the network transfer of the snapshot tar files, but the VMstore clone. Although clones take a fraction of a second, performing 28 of them negatively impacts latency. Thus, instead of representing every layer as an NFS file, Slacker (when possible) represents them with a piece of local metadata that records a snapshot ID. `ApplyDiff` simply sets this metadata instead of immediately cloning. If at some point Docker calls `Get` on that layer, Slacker will at that point perform a real `clone` before the mount.

We also use the snapshot-ID metadata for *snapshot caching*. In particular, Slacker implements `Create`, which makes a logical copy of a layer (§2.2) with a snapshot immediately followed by a clone (§5.2). If many containers are created from the same image, `Create` will be called many times on the same layer. Instead of doing a snapshot for each `Create`, Slacker only does it the first time, reusing the snapshot ID subsequent times. The snapshot cache for a layer is invalidated if the layer is mounted (once mounted, the layer could change, making the snapshot outdated).

The combination of snapshot caching and lazy cloning can make `Create` very efficient. In particular, copying from a layer A to layer B may only involve copying from A’s snapshot cache entry to B’s snapshot cache entry, with no special calls to VMstore. In Figure 2 from the background section (§2.2), we showed the 10 `Create` and `ApplyDiff` calls that occur for the pull and run of a simple four-layer image. Without lazy caching and snapshot caching, Slacker would need to perform 6 snapshots (one for each `Create`) and 10 clones (one for each `Create` or `ApplyDiff`). With our optimizations, Slacker only needs to do one snapshot and two clones. In step 9, `Create` does a lazy clone, but Docker calls Get on the

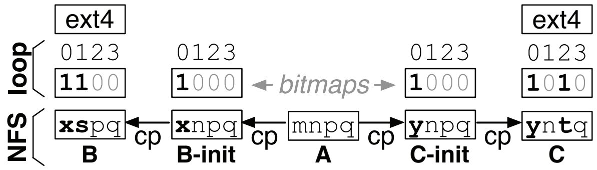


Figure 18: **Loopback Bitmaps.** *Containers B and C are started from the same image, A. Bitmaps track differences.*

E-init layer, so a real clone must be performed. For step 10, `Create` must do both a snapshot and clone to produce and mount layer E as the root for a new container.

5.4 Linux Kernel Modifications

Our analysis showed that multiple containers started from the same image tend to read the same data, suggesting cache sharing could be useful (§4.4). One advantage of the AUFS driver is that COW is done above an underlying file system. This means that different containers may warm and utilize the same cache state in that underlying file system. Slacker does COW within VMstore, beneath the level of the local file system. This means that two NFS files may be clones (with a few modifications) of the same snapshot, but cache state will not be shared, because the NFS protocol is not built around the concept of COW sharing. Cache deduplication could help save cache space, but this would not prevent the initial I/O. It would not be possible for deduplication to realize two blocks are identical until both are transferred over the network from the VMstore. In this section, we describe our technique to achieve sharing in the Linux page cache at the level of NFS files.

In order to achieve client-side cache sharing between NFS files, we modify the layer immediately above the NFS client (*i.e.*, the loopback module) to add awareness of VMstore snapshots and clones. In particular, we use bitmaps to track differences between similar NFS files. All writes to NFS files are via the loopback module, so the loopback module can automatically update the bitmaps to record new changes. Snapshots and clones are initiated by the Slacker driver, so we extend the loopback API so that Slacker can notify the module of COW relationships between files.

Figure 18 illustrates the technique with a simple example: two containers, B and C, are started from the same image, A. When starting the containers, Docker first creates two init layers (B-init and C-init) from the base (A). Docker creates a few small init files in these layers. Note that the "m" is modified to an "x" and "y" in the init layers, and that the zeroth bits are flipped to "1" to mark the change. Docker the creates the topmost container layers, B and C from B-init and C-init. Slacker uses the new loopback API to copy the B-init and C-init bitmaps to B and C respectively. As shown, the B and C bitmaps accumulate more mutations as the containers run and write data. Docker does not explicitly differentiate init layers from other layers as part of the API, but Slacker can infer layer type because Docker happens to use an "-init" suffix for the names of init layers.

Now suppose that container B reads block 3. The loopback module sees an unmodified "0" bit at position 3, indicating block 3 is the same in files B and A. Thus, the loopback module sends the read to A instead of B, thus populating A's cache state. Now suppose C reads block 3. Block 3 of C is also unmodified, so the read is again redirected to A. Now, C can benefit from the cache state of A, which B populated with its earlier read.

Of course, for blocks where B and C differ from A, it is important for correctness that reads are not redirected. Suppose B reads block 1 and then C reads from block 1. In this case, B's read will not populate the cache since B's data differs from A. Similarly, suppose B reads block 2 and then C reads from block 2. In this case, C's read will not utilize the cache since C's data differs from A.

5.5 Docker Framework Discussion

One our goals was to make no changes to the Docker registry or daemon, except within the pluggable storage driver. Although the storage-driver interface is quite simple, it proved sufficient for our needs. There are, however, a few changes to the Docker framework that would have enabled a more elegant Slacker implementation. First, it would be useful for compatibility between drivers if the registry could represent different layer formats (§5.2). Currently, if a non-Slacker layer pulls a layer pushed by Slacker, it will fail in an unfriendly way. Format tracking could provide a friendly error message, or, ideally, enable hooks for automatic format conversion. Second, it would be useful to add the notion of flattened layers. In particular, if a driver could inform the framework that a layer is flat, Docker would not need to fetch ancestor layers upon a pull. This would eliminate our need for lazy cloning and snapshot caching (§5.3). Third, it would be convenient if the framework explicitly identified init layers so Slacker would not need to rely on layer names as a hint (§5.4).

6 Evaluation

We use the same hardware for evaluation as we did for our analysis (§4). For a fair comparison, we also use the same VMstore for Slacker storage that we used for the virtual disk of the VM running the AUFS experiments.

6.1 HelloBench Workloads

Earlier, we saw that with HelloBench, push and pull times dominate while run times are very short (Figure 9). We repeat that experiment with Slacker, presenting the new results alongside the AUFS results in Figure 19. On average, the push phase is 153× faster and the pull phase is 72× faster, but the run phase is 17% slower (the AUFS pull phase warms the cache for the run phase).

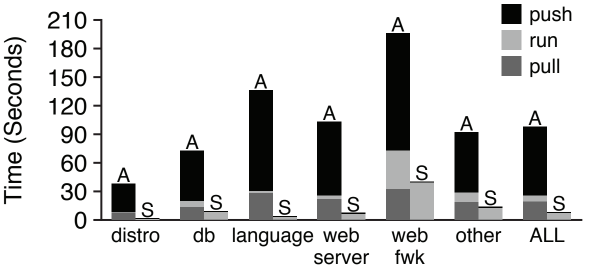


Figure 19: **AUFS vs. Slacker (Hello).** *Average push, run, and pull times are shown for each category. Bars are labeled with an "A" for AUFS or "S" for Slacker.*

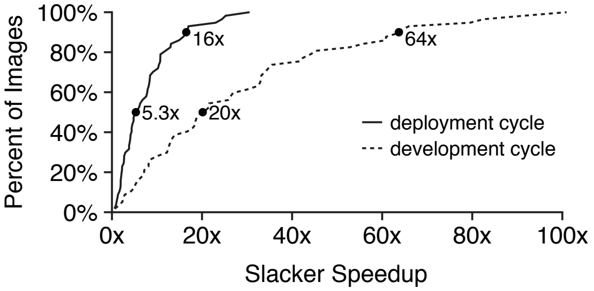


Figure 20: **Slacker Speedup.** *The ratio of AUFS-driver time to Slacker time is measured, and a CDF shown across HelloBench workloads. Median and 90th-percentile speedups are marked for the development cycle (push, pull, and run), and the deployment cycle (just pull and run).*

Different Docker operations are utilized in different scenarios. One use case is the *development cycle*: after each change to code, a developer pushes the application to a registry, pulls it to multiple worker nodes, and then runs it on the nodes. Another is the *deployment cycle*: an infrequently-modified application is hosted by a registry, but occasional load bursts or rebalancing require a pull and run on new workers. Figure 20 shows Slacker's speedup relative to AUFS for these two cases. For the median workload, Slacker improves startup by 5.3× and 20× for the deployment and development cycles respectively. Speedups are highly variable: nearly all workloads see at least modest improvement, but 10% of workloads improve by at least 16× and 64× for deployment and development respectively.

6.2 Long-Running Performance

In Figure 19, we saw that while pushes and pulls are much faster with Slacker, runs are slower. This is expected, as runs start before any data is transferred, and binary data is only lazily transferred as needed. We now run several long-running container experiments; our goal is to show that once AUFS is done pulling all image data and Slacker is done lazily loading hot image data, AUFS and Slacker have equivalent performance.

For our evaluation, we select two databases and two web servers. For all experiments, we execute for five minutes, measuring operations per second. Each experiment starts with a pull. We evaluate the PostgreSQL database using pgbench, which is "loosely based on TPC-B" [5]. We evaluate Redis, an in-memory database, using a custom benchmark that gets, sets, and updates keys with equal frequency. We evaluate the Apache web server, using the `wrk` [4] benchmark to repeatedly fetch a static page. Finally, we evaluate io.js, a JavaScript-based web server similar to node.js, using the `wrk` benchmark to repeatedly fetch a dynamic page.

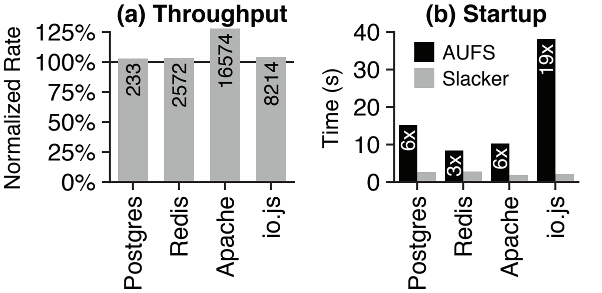


Figure 21: **Long-Running Workloads.** *Left: the ratio of Slacker's to AUFS's throughput is shown; startup time is included in the average. Bars are labeled with Slacker's average operations/second. Right: startup delay is shown.*

Figure 21a shows the results. AUFS and Slacker usually provide roughly equivalent performance, though Slacker is somewhat faster for Apache. Although the drivers are similar with regard to long-term performance, Figure 21b shows Slacker containers start processing requests 3-19× sooner than AUFS.

6.3 Caching

We have shown that Slacker provides much faster startup times relative to AUFS (when a pull is required) and equivalent long-term performance. One scenario where Slacker is at a disadvantage is when the same short-running workload is run many times on the same machine. For AUFS, the first run will be slow (as a pull is required), but subsequent runs will be fast because the image data will be stored locally. Moreover, COW is done locally, so multiple containers running from the same start image will benefit from a shared RAM cache.

Slacker, on the other hand, relies on the Tintri VMstore to do COW on the server side. This design enables rapid distribution, but one downside is that NFS clients are not naturally aware of redundancies between files without our kernel changes. We compare our modified loopback driver (§5.4) to AUFS as a means of sharing cache state. To do so, we run each HelloBench workload twice, measuring the latency of the second run (after the first has warmed the cache). We compare AUFS to Slacker, with and without kernel modifications.

Figure 22 shows a CDF of run times for all the workloads with the three systems (note: these numbers were

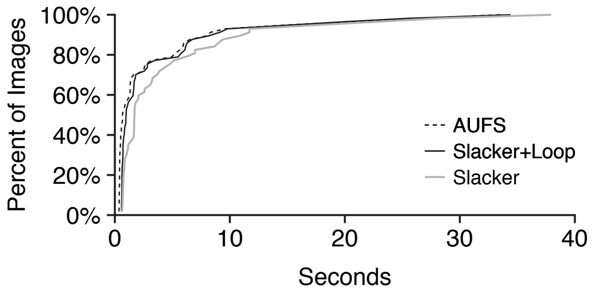


Figure 22: **Second Run Time (CDF).** *A distribution of run times are shown for the AUFS driver and for Slacker, both with and without use of the modified loopback driver.*

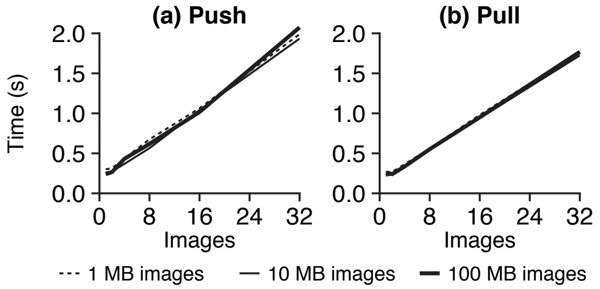


Figure 23: **Operation Scalability.** *A varying number of artificial images (x-axis), each containing a random file of a given size, are pushed or pulled simultaneously. The time until all operations are complete is reported (y-axis).*

collected with a VM running on a ProLiant DL360p Gen8). Although AUFS is still fastest (with median runs of 0.67 seconds), the kernel modifications significantly speed up Slacker. The median run time of Slacker alone is 1.71 seconds; with kernel modifications to the loopback module it is 0.97 seconds. Although Slacker avoids unnecessary network I/O, the AUFS driver can directly cache ext4 file data, whereas Slacker caches blocks beneath ext4, which likely introduces some overhead.

6.4 Scalability

Earlier (§4.2), we saw that AUFS scales poorly for pushes and pulls with regard to image size and the number of images being manipulated concurrently. We repeat our earlier experiment (Figure 10) with Slacker, again creating synthetic images and pushing or pulling varying numbers of these concurrently.

Figure 23 shows the results: image size no longer matters as it does for AUFS. Total time still correlates with the number of images being processed simultaneously, but the absolute times are much better; even with 32 images, push and pull times are at most about two seconds. It is also worth noting that push times are similar to pull times for Slacker, whereas pushes were much more expensive for AUFS. This is because AUFS uses compression for its large data transfers, and compression is typically more costly than decompression.

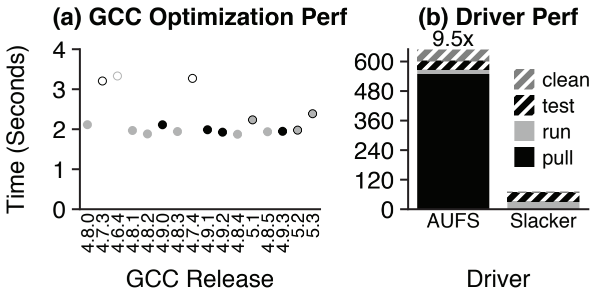


Figure 24: **GCC Version Testing.** *Left: run time of a C program doing vector arithmetic. Each point represents performance under a different GCC release, from 4.8.0 (Mar '13) to 5.3 (Dec '15). Releases in the same series have a common style (e.g., 4.8-series releases are solid gray). Right: performance of MultiMake is shown for both drivers. Time is broken into pulling the image, running the image (compiling), testing the binaries, and deleting the images from the local daemon.*

7 Case Study: MultiMake

When starting Dropbox, Drew Houston (co-founder and CEO) found that building a widely-deployed client involved a lot of "grungy operating-systems work" to make the code compatible with the idiosyncrasies of various platforms [18]. For example, some bugs would only manifest with the Swedish version of Windows XP Service Pack 3, whereas other very similar deployments (including the Norwegian version) would be unaffected. One way to avoid some of these bugs is to broadly test software in many different environments. Several companies provide containerized integration-testing services [33, 39], including for fast testing of web applications against dozens of releases of of Chrome, Firefox, Internet Explorer, and other browsers [36]. Of course, the breadth of such testing is limited by the speed at which different test environments can be provisioned.

We demonstrate the usefulness of fast container provisioning for testing with a new tool, MultiMake. Running MultiMake on a source directory builds 16 different versions of the target binary using the last 16 GCC releases. Each compiler is represented by a Docker image hosted by a central registry. Comparing binaries has many uses. For example, certain security checks are known to be optimized away by certain compiler releases [44]. MultiMake enables developers to evaluate the robustness of such checks across GCC versions.

Another use for MultiMake is to evaluate the performance of code snippets against different GCC versions, which employ different optimizations. As an example, we use MultiMake on a simple C program that does 20M vector arithmetic operations, as follows:

```
for (int i=0; i<256; i++) {
  a[i] = b[i] + c[i] * 3;
}
```

Figure 24a shows the result: most recent GCC releases optimize the vector operations well, but the but the code generated by the 4.6- and 4.7-series compilers takes about 50% longer to execute. GCC 4.8.0 produces fast code, even though it was released before some of the slower 4.6 and 4.7 releases, so some optimizations were clearly not backported. Figure 24b shows that collecting this data is 9.5× faster with Slacker (68 seconds) than with the AUFS driver (646 seconds), as most of the time is spent pulling with AUFS. Although all the GCC images have a common Debian base (which must only be pulled once), the GCC installations represent most of the data, which AUFS pulls every time. Cleanup is another operation that is more expensive for AUFS than Slacker. Deleting a layer in AUFS involves deleting thousands of small ext4 files, whereas deleting a layer in Slacker involves deleting one large NFS file.

The ability to rapidly run different versions of code could benefit other tools beyond MultiMake. For example, `git bisect` finds the commit that introduced a bug by doing a binary search over a range of commits [23]. Alongside container-based automated build systems [35], a bisect tool integrated with Slacker could very quickly search over a large number of commits.

8 Related Work

Work optimizing the multi-deployment of disk images is similar to ours, as the ext4-formatted NFS files used by Slacker resemble virtual-disk images. Hibler *et al*. [16] built Frisbee, a system that optimizes differential image updates by using techniques based on file-system awareness (*e.g.*, Frisbee does not consider blocks that are unused by the file system). Wartel *et al*. [45] compare multiple methods of lazily distributing virtual-machine images from a central repository (much like a Docker registry). Nicolae *et al*. [28] studied image deployment and found *"prepropagation is an expensive step, especially since only a small part of the initial VM is actually accessed."* They further built a distributed file system for hosting virtual machine images that supports lazy propagation of VM data. Zhe *et al*. [50] built Twinkle, a cloud-based platform for web applications that is designed to handle *"flash crowd events."* Unfortunately, virtual-machines tend to be heavyweight, as they note: *"virtual device creation can take a few seconds."*

Various cluster management tools provide container scheduling, including Kubernetes [2], Google's Borg [41], Facebook's Tupperware [26], Twitter's Aurora [21], and Apache Mesos [17]. Slacker is complementary to these systems; fast deployment gives cluster managers more flexibility, enabling cheap migration and fine-tuned load balancing.

A number of techniques bear resemblance to our strategy for sharing cache state and reducing redundant I/O. VMware ESX server [43] and Linux KSM [9] (Kernel Same-page Merging) both scan and deduplicate memory. While this technique saves cache space, it does not prevent initial I/O. Xingbo *et al*. [47] also observed the problem where reads to multiple nearly identical files cause avoidable I/O. They modified btrfs to index cache pages by disk location, thus servicing some block reads issued by btrfs with the page cache. Sapuntzakis *et al*. [32] use dirty bitmaps for VM images to identify a subset of the virtual-disk image blocks that must be transferred during migration. Lagar-Cavilla *et al*. [20] built a "VM fork" function that rapidly creates many clones of a running VM. Data needed by one clone is multicast to all the clones as a means of prefetch. Slacker would likely benefit from similar prefetching.

9 Conclusions

Fast startup has applications for scalable web services, integration testing, and interactive development of distributed applications. Slacker fills a gap between two solutions. Containers are inherently lightweight, but current management systems such as Docker and Borg are very slow at distributing images. In contrast, virtual machines are inherently heavyweight, but multi-deployment of virtual machine images has been thoroughly studied and optimized. Slacker provides highly efficient deployment for containers, borrowing ideas from VM image-management, such as lazy propagation, as well as introducing new Docker-specific optimizations, such as lazy cloning. With these techniques, Slacker speeds up the typical deployment cycle by 5× and development cycle by 20×. HelloBench and a snapshot [15] of the images we use for our experiments in this paper are available online: `https://github.com/Tintri/hello-bench`

10 Acknowledgements

We thank the anonymous reviewers and Atul Adya (our shepherd) for their tremendous feedback, as well as members of our research group at UW-Madison and coworkers at Tintri for their thoughts and comments on this work at various stages. We especially thank Zev Weiss, John Schmitt, Sean Chen, and Kieran Harty for their design suggestions and help with experiments.

This material was supported by funding from NSF grants CNS-1218405, CNS-1319405, CNS-1419199, and CNS-1421033, as well as generous donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Samsung, Tintri, Veritas, Seagate, and VMware as part of the WISDOM research institute sponsorship. Tyler Harter is supported by Tintri and an NSF Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

[1] Tintri VMstore(tm) T600 Series. http://www.tintri.com/sites/default/files/field/pdf/document/t600-datasheet_0.pdf, 2013.

[2] Kubernetes. http://kubernetes.io, August 2014.

[3] Docker Hub. https://hub.docker.com/u/library/, 2015.

[4] Modern HTTP Benchmarking Tool. https://github.com/wg/wrk/, 2015.

[5] pgbench. http://www.postgresql.org/docs/devel/static/pgbench.html, September 2015.

[6] Tintri Operating System. https://www.tintri.com/sites/default/files/field/pdf/whitepapers/tintri-os-datasheet-150701t10072.pdf, 2015.

[7] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, Washington, March 2008.

[8] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters.

[9] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the linux symposium*, pages 19–28, 2009.

[10] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[11] Jens Axboe, Alan D. Brunelle, and Nathan Scott. blktrace(8) - Linux man page. http://linux.die.net/man/8/blktrace, 2006.

[12] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[13] Jeremy Elson and Jon Howell. Handling Flash Crowds from Your Garage. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 171–184, Berkeley, CA, USA, 2008. USENIX Association.

[14] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'2006)*, Melbourne, Australia, Nov 2006.

[15] Tyler Harter. HelloBench. http://research.cs.wisc.edu/adsl/Software/hello-bench/, 2015.

[16] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, Scalable Disk Imaging with Frisbee. In *USENIX Annual Technical Conference, General Track*, pages 283–296, 2003.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, volume 11, pages 22–22, 2011.

[18] Drew Houston. https://www.youtube.com/watch?t=1278&v=NZINmtuTSu0, 2014.

[19] Michael Kerrisk and Eric W. Biederman. namespaces(7) - overview of Linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html, 2013.

[20] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12. ACM, 2009.

[21] Dave Lester. All about Apache Aurora. https://blog.twitter.com/2015/all-about-apache-aurora, 2015.

[22] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical Disentanglement in a Container-Based File System. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[23] Git Manpages. git-bisect(1) Manual Page. https://www.kernel.org/pub/software/scm/git/docs/git-bisect.html, 2015.

[24] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. Maestro: quality-of-service in large disk arrays.

[25] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. Linux Journal, Issue 239, March 2014.

[26] Aravind Narayanan. Tupperware: Containerized Deployment at Facebook. http://www.slideshare.net/Docker/aravindnarayanan-facebook140613153626phpapp02-37588997, 2014.

[27] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds.

[28] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel Antoniu. Going back and forth: Efficient multideployment and multisnapshotting on clouds. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 147–158. ACM, 2011.

[29] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[30] John Pescatore. Nimda Worm Shows You Can't Always Patch Fast Enough. https://www.gartner.com/doc/340962, September 2001.

[31] David Saff and Michael D Ernst. An Experimental Evaluation of Continuous Testing During Development. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 76–85. ACM, 2004.

[32] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, December 2002.

[33] Sunil Shah. Integration Testing with Mesos, Chronos and Docker. http://mesosphere.com/blog/2015/03/26/integration-testing-with-mesos-chronos-docker/, 2015.

[34] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, California, October 2012.

[35] Matt Soldo. Upgraded Autobuild System on Docker Hub. http://blog.docker.com/2015/11/upgraded-autobuild-docker-hub/, 2015.

[36] spoon.net. Containerized Selenium Testing. https://blog.spoon.net/running-a-selenium-grid-using-containers/, 2015.

[37] The Linux Community. LXC – Linux Containers, 2014.

[38] E. Thereska, A. Rowstron, H. Ballani, G. O'Shea, T. Karagiannis, 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 '13)*, Farmington, Pennsylvania, November 2013.

[39] Paul van der Ende. Fast and Easy Integration Testing with Docker and Overcast. http://blog.xebia.com/2014/10/13/fast-and-easy-integration-testing-with-docker-and-overcast/, 2014.

[40] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance Isolation: Sharing and Isolation in Shared-memory Multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 181–192, San Jose, California, October 1998.

[41] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[42] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[43] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[44] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 260–275. ACM, 2013.

[45] Romain Wartel, Tony Cass, Belmiro Moreira, Ewan Roche, Manuel Guijarro, Sebastien Goasguen, and Ulrich Schwickerath. Image distribution mechanisms in large scale cloud providers. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 112–117. IEEE, 2010.

[46] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, CA, February 2015.

[47] Xingbo Wu, Wenguang Wang, and Song Jiang. TotalCOW: Unleash the Power of Copy-On-Write for Thin-provisioned Containers. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, pages 15:1–15:7, New York, NY, USA, 2015. ACM.

[48] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-level I/O Scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 474–489, New York, NY, USA, 2015. ACM.

[49] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters.

[50] Jun Zhu, Zhefu Jiang, and Zhen Xiao. Twinkle: A Fast Resource Provisioning Mechanism for Internet Services. In *INFOCOM, 2011 Proceedings IEEE*, pages 802–810. IEEE, 2011.

sRoute: Treating the Storage Stack Like a Network

Ioan Stefanovici and Bianca Schroeder
University of Toronto

Greg O'Shea
Microsoft Research

Eno Thereska
Confluent and Imperial College London

Abstract

In a data center, an IO from an application to distributed storage traverses not only the network, but also several software stages with diverse functionality. This set of ordered stages is known as the storage or IO stack. Stages include caches, hypervisors, IO schedulers, file systems, and device drivers. Indeed, in a typical data center, the number of these stages is often larger than the number of network hops to the destination. Yet, while packet routing is fundamental to networks, no notion of IO routing exists on the storage stack. The path of an IO to an endpoint is predetermined and hard-coded. This forces IO with different needs (e.g., requiring different caching or replica selection) to flow through a one-size-fits-all IO stack structure, resulting in an ossified IO stack.

This paper proposes sRoute, an architecture that provides a routing abstraction for the storage stack. sRoute comprises a centralized control plane and "sSwitches" on the data plane. The control plane sets the forwarding rules in each sSwitch to route IO requests at runtime based on application-specific policies. A key strength of our architecture is that it works with unmodified applications and VMs. This paper shows significant benefits of customized IO routing to data center tenants (e.g., a factor of ten for tail IO latency, more than 60% better throughput for a customized replication protocol and a factor of two in throughput for customized caching).

1 Introduction

An application's IO stack is rich in stages providing compute, network, and storage functionality. These stages include guest OSes, file systems, hypervisors, network appliances, and distributed storage with caches and schedulers. Indeed, there are over 18+ types of stages on a typical data center IO stack [53]. Furthermore, most IO stacks support the injection of new stages with new functionality using filter drivers common in most OSes [18, 34, 38], or appliances over the network [48].

Controlling or programming how IOs flow through this stack is hard if not impossible, for tenants and service providers alike. Once an IO enters the system, the path to its endpoint is pre-determined and static. It must pass through all stages on the way to the endpoint. A new stage with new functionality means a longer path with added latency for every IO. As raw storage and networking speeds improve, the length of the IO stack is increasingly becoming a new bottleneck [43]. Furthermore, the IO stack stages have narrow interfaces and operate in isolation. Unlocking functionality often requires coordinating the functionality of multiple such stages. These reasons lead to applications running on a general-purpose IO stack that cannot be tuned to any of their specific needs, or to one-off customized implementations that require application and system rewrite.

This paper's main contribution is experimenting with applying a well-known networking primitive, *routing*, to the storage stack. IO routing provides the ability to dynamically change the path and destination of an IO, like a `read` or `write`, at runtime. Control plane applications use IO routing to provide customized data plane functionality for tenants and data center services.

Consider three specific examples of how routing is useful. In one example, a load balancing service selectively routes `write` requests to go to less-loaded servers, while ensuring `read` requests are always routed to the latest version of the data (§5.1). In another example, a control application provides per-tenant throughput versus latency tradeoffs for replication update propagation, by using IO routing to set a tenant's IO read- and write-set at runtime (§5.2). In a third example, a control application can route requests to per-tenant caches to maintain cache isolation (§5.3).

IO routing is challenging because the storage stack is stateful. Routing a `write` IO through one path to endpoint A and a subsequent `read` IO through a different path or to a different endpoint B needs to be mindful of application consistency needs. Another key challenge is data plane efficiency. Changing the path of an IO at runtime requires determining where on the data plane to insert sSwitches to minimize the number of times an IO traverses them, as well as to minimize IO processing times.

sRoute's approach builds on the IOFlow storage architecture [53]. IOFlow already provides a separate control plane for storage traffic and a logically centralized controller with global visibility over the data center topology. As an analogy to networking, sRoute builds on IOFlow just like software-defined networking (SDN) functions build on OpenFlow [35]. IOFlow also made a case for request routing. However, it only explored the concept of *bypassing* stages along the IO path, and did not consider the full IO routing spectrum where the path and endpoint can also change, leading to consistency concerns. This paper provides a more complete routing abstraction.

This paper makes the following contributions:

- We propose an IO routing abstraction for the IO stack.
- sRoute provides per-IO and per-flow routing configuration updates with strong semantic guarantees.
- sRoute provides an efficient control plane. It does so by distributing the control plane logic required for IO routing using *delegate functions*.
- We report on our experience in building three control applications using IO routing: tail latency control, replica set control, and file caching control.

The results of our evaluation demonstrate that data center tenants benefit significantly from IO stack customization. The benefits can be provided to today's unmodified tenant applications and VMs. Furthermore, writing specialized control applications is straightforward because they use a common IO routing abstraction.

2 Routing types and challenges

The data plane, or *IO stack* comprises all the stages an IO request traverses from an application until it reaches its destination. For example, a read to a file will traverse a guest OS' file system, buffer cache, scheduler, then similar stages in the hypervisor, followed by OSes, file systems, caches and device drivers on remote storage servers. We define per-IO routing in this context as the ability to control the IO's endpoint as well as the path to that endpoint. The first question is what the above definition means for storage semantics. A second question is whether IO routing is a useful abstraction.

To address the first question, we looked at a large set of storage system functionalities and distilled from them three types of IO routing that make sense semantically in the storage stack. Figure 1 illustrates these three types. In *endpoint* routing, IO from a source p to a destination file X is routed to another destination file Y. In *waypoint* routing, IOs from sources p and r to a file X are first routed to a specialized stage W. In *scatter* routing, IOs from p and r are routed to a subset of data replicas.

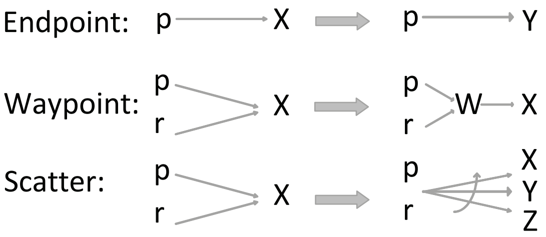


Figure 1: Three types of IO routing: endpoint, waypoint and scatter. p,r refer to sources such as VMs or containers. X,Y,Z are endpoints such as files. W represents a waypoint stage with specialized functionality, for example a file cache or scheduler.

	Functionality	How IO routing helps
E	Tail latency control Copy-on-write File versioning	Route IO to less loaded servers Route writes to new location Route IO to right version
W	Cache size guarantee Deadline policies	Route IO to specialized cache Route IO to specialized scheduler
S	Maximize throughput Minimize latency Logging/Debugging	Route `reads` to all replicas Route `writes` to replica subset Route selected IOs to loggers

Table 1: Examples of specialized functionality and the type of IO routing (E)ndpoint, (W)aypoint and (S)catter that enables them.

This paper makes the case that IO routing is a useful abstraction. We show that many specialized functions on the storage stack can be recast as routing problems. Our hypothesis when we started this work was that, because routing is inherently programmable and dynamic, we could substitute hard-coded one-off implementations with one common routing core. Table 1 shows a diverse set of such storage stack functionalities, categorized according to the type of IO routing that enables them.

Endpoint routing. Routes IO from a single-source application p to a file X to another file Y. The timing of the routing and operation semantics is dictated by the control logic. For example, write requests could go to the new endpoint and reads could be controlled to go to the old or new endpoints. Endpoint routing enables functionality such as improving *tail latency* [14, 41], *copy-on-write* [21, 42, 46], *file versioning* [37], and *data re-encoding* [1]. These policies have in common the need for a dynamic mechanism that changes the endpoint of new data and routes IO to the appropriate endpoint. Section §5.1 shows how we implement tail latency control using endpoint routing.

Waypoint routing. Routes IO from a multi-source application $\{p,r\}$ to a file X through an intermediate waypoint stage W. W could be a file cache or scheduler. Waypoint routing enables specialized appliance processing [48]. These policies need a dynamic mechanism to inject specialized waypoint stages or appliances along the stack and to selectively route IO to those stages. Section §5.3 shows how we implement file cache control using waypoint routing.

Scatter routing. Scatters IO from file X to additional endpoints Y and Z. The control logic dictates which subset of endpoints to read data from and write data to. Scatter routing enables specialized *replication* and *erasure coding* policies [33, 51] These policies have in common the need for a dynamic mechanism to choose which endpoint to write to and read from. This control enables programmable tradeoffs around throughput and update propagation latency. Section §5.2 shows how we implement replica set control using scatter routing.

2.1 Challenges

IO routing is challenging for several reasons:

Consistent system-wide configuration updates. IO routing requires a control-plane mechanism for changing the path of an IO request. The mechanism needs to coordinate the forwarding rules in each sSwitch in the data plane. Any configuration changes must not lead to system instability, where an IO's semantic guarantees are violated by having it flow through an incorrect path.

Metadata consistency. IO routing allows `read` and `write` IOs to be sent to potentially different endpoints. Several applications benefit from this flexibility. Some applications, however, have stricter consistency requirements and require, for example, that a `read` always follow the path of a previous `write`. A challenge is keeping track of the data's latest location. Furthermore, IO routing metadata needs to coexist consistently with metadata in the rest of the system. The guest file system, for example, has a mapping of files to blocks and the hypervisor has a mapping of blocks to virtual disks on an (often) remote storage backend. The backend could be a distributed system of its own with a metadata service mapping files or chunks to file systems to physical drives.

File system semantics. Some file system functionality (such as byte-range file locking when multiple clients access the same file) depends on consulting file system state to determine the success and semantics of individual IO operations. The logic and state that dictates the semantics of these operations resides inside the file system, at the destination endpoint of these IOs. IO routing needs to maintain the same file system functionality and semantics in the storage stack.

Efficiency. Providing IO stack customization requires a different way of building specialized functionality. We move away from an architecture that hard-codes functionality on the IO stack to an architecture that dynamically directs IOs to specialized stages. Any performance overheads incurred must be minimal.

3 Design

Figure 2 shows sRoute's architecture. It is composed of **sSwitches** on the data plane, that change the route of IOs according to forwarding rules. sSwitches are programmable through a simple API with four calls shown in Table 2. The sSwitches forward IOs to other file destinations, the controller, or to specialized stages (e.g., one that implements a particular caching algorithm). **A control plane** with a logically-centralized controller specifies the location of the sSwitches and inserts forwarding rules in them. Specialized **stages** take an IO as an input, perform operations on its payload and return the IO back to the sSwitch for further forwarding.

3.1 Baseline architecture

The baseline system architecture our design builds on is that of an enterprise data center. Each tenant is allocated VMs or containers[1] and runs arbitrary applications or services in them. Network and storage are virtualized and VMs are unaware of their topology and properties.

The baseline system is assumed to already have separate control and data planes and builds on the IOFlow architecture [53]. That architecture provides support for flow-based classification and queuing and communication of basic per-flow statistics to a controller.

3.2 Design goals

sRoute's design targets several goals. First, we want a solution that does not involve application or VM changes. Applications have limited visibility of the data center's IO stack. This paper takes the view that data center services are better positioned for IO stack customization. These are then exposed to applications through new

[1] This paper's implementation uses VMs.

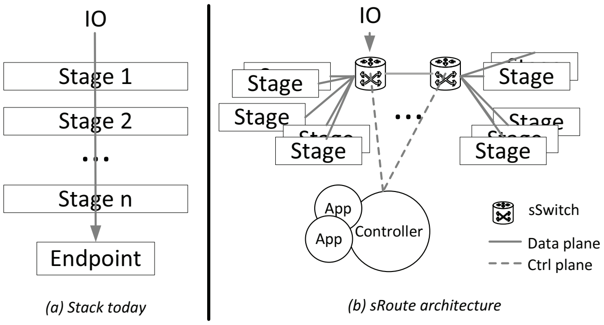


Figure 2: System architecture. sSwitches can route IO within a physical machine's IO stack and across machines over the network.

`Insert` (IOHeader, Delegate)	Creates a new fwd. rule matching the IO header, using dynamic control delegate to look up destination
`Delete` (IOHeader)	Deletes all rules matching the header
`Quiesce` (IOHeader, Boolean)	Blocks or unblocks incoming IO matching IO header when Boolean is true or false respectively
`Drain` (IOHeader)	Drains all pending IOs matching the IO header

Table 2: Control API to the sSwitch.

Rule	:=	$IOHeader \rightarrow Delegate(IOHeader)$
IOHeader	:=	$< Source, Operation, File >$
Delegate	:=	$F(IOHeader); return\{Detour\}$
Source	:=	Unique Security Identifier
Operation	:=	`read`|`write`|`create`|`delete`
File	:=	$< FileName, Offset, Length >$
Detour	:=	$< IO \| IOHeader, DetourLoc >$
DetourLoc	:=	$File \| Stage \| Controller$
Stage	:=	$< HostName, DriverName >$
F	:=	Restricted code

Figure 3: Construct definitions.

types of service level agreements (SLA), e.g., guaranteeing better throughput and latency. Second, data-plane performance overheads should be minimal. Third, the control plane should be flexible and allow for a diverse set of application policies.

The rest of this section focuses on the sSwitches and the control plane interfaces to them. Section 5 focuses on control applications. Figure 3 provides the construct definitions used in the rest of the paper.

3.3 sSwitches on the data plane

An sSwitch is a special stage that is inserted into the IO stack (data plane) to provide IO routing. An sSwitch forwards IO according to rules specified by the control plane. A forwarding rule contains two parts: an IO header and an action or delegate function[2]. IO packets are matched against the IO header, and the associated delegate in the *first* successful rule match executes (hence, the order of installed rules matters). In the simplest form, this delegate returns a set of stages where the IO should next be directed. For example, routing all traffic from VM_1 for file X on server S_1 to file Y on server S_2 can be represented with this rule:

1: $< VM_1, *, //S_1/X > \rightarrow (return < IO, //S_2/Y >)$

An sSwitch implements four control plane API calls as shown in Table 2. The APIs allow the control plane to `Insert` a forwarding rule, or `Delete` it. Rules can be changed dynamically by two entities on the control plane: the controller, or a control `Delegate` function.

As defined in Figure 3, the IO header is a tuple containing the source of an IO, the operation, and the file affected. The source of an IO can be a process or a VM uniquely authenticated through a security identifier. The destination is a file in a (possibly remote) share or directory. Building on IOFlow's classification mechanism [53] allows an sSwitch to have visibility over all the above and other relevant IO header entries at any point in the IO stack (without IOFlow, certain header entries such as the source, could be lost or overwritten as IO flows through the system).

The operation can be one of `read`, `write`, `create` or `delete`. Wildcards and longest prefix matching can be used to find a match on the IO header. A default match rule sends an IO to its original destination. A detour location could be a file (e.g., another file on a different server from the original IO's destination), a stage on the path to the endpoint (example rule 1 below), or the centralized controller (example rule 2 below that sends the IO header for all writes from VM_2 to the controller):

1: $< VM_1, *, //S_1/X > \rightarrow (return < IO, //S_2/C >)$
2: $< VM_2, w, * > \rightarrow (return < IOHeader, Controller >)$

The sSwitch is responsible for transmitting the full IO or just its header to a set of stages. The response does not have to flow through the same path as the request, as

[2] The reason the second part of the rule is a function (as opposed to simply a set of routing locations) is for control plane efficiency in some situations, as is explained further in this section.

long as it reaches the initiating source[3].

Unlike in networking, the sSwitch needs to perform more work than just forwarding. It also needs to prepare the endpoint stages to accept IO, which is unique to storage. When a rule is first installed, the sSwitch needs to open a file connection to the endpoint stages, in anticipation of IO arriving. The sSwitch needs to create it and take care of any namespace conflicts with existing files (§4). `Open` and `create` operations are expensive synchronous metadata operations. There is an inherent tradeoff between lazy file creation upon the first IO arriving and file creation upon rule installation. The former avoids unnecessarily creating files for rules that do not have any IO matching them, but upon a match the first IO incurs a large latency. The latter avoids the latency but could create several empty files. The exact tradeoff penalties depend on the file systems used. By default this paper implements the latter, but ideally this decision would also be programmable (but it is not so yet.)

sSwitches implement two additional control plane APIs. A `Quiesce` call is used to block any further requests with the same IO header from propagating further. The implementation of this call builds on the lower-level IOFlow API that sets the token rate on a queue [53]. `Drain` is called on open file handles to drain any pending IO requests downstream. Both calls are synchronous. These calls are needed to change the path of IOs in a consistent manner, as discussed in the next section.

3.4 Controller and control plane

A logically centralized controller has global visibility over the stage topology of the data center. This topology comprises of all physical servers, network and storage components as well as the software stages within a server. Maintaining this topology in a fault-tolerant manner is already feasible today [24].

The controller is responsible for three tasks. First, it takes a high level tenant policy and translates it into sSwitch API calls. Second, it decides *where* to insert the sSwitches and specialized stages in the IO stack to implement the policy. Third, it disseminates the forwarding rules to the sSwitches. We show these tasks step-by-step for two simple control applications below.

The first control application directs a tenant's IO to a **specialized file cache**. This policy is part of a case study detailed in Section 5.3. The tenant is distributed over 10 VMs on 10 different hypervisors and accesses a read-only dataset X. The controller forwards IO from this set of VMs to a specialized cache C residing on a remote machine connected to the hypervisors through a fast RDMA network. The controller knows the topology of the data paths from each VM to C and injects sSwitches at each hypervisor. It then programs each sSwitch as follows:

```
1: for $i \leftarrow 1, 10$ do
2:     Quiesce (< $VM_i$, *, //$S_1$/X>, true)
3:     Drain (< $VM_i$, *, //$S_1$/X>)
4:     Insert (< $VM_i$, *, //$S_1$/X>, (return <IO, //server $S_2$/C>))
5:     Quiesce (< $VM_i$, *, //$S_1$/X>, false)
```

The first two lines are needed to complete any IOs in-flight. This is done so that the sSwitch does not need to keep any extra metadata to know which IOs are on the old path. That metadata would be needed, for example, to route a newly arriving `read` request to the old path since a previous `write` request might have been buffered in an old cache on that path. The delegate in line 4 simply returns the cache stage. The controller also injects an sSwitch at server S_2 where the specialized cache resides, so that any requests that miss in cache are sent further to the file system of server S_1. The rule at S_2 matches IOs from C for file X and forwards them to server S_1:

```
Insert (< $C$, *, //$S_1$/X>, (return <IO, //$S_1$/X>))
```

The second control application improves a tenant's **tail latency** and illustrates a more complex control delegate. The policy states that queue sizes across servers should be balanced. This policy is part of a case study detailed in Section 5.1. When a load burst arrives at a server S_1 from a source VM_1 the control application decides to temporarily forward that load to a less busy server S_2. The controller can choose to insert an sSwitch in the VM_1's hypervisor or at the storage server S_1. The latter means that IOs go to S_1 as before and S_1 forwards them to S_2. To avoid this extra network hop the controller chooses the former. It then calls the following functions to insert rules in the sSwitch:

```
Insert (<$VM_1$, w, //$S_1$/X>, (F(); return <IO, //$S_2$/X>))
Insert (<$VM_1$, r, //$S_1$/X>, (return <IO, //$S_1$/X>))
```

The rules specify that `writes` "w" are forwarded to the new server, whereas `reads` "r" are still forwarded to the old server. This application demands that `reads` return the latest version of the data. When subsequently a `write` for the first 512KB of data arrives[4], the delegate function updates the read rule through function $F()$ whose body is shown below:

```
Delete (<$VM_1$, r, //$S_1$/X>)
Insert (<$VM_1$, r, //$S_1$/X, $0, 512KB$ >, (return <IO, //$S_2$/X>))
Insert (<$VM_1$, r, //$S_1$/X>, (return <IO, //$S_1$/X>)
```

Note that quiescing and draining are not needed in this

[3] sSwitches cannot direct IO responses to sources that did not initiate the IO. Finding scenarios that need such source routing and the mechanism for doing so is future work.

[4] The request's start offset and data length are part of the IO header.

scenario since the sSwitch is keeping the metadata necessary (in the form of new rules) to route a request correctly. A subsequent `read` for a range between 0 and 512KB will match the rule in line 2 and will be sent to S_2. Note that sSwitch matches on byte ranges as well, so a `read` for a range between 0 and 1024KB will be now split into two reads. The sSwitch maintains enough buffer space to coalesce the responses.

3.4.1 Delegates

The above examples showed instances of control delegates. Control delegates are restricted control plane functions that are installed at sSwitches for control plane efficiency. In the second example above, the path of an IO depends on the workload. `Write` requests can potentially change the location of a subsequent `read`. One way to handle this would be for all requests to be sent by the sSwitch to the controller using the following alternate rules and delegate function:

```
Insert (<$VM_1$, w, //$S_1$/X>, (return <IO, Controller>))
Insert (<$VM_1$, r, //$S_1$/X>, (return <IO, Controller>))
```

The controller would then serialize and forward them to the appropriate destination. Clearly, this is inefficient, bottlenecking the IO stack at the controller. Instead, the controller uses restricted delegate functions that make control decisions locally at the sSwitches.

This paper assumes a non-malicious controller, however the design imposes certain functionality restrictions on the delegates to help guard against accidental errors. In particular, delegate functions may only call the APIs in Table 2 and may not otherwise keep or create any other state. They may insert or delete rules, but may not rewrite the IO header or IO data. That is important since the IO header contains entries such as source security descriptor that are needed for file access control to work in the rest of the system. These restrictions allow us to consider the delegates as a natural extension of the centralized controller. Simple programming language checks and passing the IO as read-only to the delegate enforce these restrictions. As part of future work we intend to explore stronger verification of control plane correctness properties, much like similar efforts in networking [27].

3.5 Consistent rule updates

Forwarding rule updates could lead to instability in the system. This section introduces the notion of consistent rule updates. These updates preserve well-defined storage-specific properties. Similar to networking [45] storage has two different consistency requirements: per-IO and per-flow.

Per-IO consistency. Per-IO consistent updates require that each IO flows either through an old set of rules or an updated set of rules, but not through a stack that is composed of old and new paths. The `Quiesce` and `Drain` calls in the API in Table 2 are sufficient to provide per-IO consistent updates.

Per-flow consistency. Many application require a stream of IOs to behave consistently. For example, an application might require that a `read` request obtains the data from the latest previous `write` request. In cases where the same source sends both requests, then per-IO consistency also provides per-flow consistency. However, the second request can arrive from a different source, like a second VM in the distributed system. In several basic scenarios, it is sufficient for the centralized controller to serialize forwarding rule updates. The controller disseminates the rules to all sSwitches in two phases. In the first phase, the controller quiesces and drains requests going to the old paths and, in the second phase, the controller updates the forwarding rules.

However, a key challenge are scenarios where delegate functions create new rules. This complicates update consistency since serializing these new rules through the controller is inefficient when rules are created frequently (e.g., for every `write` request). In these cases, control applications attempt to provide all serialization through the sSwitches themselves. They do so as follows. First, they consult the topology map to identify points of serialization along the IO path. The topology map identifies common stages among multiple IO sources on their IO stack. For example, if two clients are reading and writing to the same file X, the control application has the option of inserting two sSwitches with delegate functions close to the two sources to direct both clients' IOs to Y. This option is shown in Figure 4(a). The sSwitches would then need to use two-phase commit between themselves to keep rules in sync, as shown in the Figure. This localizes updates to participating sSwitches, thus avoiding the need for the controller to get involved.

A second option would be to insert a single sSwitch close to X (e.g., at the storage server) that forwards IO to Y. This option is shown in Figure 4(b). A third option would be to insert an sSwitch at Y that forwards IO back to X if the latest data is not on Y. This type of forwarding rule can be thought of as implementing *backpointers*. Note that two additional sSwitches are needed close to the source to forward all traffic, i.e., reads and writes, to Y, however these sSwitches do not need to perform two-phase commit. The choice between the last two options

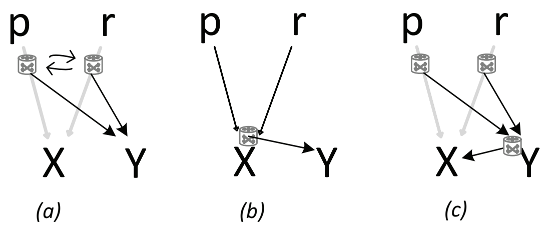


Figure 4: Three possible options for placing sSwitches for consistent rule updates. Either can be chosen programmatically at runtime.

depends on the workload. If the control application expects that most IO will go to the new file the third option would eliminate an extra network hop.

3.6 Fault tolerance and availability

This section analyzes new potential risks on fault tolerance and availability induced by our system. Data continues to be N-way replicated for fault tolerance and its fault tolerance is the same as in the original system.

First, the controller service is new in our architecture. The service can be replicated for availability using standard Paxos-like techniques [31]. If the controller is temporarily unavailable, the implication on the rest of the system is at worst slower performance, but correctness is not affected. For example, IO that matches rules that require transmission to the controller will be blocked until the controller recovers.

Second, our design introduces new metadata in the form of forwarding rules at sSwitches. It is a design goal to maintain all state at sSwitches as soft-state to simplify recovery — also there are cases where sSwitches do not have any local storage available to persist data. The controller itself persists all the forwarding rules before installing them at sSwitches. The controller can choose to replicate the forwarding rules, e.g., using 3-way replication (using storage space available to the controller — either locally or remotely).

However, forwarding rules created at the control delegates pose a challenge because they need persisting. sRoute has two options to address this challenge. The first is for the controller to receive all delegate updates synchronously, ensure they are persisted and then return control to the delegate function. This option involves the controller on the critical path. The second option (the default) is for the delegate rules to be stored with the forwarded IO data. A small header is prepended to each IO containing the updated rule. On sSwitch failure, the controller knows which servers IO has been forwarded to and recovers all persisted forwarding rules from them.

Third, sSwitches introduce new code along the IO stack, thus increasing its complexity. When sSwitches are implemented in the kernel (see Section 4), an sSwitch failure may cause the entire server to fail. We have kept the code footprint of sSwitches small and we plan to investigate software verification techniques in the future to guard against such failures.

3.7 Design limitations

In the course of working with sRoute we have identified several design limitations. First, sRoute currently lacks any verification tools that could help programmers. For example, it is possible to write incorrect control applications that route IOs to arbitrary locations, resulting in data loss. Thus, the routing flexibility is powerful, but unchecked. There are well-known approaches in networking, such as header space analysis [27], that we believe could also apply to storage, but we have not investigated them yet.

Second, we now have experience with SDN controllers and SDS controllers like the one in this paper. It would be desirable to have a control plane that understands both the network and storage. For example, it is currently possible to get into inconsistent end-to-end policies when the storage controller decides to send data from server A to B while the network controller decides to block any data from A going to B. Unifying the control plane across resources is an important area for future work.

4 Implementation

An sSwitch is implemented partly in kernel-level and partly in user-level. The kernel part is written in C and its functionality is limited to partial IO classification through longest prefix matching and forwarding within the same server. The user-level part implements further sub-file-range classification using hash tables. It also implements forwarding IO to remote servers. An sSwitch is a total of 25 kLOC.

Routing within a server's IO stack. Our implementation makes use of the filter driver architecture in Windows [39]. Each filter driver implements a stage in the kernel and is uniquely identified using an altitude ID in the IO stack. The kernel part of the sSwitch automatically attaches control code to the beginning of each filter driver processing. Bypassing a stage is done by simply returning from the driver early. Going through a stage means going through all the driver code.

Routing across remote servers. To route an IO to an arbitrary remote server's stage, the kernel part of the sSwitch first performs an upcall sending the IO to the user-level part of the sSwitch. That part then transmits the IO to a remote detour location using TCP or RDMA (default) through the SMB file system protocol. On the remote server, an sSwitch intercepts the arriving packet and routes it to a stage within that server.

sSwitch and stage identifiers. An sSwitch is a stage and has the same type of identifier. A stage is identified by a server host name and a driver name. The driver name is a tuple of <device driver name, device name, altitude>. The altitude is an index into the set of drivers or user-level stages attached to a device.

Other implementation details. For the case studies in this paper, it has been sufficient to inject one sSwitch inside the Hyper-V hypervisor in Windows and another on the IO stack of a remote storage server just above the NTFS file system using file system filter drivers [39]. Specialized functionality is implemented entirely in user-level stages in C#. For example, we have implemented a user-level cache (Section 5.3). The controller is also implemented in user-level and communicates with both kernel- and user-level stages through RPCs over TCP.

Routing happens on a per-file basis, at block granularity. Our use cases do not employ any semantic information about the data stored in each block. For control applications that require such information, the functionality would be straightforward to implement, using mini-port drivers, instead of filter drivers.

Applications and VMs always run unmodified on our system. However, some applications pass several static hints such as "write through" to the OS using hard-coded flags. The sSwitches intercept `open/create` calls and can change these flags. In particular, for specialized caching (Section 5.3) the sSwitches disable OS caching by specifying `Write-through` and `No-buffering` flags. Caching is then implemented through the control application. To avoid namespace conflict with existing files, sRoute stores files in a reserved "sroute-folder" directory on each server. That directory is exposed to the cluster as an SMB share writable by internal processes only.

Implementation limitations. A current limitation of the implementation is that sSwitches cannot intercept individual IO to memory mapped files. However, they can intercept bulk IO that loads a file to memory and writes pages to disk, which is sufficient for most scenarios.

Another current limitation of our implementation is that it does not support byte-range file locking for multiple clients accessing the same file, while performing endpoint routing. The state to support this functionality is kept in the file system, at the original endpoint of the flow. When the endpoint is changed, this state is unavailable. To support this functionality, the sSwitches can intercept `lock`/`unlock` calls and maintain the necessary state, however this is not currently implemented.

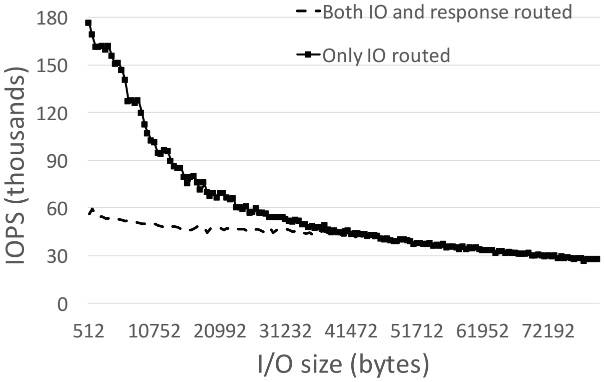


Figure 5: Current performance range of an sSwitch.

The performance range of the current implementation of an sSwitch is illustrated in Figure 5. This throughput includes passing an IO through both kernel and user-level. Two scenarios are shown. In the "Only IO routed" scenario, each IO has a routing rule but an IO's response is not intercepted by the sSwitch (the response goes straight to the source). In the "Both IO and response routed" scenario both an IO and its response are intercepted by the sSwitch. Intercepting responses is important when the response needs to be routed to a non-default source as well (one of our case studies for caches in Section 5.3 requires response routing). Intercepting an IO's response in Windows is costly (due to interrupt handling logic beyond the scope of this paper) and the performance difference is a result of the OS, not of the sSwitch. Thus the performance range for small IO is between 50,000-180,000 IOPS which makes sSwitches appropriate for an IO stack that uses disk or SSD backends, but not yet a memory-based stack.

5 Control applications

This section makes three points. First, we show that a diverse set of control applications can be built on top of IO routing. Thus, we show that the programmable routing abstraction can replace one-off hardcoded implementations. We have built and evaluated three control applications implementing tail latency control, replica set control and file cache control. These applications cover each of the detouring types in Table 1. Second, we show that tenants benefit significantly from the IO customization

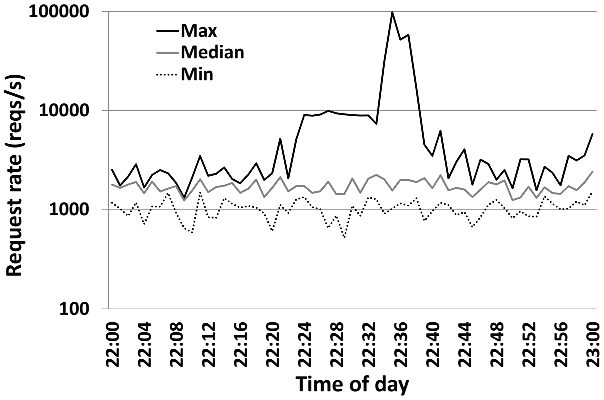


Figure 6: Load on three Exchange server volumes showing load imbalances.

provided by the control applications. Third, we evaluate data and control plane performance.

Testbed. The experiments are run on a testbed with 12 servers, each with 16 Intel Xeon 2.4 GHz cores, 384 GB of RAM and Seagate Constellation 2 disks. The servers run Windows Server 2012 R2 operating system and can act as either Hyper-V hypervisors or as storage servers. Each server has a 40 Gbps Mellanox ConnectX-3 NIC supporting RDMA and connected to a Mellanox MSX1036B-1SFR switch.

Workloads. We use three different workloads in this section. The first is TPC-E [55] running over unmodified SQL Server 2012 R2 databases. TPC-E is a transaction processing OLTP workload with small IO sizes. The second workload is a public IO trace from an enterprise Exchange email server [49]. The third workload is IoMeter [23], which we use for controlled micro-benchmarks.

5.1 Tail latency control

Tail latency in data centers can be orders of magnitude higher than average latency leading to application unresponsiveness [14]. One of the reasons for high tail latency is that IOs often arrive in bursts. Figure 6 illustrates this behavior in publicly available Exchange server traces [49], showing traffic to three different volumes of the Exchange trace. The difference in load between the most loaded volume and the least loaded volume is two orders of magnitude and lasts for more than 15 minutes.

Data center providers have load balancing solutions for CPU and network traffic [19]. IO to storage on the other hand is difficult to load balance at short timescales because it is stateful. An IO to an overloaded server S must go to S since it changes state there. The first control application addresses the tail latency problem by temporarily forwarding IOs from loaded servers onto less loaded ones while ensuring that a `read` always accesses the last acknowledged update. This is a type of *endpoint routing*. The functionality provided is similar to Everest [41] but written as a control application that decides when and where to forward to based on global system visibility.

The control application attempts to balance queue sizes at each of the storage servers. To do so, for each storage server, the controller maintains two running averages based on stats it receives[5]: Req_{Avg}, and Req_{Rec}. Req_{Avg} is an exponential moving average over the last hour. Req_{Rec} is an average over a sliding window of one minute, meant to capture the workload's recent request rate. The controller then temporarily forwards IO if:

$$Req_{Rec} \quad > \quad \alpha Req_{Avg}$$

where α represents the relative increase in request rate that triggers the forwarding. We evaluate the impact of this control application on the Exchange server traces shown in Figure 6, but first we show how we map this scenario into forwarding rules.

There are three flows in this experiment. Three different VMs VM_{max}, VM_{min} and VM_{med} on different hypervisors access one of the three volumes in the trace "Max", "Min" and "Median". Each volume is mapped to a VHD file VHD_{max}, VHD_{min} and VHD_{med} residing on three different servers S_{max}, S_{min} and S_{med} respectively. When the controller detects imbalanced load, it forwards write IOs from the VM accessing S_{max} to a temporary file T on server S_{min}:

```
$< *, w, //S_{max}/VHD_{max} > \rightarrow (F(); return < IO, //S_{min}/T >)$
$< *, r, //S_{max}/VHD_{max} > \rightarrow (return < IO, //S_{max}/VHD_{max} >)$
```

Read IOs follow the path to the most up-to-date data, whose location is updated by the delegate function $F()$ as the write IOs flow through the system. We showed how $F()$ updates the rules in Section 3.4. Thus, the forwarding rules always point a read to the latest version of the data. If no writes have happened yet, all reads by definition go to the old server VM_{max}. The control application may also place a specialized stage O in the new path that implements an optional log-structured layout that converts all writes to streaming writes by writing them sequentially to S_{min}. The layout is optional since SSDs already implement it internally and it is most useful for disk-based backends[6]. The control application in-

[5] The controller uses IOFlow's `getQueueStats` API [53] to gather system-wide statistics for all control applications.

[6] We have also implemented a 1:1 layout that uses sparse files, but do not describe it here due to space restrictions.

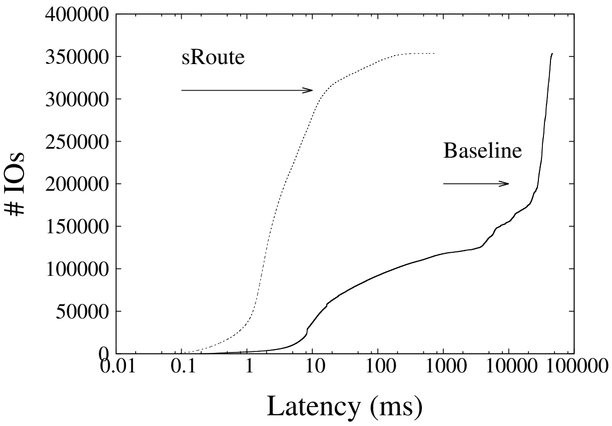


Figure 7: CDF of response time for baseline system and with IO routing.

serts a rule forwarding IO from the VM first to O (rule 1 below), and another to route from O to S_{min} (rule 2).

1: $< *,*, //S_{max}/VHD_{max} > \rightarrow (return < IO, //S_{min}/O >)$
2: $< O,*, //S_{max}/VHD_{max} > \rightarrow (return < IO, //S_{min}/T)$

Note that in this example data is partitioned across VMs and no VMs share data. Hence, the delegate function in the sSwitch is the only necessary point of metadata serialization in system. This is a simple version of case (a) in Figure 4 where sSwitches do not need two-phase commit. The delegate metadata is temporary. When the controller detects that a load spike has ended, it triggers data *reclaim*. All sSwitch rules for writes are changed to point to the original file VHD_{max}. Note that read rules still point to T until new arriving writes overwrite those rules to point to VHD_{max} through their delegate functions. The controller can optionally speed up the reclaim process by actively copying forwarded data to its original location. When the reclaim process ends, all rules can be deleted, the sSwitches and specialized stage removed from the IO stack, since all data resides in and can be accessed again from the original server S_{max}.

We experiment by replaying the Exchange traces using a time-accurate trace replayer on the disk-based testbed. We replay a 30 minute segment of the trace, capturing the peak interval and allowing for all forwarded data to be reclaimed. Figure 7 shows the results. IO routing results in two orders of magnitude improvements in tail latency for the flow to S_{max}. The change latency distribution for S_{min} (not shown) is negligible.

Overheads. 2.8GB of data was forwarded and the delegate functions persisted approximately 100,000 new control plane rules with no noticeable overhead. We experimentally triggered one sSwitch failure, and measured that it took approximately 30 seconds to recover the rules from the storage server. The performance benefit obtained is similar to specialized implementations [41]. The CPU overhead at the controller was less than 1%.

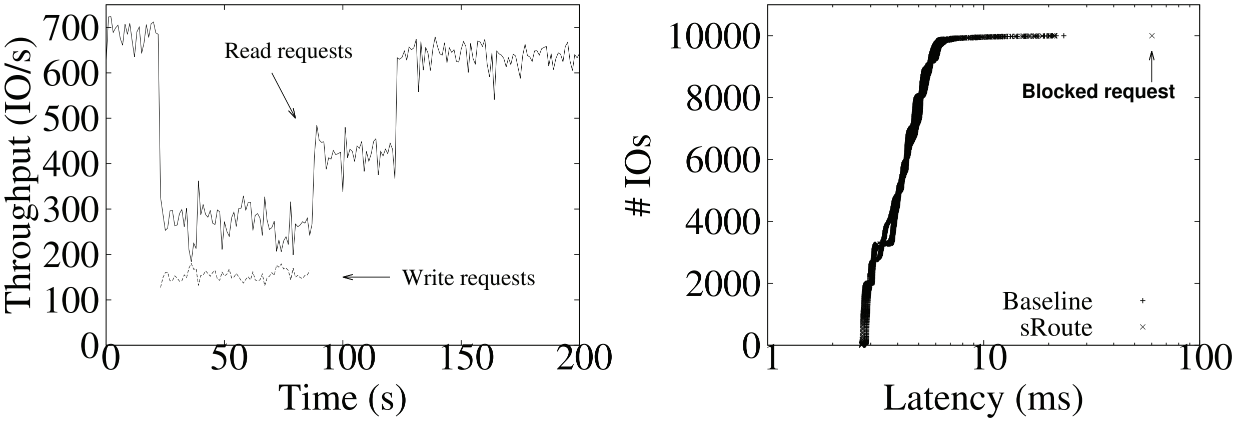
5.2 Replica set control

No one replication protocol fits all workloads [1,33,51]. Data center services tend to implement one particular choice (e.g, primary-based serialization) and offer it to all workloads passing through the stack (e.g., [7]). One particularly important decision that such an implementation hard-codes is the choice of write-set and read-set for a workload. The write-set specifies the number of servers to contact for a `write` request. The size of the write-set has implications on request latency (a larger set usually means larger latency). The read-set specifies the number of servers to contact for `read` requests. A larger read-set usually leads to higher throughput since multiple servers are read in parallel.

The write- and read-sets need to intersect in certain ways to guarantee a chosen level of consistency. For example, in primary-secondary replication, the intersection of the write- and read-sets contains just the primary server. The primary then writes the data to a write-set containing the secondaries. The request is completed once a subset of the write-set has acknowledged it (the entire write-set by default).

The replica set control application provides a configurable write- and read-set. It uses only *scatter routing* to do so, without any specialized stages. In the next experiment the policy at the control application specifies that if the workload is read-only, then the read-set should be all replicas. However, for correct serialization, if the workload contains writes, all requests must be serialized through the primary, i.e., the read-set should be just the primary replica. In this experiment, the application consists of 10 IoMeters on 10 different hypervisors reading and writing to a 16GB file using 2-way primary-based replication on the disk testbed. IoMeter uses 4KB random-access requests and each IoMeter maintains 4 requests outstanding (MPL).

The control application monitors the read:write ratio of the workload through IOFlow and when it detects it has been read-only for more than 30 seconds (a configurable parameter) it switches the read-set to be all replicas. To do that, it injects sSwitches at each hypervisor and sets up rules to forward reads to a randomly chosen server S_{rand}. This is done through a control delegate that picks the next server at random. To make the switch between old and new rule the controller firsts `quiesces`



(a) Read and write throughput

(b) Write latency CDF

Figure 8: Reads benefit from parallelism during read-only phases and the system performs correct serialization during read:write phases (a). The first write needs to block until forwarding rules are changed (b).

writes, then drains them. It then inserts the new read-set rule (rule 1):

1: $< *, r, \ //S_1/X > \rightarrow (F(); return < IO, \ //S_{rand}/X >)$
2: $< *, w, * > \rightarrow (return < IOHeader, \ Controller >)$

The controller is notified of the arrival of any write requests by the rule (2). The controller then proceeds to revert the read-set rule, and restarts the write stream.

Figure 8 shows the results. The performance starts high since the workload is in a read-only state. When the first write arrives at time 25, the controller switches the read-set to contain just the primary. In the third phase starting at time 90, writes complete and read performance improves since reads do not contend with writes. In the fourth phase at time 125, the controller switches the read-set to be both replicas, improving read performance by 63% as seen in Figure 8(a). The tradeoff is that the first write requests that arrive incurr a latency overhead from being temporarily blocked while the write is signalled to the controller, as shown in Figure 8(b). Depending on the application performance needs, this latency overhead can be amortized appropriately by increasing the time interval before assuming the workload is read-only. The best-case performance improvement expected is 2x, but the application (IoMeter) has a low MPL and does not saturate storage in this example.

Overheads. The control application changes the forwarding rules infrequently at most every 30 seconds. In an unoptimized implementation, a rule change translated to 418Bytes/flow for updates (40MB for 100,000 flows). The control application received stats every second using 302Bytes/flow for statistics (29MB/s for 100,000 flows). The CPU overhead at the controller is negligible.

5.3 File cache control

File caches are important for performance: access to data in the cache is more than 3 orders of magnitude faster than to disks. A well-known problem is that data center tenants today have no control over the location of these caches or their policies [2, 8, 16, 50]. The only abstraction the data center provides to a tenant today is a VMs's memory size. This is inadequate in capturing all the places in the IO stack where memory could be allocated. VMs are inadequate even in providing isolation: an aggressive application within a VM can destroy the cache locality of another application within that VM.

Previous work [50] has explored the programmability of caches on the IO stack, and showed that applications and cloud providers can greatly benefit from the ability to customize cache size, eviction and write policies, as well as explicitly control the placement of data in caches along the IO stack. Such explicit control can be achieved by using filter rules [50] installed in a cache. All incoming IO headers are matched against installed filter rules, and an IO is cached if its header matches an installed rule. However, this type of simple control only allows IOs to be cached at some point along their *fixed* path from the application to the storage server. The ability to route IOs to arbitrary locations in the system using sSwitches while maintaining desired consistency semantics allows *disaggregation* of cache memory from the rest of a workload's allocated resources.

This next file cache control application provides several IO stack customizations through *waypoint routing*. We focus on one here: cache isolation among tenants. Cache isolation in this context means that a) the con-

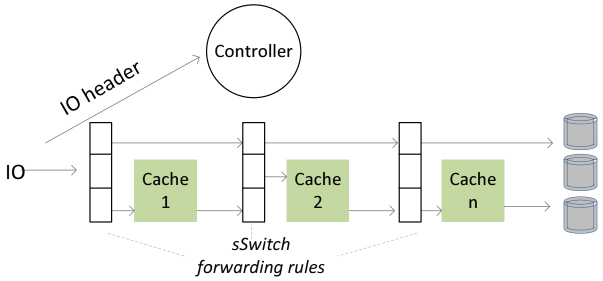


Figure 9: Controller sets path of an IO through multiple cache using forwarding rules in sSwitches.

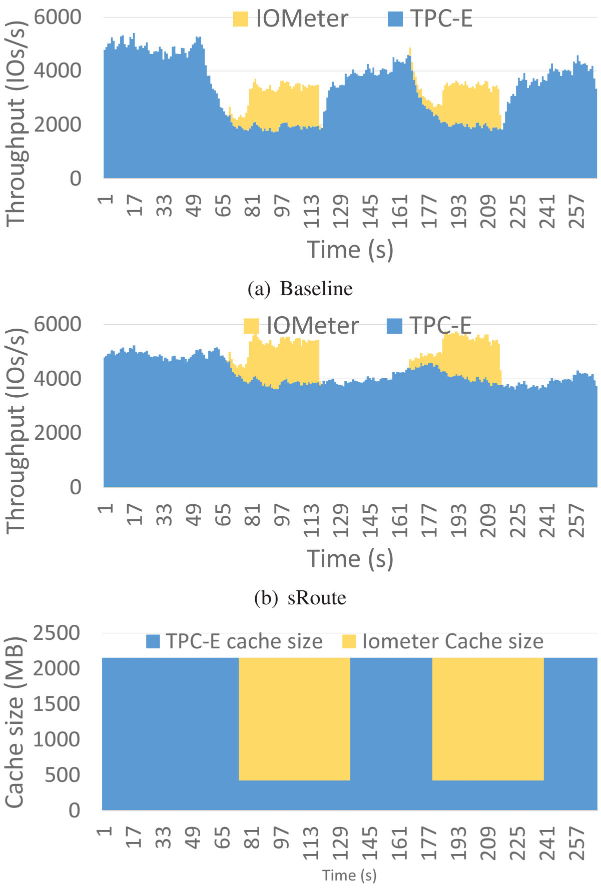
troller determines how much cache each tenant needs and b) the sSwitches isolate one tenant's cache from another's. sRoute controls the path of an IO. It can forward an IO to a particular cache on the data plane. It can also forward an IO to bypass a cache as shown in Figure 9.

The experiment uses two workloads, TPC-E and IoMeter, competing for a storage server's cache. The storage backend is disks. The TPC-E workload represents queries from an SQL Server database with a footprint of 10GB running within a VM. IoMeter is a random-access read workload with IO sizes of 512KB. sRoute's policy in this example is to maximize the utilization of the cache with the hit rate measured in terms of IOPS. In the first step, all IO headers are sent to the controller which computes their miss ratio curves using a technique similar to SHARDS [56].

Then, the controller sets up sSwitches so that the IO from IOMeter and from TPC-E go to different caches $C_{IOMeter}$ and C_{TPCE} with sizes provided by SHARDS respectively (the caches reside at the storage server):

1: $< IOMeter,*,* >,(return < IO,C_{IOMeter} >)$
2: $< TPCE,*,* >,(return < IO,C_{TPCE} >)$

Figure 10 shows the performance of TPC-E when competing with two bursts of activity from the IoMeter workload, with and without sRoute. When sRoute is enabled (Figure 10(b)), total throughput increases when both workloads run. In contrast, with today's caching (Figure 10(a)) total throughput actually drops. This is because IoMeter takes enough cache away from TPC-E to displace its working set out of the cache. With sRoute, total throughput improves by 57% when both workloads run, and TPC-E's performance improves by 2x.



(a) Baseline

(b) sRoute

(c) Estimated cache sizes

Figure 10: Maximizing hit rate for two tenants with different cache miss curves.

Figure 10(c) shows the cache allocations output by our control algorithm when sRoute is enabled. Whenever IoMeter runs, the controller gives it 3/4 of the cache, whereas TPC-E receives 1/4 of the cache, based on their predicted miss ratio curves. This cache allocation leads to each receiving around 40% cache hit ratio. Indeed, the allocation follows the miss ratio curve that denotes what the working set of the TPC-E workload is – after this point diminishing returns can be achieved by providing more cache to this workload. Notice that the controller apportions unused cache to the TPC-E workload 15 seconds after the IoMeter workload goes idle.

Overheads. The control application inserted forwarding rules at the storage server. Rule changes were infrequent (the most frequent was every 30 seconds). The control plane uses approximately 178Bytes/flow for rule updates (17MB for 100,000 flows). The control plane subsequently collects statistics from sSwitches and cache stages every control interval (default is 1 second). The statistics are around 456Bytes/flow (roughly 43MB for 100,000 flows). We believe these are reasonable control plane overheads. When SHARD ran it consumed 100% of two cores at the controller.

6 Open questions

Our initial investigation on treating the storage stack like a network provided useful insights into the pros and cons of our approach. We briefly enumerate several open questions that could make for interesting future work. In Section 3.7 we already discussed two promising areas of future work: 1) sRoute currently lacks verification tools that could help programmers and 2) it would be interesting to merge a typical SDN controller with our storage controller into one global controller.

Related to the first area above, more experience is needed, for example, to show whether sRoute rules from multiple control applications can co-exist in the same system safely. Another interesting area of exploration relates to handling policies for storage data at rest. Currently sRoute operates on IO as it is flowing through the system. Once the IO reaches the destination it is considered at rest. It might be advantageous for an sSwitch to initiate itself data movement for such data at rest. That would require new forwarding rule types and make an sSwitch more powerful.

7 Related work

Our work is most related to software-defined networks (SDNs) [9,13,17,25,28,44,54,58] and software-defined storage (SDS) [2, 53]. Specifically, our work builds directly upon the control-data decoupling enabled by IOFlow [53], and borrows two specific primitives: classification and rate limiting based on IO headers for quiescing. IOFlow also made a case for request routing. However, it only explored the concept for bypassing stages along the path, and did not consider the full IO routing spectrum where the path and endpoint can also change, leading to consistency concerns. This paper provides the full routing abstraction.

There has been much work in providing applications with specialized use of system resources [2,4,6,16,26]. The Exokernel architecture [16, 26] provides applications direct control over resources with minimal kernel involvement. SPIN [6] and Vino [47] allow applications to download code into the kernel, and specialize resource management for their needs. Another orthogonal approach is to extend existing OS interfaces and pass hints vertically along the IO stack [2–4,36]. Hints can be passed in both directions between the application and the system, exposing application needs and system resource capabilities to provide a measure of specialization.

In contrast to the above approaches, this paper makes the observation that modern IO stacks support mechanisms for injecting stages with specialized functionality (e.g., in Windows [38], FreeBSD [18] and Linux [34]). sRoute transforms the problem of providing application flexibility into an IO routing problem. sRoute provides a control plane to customize an IO stack by forwarding a tenants' IO to the right stages without changing the application or requiring a different OS structure.

We built three control applications on top of IO routing. The functionality provided from each has been extensively studied in isolation. For example, application-specific file cache management has shown significant performance benefits [8, 20, 22, 29, 32, 50, 57]. Snapshots, copy-on-write and file versioning all have at their core IO routing. Hard-coded implementations can be found in file systems like ZFS [42], WAFL [21] and btrfs [46]. Similarly, Narayanan et al. describe an implementation of load balancing through IO offloading of write requests [40, 41]. Abd-el-malek et al. describe a system implementation where data can be re-encoded and placed on different servers [1]. Finally, several distributed storage systems each offer different consistency guarantees [5,7,10–12,15,30,33,51,52] .

In contrast to these specialized implementations, sRoute offers a programmable IO routing abstraction that allows for all this functionality to be specified and customized at runtime.

8 Conclusion

This paper presents sRoute, an architecture that enables an IO routing abstraction, and makes the case that it is useful. We show that many specialized functions on the storage stack can be recast as routing problems. Our hypothesis when we started this work was that, because routing is inherently programmable and dynamic, we could substitute hard-coded one-off implementations with one common routing core. This paper shows how sRoute can provide unmodified applications with specialized tail latency control, replica set control and achieve file cache isolation, all to substantial benefit.

9 Acknowledgements

We thank the anonymous FAST reviewers, our shepherd Jason Flinn, and others, including Hitesh Ballani, Thomas Karagiannis and Antony Rowstron for their feedback.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. In *Proceedings of USENIX FAST*, Dec. 2005.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of ACM SOSP*, Banff, Alberta, Canada, 2001.

[3] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, L. N. Bairavasundaram, T. E. Denehy, F. I. Popovici, V. Prabhakaran, and M. Sivathanu. Semantically-smart disk systems: Past, present, and future. *SIGMETRICS Perform. Eval. Rev.*, 33(4):29–35, Mar. 2006.

[4] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with Infokernel. In *Proceedings ACM SOSP*, SOSP '03, Bolton Landing, NY, USA, 2003.

[5] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of CIDR*, Asilomar, California, 2011.

[6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings ACM SOSP*, Copper Mountain, Colorado, USA, 1995.

[7] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *Proceedings ACM SOSP*, Cascais, Portugal, 2011.

[8] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, 14(4):311–343, Nov. 1996.

[9] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *Proceedings of ACM SIGCOMM*, Kyoto, Japan, 2007.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of USENIX OSDI*, Seattle, WA, 2006.

[11] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings USENIX OSDI*, Hollywood, CA, USA, 2012.

[13] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield. Strata: Scalable high-performance storage on virtualized non-volatile memory. In *Proceedings USENIX FAST*, pages 17–31, Santa Clara, CA, 2014.

[14] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[15] 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 *Proceedings of ACM SOSP*, Stevenson, Washington, USA, 2007.

[16] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of ACM SOSP*, Copper Mountain, Colorado, USA, 1995.

[17] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking:

An API for application control of SDNs. In *Proceedings of ACM SIGCOMM*, Hong Kong, 2013.

[18] FreeBSD. Freebsd handbook - GEOM: Modular disk transformation framework. `http://www.freebsd.org/doc/handbook/`.

[19] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of ACM SIGCOMM*, Barcelona, Spain, 2009.

[20] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of ACM ASPLOS*, Boston, Massachusetts, USA, 1992.

[21] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of USENIX ATC*, San Francisco, California, 1994.

[22] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *Proceedings of ACM SOSP*, Farminton, Pennsylvania, 2013.

[23] Intel Corporation. IoMeter benchmark, 2014. `http://www.iometer.org/`.

[24] M. Isard. Autopilot: Automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, Apr. 2007.

[25] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of ACM SIGCOMM*, Hong Kong, China, 2013.

[26] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of ACM SOSP*, Saint Malo, France, 1997.

[27] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proceedings of NSDI*, Lombard, IL, 2013.

[28] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of USENIX OSDI*, Vancouver, BC, Canada, 2010.

[29] K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson. Tools for the development of application-specific virtual memory management. In *Proceedings of ACM OOPSLA*, Washington, D.C., USA, 1993.

[30] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[31] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[32] C.-H. Lee, M. C. Chen, and R.-C. Chang. Hipec: High performance external virtual memory caching. In *Proceedings of USENIX OSDI*, Monterey, California, 1994.

[33] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of USENIX OSDI*, OSDI'12, Hollywood, CA, USA, 2012.

[34] R. Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[35] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, Mar. 2008.

[36] M. Mesnier, F. Chen, T. Luo, and J. B. Akers. Differentiated storage services. In *Proceedings of ACM SOSP*, pages 57–70, Cascais, Portugal, 2011.

[37] Microsoft. Virtual hard disk performance. `http://download.microsoft.com/download/0/7/7/0778C0BB-5281-4390-92CD-EC138A18F2F9/WS08_R2_VHD_Performance_WhitePaper.docx`.

[38] Microsoft Corporation. File system minifilter allocated altitudes (MSDN), 2013. `http://msdn.microsoft.com/`.

[39] Microsoft Corporation. File system minifilter drivers (MSDN), 2014. `http://code.msdn.microsoft.com/`.

[40] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23, Nov. 2008.

[41] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through I/O off-loading. In *Proceedings USENIX OSDI*, San Diego, CA, 2008.

[42] Oracle. Oracle Solaris ZFS administration guide. `http://docs.oracle.com/cd/E19253-01/819-5461/index.html`.

[43] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, Oct. 2014. USENIX Association.

[44] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, S. Vyas, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *Proceedings of the ACM SIGCOMM*, Hong Kong, 2013.

[45] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM*, Helsinki, Finland, 2012.

[46] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[47] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of USENIX OSDI*, Seattle, Washington, USA, 1996.

[48] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM*, Helsinki, Finland, 2012.

[49] SNIA. Exchange server traces. `http://iotta.snia.org/traces/130`.

[50] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *ACM Symposium on Cloud Computing (SOCC)*, Kohala Coast, HI, USA, 2015.

[51] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of ACM SOSP*, Farminton, Pennsylvania, 2013.

[52] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of ACM SOSP*, Copper Mountain, Colorado, United States, 1995.

[53] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstrow, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *Proceedings of ACM SOSP*, 2013.

[54] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for internet data transfer. In *Proceedings of USENIX NSDI*, NSDI'06, San Jose, CA, 2006.

[55] TPC Council. TPC-E. `http://www.tpc.org/tpce/`.

[56] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient mrc construction with shards. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, Santa Clara, CA, 2015. USENIX Association.

[57] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of USENIX ATC*, Monterey, California, 2002.

[58] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: a 4D network control plane. In *Proceedings of USENIX NSDI*, Cambridge, MA, 2007.

Flamingo: Enabling Evolvable HDD-based Near-Line Storage

Sergey Legtchenko *Microsoft Research*

Xiaozhou Li[*] *Microsoft Research*

Antony Rowstron *Microsoft Research*

Austin Donnelly *Microsoft Research*

Richard Black *Microsoft Research*

Abstract

Cloud providers and companies running large-scale data centers offer near-line, cold, and archival data storage, which trade access latency and throughput performance for cost. These often require physical rack-scale storage designs, e.g. Facebook/Open Compute Project (OCP) Cold Storage or Pelican, which co-design the hardware, mechanics, power, cooling and software to minimize costs to support the desired workload. A consequence is that the rack resources are restricted, requiring a software stack that can operate within the provided resources. The co-design makes it hard to understand the end-to-end performance impact of relatively small physical design changes and, worse, the software stacks are brittle to these changes.

Flamingo supports the design of near-line HDD-based storage racks for cloud services. It requires a physical rack design, a set of resource constraints, and some target performance characteristics. Using these Flamingo is able to automatically parameterize a generic storage stack to allow it to operate on the physical rack. It is also able to efficiently explore the performance impact of varying the rack resources. It incorporates key principles learned from the design and deployment of cold storage systems. We demonstrate that Flamingo can rapidly reduce the time taken to design custom racks to support near-line storage.

1 Introduction

Storage tiering has been used to minimize storage costs. The cloud is no exception, and cloud providers are creating near-line cloud storage services optimized to support cold or archival data, for example Amazon's Glacier Service [2], Facebook's Cold Data Storage [17], Google near-line storage [19] and Microsoft's Pelican [8]. In contrast to online storage [16], near-line storage trades data access latency and throughput for lower cost; access latencies of multiple seconds to minutes are normal and throughput is often lower or restricted.

To achieve the cost savings many of these near-line storage services use custom rack-scale storage designs, with resources such as power, cooling, network bandwidth, CPU, memory and disks provisioned appropriately for the expected workload. This is achieved through co-designing the rack hardware and software together, and systems like Pelican [8] and OCP Cold Storage [20] have publicly demonstrated that designing custom racks for near-line storage can result in significant cost savings. For example, in both of these designs there is insufficient rack-level power provisioned to allow all the hard disk drives (HDDs) to be concurrently spinning. By implication, the rack cooling is then only provisioned to handle the heat generated from a subset of the HDDs spinning. The goal of the rack's storage stack is to achieve the best possible performance without exceeding the physical resource provisioning in the rack.

The most common way of managing these constrained resources is by controlling how data is striped across HDDs, and by ensuring the individual IO requests are scheduled taking into account resource provisioning. In particular, the *data layout* defines the set of disks for a single IO that need to be read from or written to, and the *IO scheduler* defines the set of disks that need to be accessed for multiple IOs being concurrently performed. Our experience building near-line storage is that, given a well-designed storage stack, it is feasible to only redesign the data layout and IO scheduler in the stack to handle different rack designs and/or performance goals. Unfortunately, it is also the case that even simple and seemingly small design changes *require* a redesign of the data layout and IO scheduler. Designing the data layout and IO scheduler is challenging and time consuming even for experts, and it is hard to know if they are achieving the best possible performance from the rack.

Flamingo is a system that we use to help automate and

[*] Currently a PhD. student at Princeton.

reduce the complexity of designing near-line storage. It incorporates the many lessons learned during the design and deployment of Pelican. Flamingo uses a generalized storage stack that is derived from the one used in Pelican and described in [8], and a tool chain to automatically synthesize the configuration parameters for the storage stack. Flamingo requires a physical rack description, a set of resource descriptions in the form of resource constraints, and expected performance characteristics. Under typical operation the tool chain takes a few hours to produce the configuration parameters. Flamingo has been used to determine the impact of and to drive design and component changes to Pelican.

Flamingo is also able to help designers explore the physical rack design space by automatically quantifying the impact of varying the physical resource provisioning in the rack. It is able to determine the minimum increase in a resource, such as power, that would yield a change in performance. It is also able to determine the impact of using components with different properties, such as a new HDD with a different power profile. In such cases, it can also evaluate how much extra performance could be gained by reconfiguring the storage stack to exploit that component. Flamingo can handle significantly more complexity than a human and it is able to generate configurations and determine the likely performance of a physical design before it is even built.

This paper is organized as follows: Section 2 introduces near-line storage, Pelican and motivates the problems solved by Flamingo. Section 3 and 4 describe Flamingo, and the core algorithms used. Section 5 shows results, Section 6 describes related work and Section 7 concludes.

2 Background: Near-line storage

A cloud-scale storage service will consist of thousands of storage racks. A deployed rack will be used for many years, and then retired. Rack designs will be revised as price points for components change or newer versions are released. Hence, at any point in time, a small number of different storage rack designs will be deployed in a single cloud-scale storage service. A near-line storage rack will usually consist of servers and HDDs, and each server will run an instance of a storage stack. In online storage it is common to have 30-60 HDDs per server, while in near-line it can be 500+ HDDs per server. We provide a brief overview of Pelican as Flamingo uses many of its key principles, but for the full details see [8].

Pelican A Pelican rack has 1,152 HDDs and two servers. Each HDD is connected to a SATA 4-port multiplier, which is connected to a 4-port SATA HBA. Pelican uses PCIe to connect the 72 HBAs to the server, such that each HBA can be attached to either one of the servers. Power and cooling are provisioned to allow only a small fraction of the HDDs to concurrently be spinning and ready to perform IO (active) while the other HDD platters are spun down (standby).

HDDs are physically located in multiple physical resource domains: power, cooling, vibration and bandwidth. A Pelican power domain contains 16 HDDs and has sufficient power to support two HDDs transitioning from standby to active, with the 14 other HDDs in standby. A Pelican cooling domain has 12 HDDs and can provide sufficient heat dissipation to support one HDD transitioning from standby to active and 11 in standby. These domains represent constraints imposed by the physical rack, and combining these two constraints means that at most 96 HDDs can be concurrently active in a Pelican.

Violating physical resource constraints leads to transient failures, can increase hardware failure rates or simply decrease performance. Hence, the storage stack needs to ensure that the operating state of the rack remains within provisioned resources. Pelican handles these constraints by first carefully managing data layout. Each HDD is assigned to a group that contains 24 HDDs. The assignment is done to ensure all HDDs in a group can be concurrently transitioned from standby to active. Hence, at most 2 HDDs per group can be in the same power domain. Pelican stripes a stored file across multiple HDDs in the same group and, if required, erasure coding can be used. The Pelican prototype striped a file across eighteen HDDs with fifteen data fragments and three redundancy fragments. The mapping of HDDs to groups, the group and stripe size and erasure coding parameters are the *data layout configuration*. They are a function of number of HDDs in the rack, the physical resource constraints, required data durability, target throughput, and the capacity overhead. They are unique to a particular hardware design and set of resource constraints. To determine them is complex and during the original Pelican design it took many months to determine the correct parameters.

Within the Pelican software stack the other part which interacts closely with the physical rack and resource constraints is the IO scheduler. The IO scheduler determines the order in which IO requests are serviced, and it attempts to balance performance with fairness. Flamingo uses a new IO scheduler that is configurable and we discuss this in detail in Section 3.2.

Real-world lessons Pelican makes a number of simplifying assumptions. Notably, it assumes that an active HDD uses the same resources as a HDD transitioning from standby to active. This makes the problem more tractable, but can lead to resource underutilization that results in lower performance than theoretically supported. Some elements of the Pelican software stack

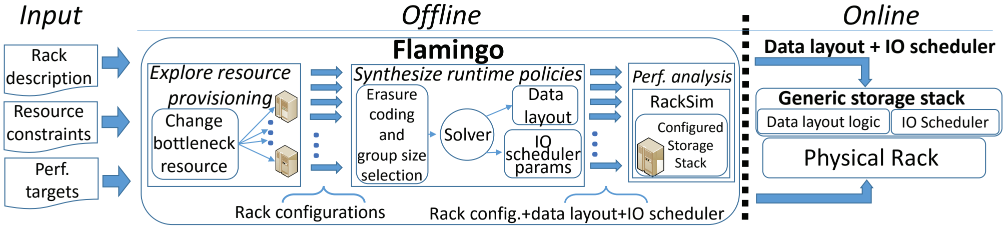


Figure 1: Flamingo overview.

proved to be very brittle to design changes. Subtle and often seemingly innocuous changes to the physical rack or components require significant redesign of the data layout and IO scheduler. For example, during the design of Pelican changing HDDs introduced new vibration issues, and also changed the power and cooling profiles. These changes provided the potential to have more HDDs to be concurrently active. However, without redesigning the data layout and IO scheduling in Pelican we were unable to unlock the better performance these HDDs could offer. This also requires using HDDs that offer similar or better properties compared to the HDDs we designed for originally. Subtle changes can result in resources being violated, which is often hard to detect when they do not lead to immediate failure. Finally, the cooling properties of a rack are a function of the ambient temperature in the data center in which it operates. This varies across data center designs and data center cooling technologies. This means that to maximize resource usage given a set of physical resources then a per data center data layout and IO scheduler is required.

When building complex near-line storage systems it is hard to accurately estimate the performance impact of small design changes. Simplistically, adding a few extra fans will increase the fraction of HDDs that can be concurrently active in a cooling domain, but it is hard to understand the impact this will have on higher-level performance metrics.

Finally, we also believe that, based on our experiences with Pelican for near-line storage, the underlying principle in Pelican of organizing the HDDs into groups that can be migrated between states concurrently is good. This allows resource conflicts to then be handled at the group level rather than the HDD level which lowers complexity and increases performance. The design of Flamingo therefore embodies this concept.

3 Flamingo

Flamingo leverages the fact that most of the Pelican storage stack is generic and independent of the hardware constraints; it uses the Pelican stack with a new configurable IO scheduler, and then uses offline tools to synthesize the data layout and IO scheduler parameterization for a given rack design.

Flamingo also supports the co-design of the rack hardware and software. Using a rack-scale event-based simulator it allows potential hardware resource configurations to be instantiated with a storage stack and then specific workloads replayed against them to understand higher-level performance. It also automatically explores the design space for resource provisioning to determine the performance impact of increasing the resources in a rack. This information can be used to both change the physical rack design, but also to help component manufacturers optimize their components to yield better performance.

Figure 1 shows the two main Flamingo components: an offline tool and a configurable storage stack. The offline tool has three phases. The first takes a physical rack description and a set of resource constraints, and iteratively generates new sets of resource constraints that effectively provide the potential for higher performance. The physical rack description and a single set of resource constraints represents a potential configuration, and requires a parameterized storage stack. The second phase then concurrently synthesizes for each unique configuration the parameters required for the data layout and the online IO scheduler. Target performance characteristics are provided, and the goal is to synthesize the configuration for the storage stack that meets or exceeds these performance characteristics. If it can be determined in this phase that a particular performance target cannot be met, then a policy can be specified to either relax the performance target or simply reject the configuration.

Unfortunately, not all performance targets can be verified as being met or exceeded during the second phase, and the final phase uses an accurate rack-scale discrete event simulator to empirically determine the expected performance. This does a parameter sweep using synthetic and real workloads evaluating micro- and macro-level performance for each configuration point. At the end of this offline process Flamingo has generated the storage stack parameters for each configuration, and the relative performance of each configuration. If the exploration of multiple configurations is not required, then the first stage can be skipped.

Failure Domains
$failure_{rate} : [comp_{name}, AFR]$
$failure_{domain} : [dom_{name}, comp_{name}, \{HDD_{id1}, HDD_{id2}, ..., HDD_{idN}\}]$

Resource Domains
$HDD_{cost} : [res_{name}, [cost_{standby}, cost_{spinningup}, cost_{spinning}]]$
$resource_{domain} : [dom_{name}, res_{name}, \{HDD_{id1}, HDD_{id2}, ..., HDD_{idN}\}, dom_{budget}, hard \vert soft]$

Figure 2: The rack description and resource constraints.

Flamingo is able to perform this in less than 24 hours for all rack configurations that we have tried. We now describe in detail how Flamingo works, starting with the information Flamingo requires.

3.1 Flamingo requirements

Flamingo requires a rack description that captures the different resources and their domains, a set of resource constraints expressed over these domains, the target performance characteristics and a state machine that captures the HDD operating states and resource usage. The rack description captures the physical properties of the rack and the physical relationship between HDDs and resource domains. The set of resource constraints capture the resource provisioning per resource domain. Given the tight relationship between the rack description and the resource constraints we use a single input file to capture them both, and its syntax is shown in Figure 2.

Each HDD is assigned a unique HDD_{id}. Flamingo allows an arbitrary number of HDD operating states. For simplicity here, we use only three states: standby, spinning up and active. In reality, there are several other potential states, including multiple lower power states, some of which keeps the platter spinning at a low RPM. Flamingo requires a state machine showing the possible HDD operating states and transitions between them. Operating states where IO requests can be serviced need to be explicitly identified, e.g. when the HDD is not spinning, or is spinning up, the HDD is unable to service IO requests. Operating states that can service IO requests are referred to as active. Given the current design of HDDs the tool supports only a single active state currently. Flamingo also needs to know which operating states are transient, e.g. spinning up.

The rack description and the resource constraints are expressed in terms of sets of HDDs. The rack description includes information about all (component) failure domains with their associated Annualized Failure Rates (AFR). If the AFR varies over time, then the worst case AFR is specified. Each *failure domain* is expressed in terms of the set of HDDs that would become inaccessible if the component fails. For example, if there is a tray that connects 16 HDDs together, a tray failure will lead to all 16 HDDs failing. So, if there are b trays then there will be b sets each containing 16 HDDs and an AFR will be associated with the tray.

The resource constraints are captured as *resource domains* which are expressed as a set of HDDs and an associated resource budget. Examples of resource domains may be power, cooling, bandwidth, and vibration. Individual HDDs will appear in multiple resource domains. Flamingo uses no explicit knowledge of any resource types, it treats all resources as simply names with associated constraints. This allows new resources to be easily incorporated within a design or arbitrarily changed. For example, half way through the design of Pelican we realized that the layout needed to handle vibration. Because Flamingo has no knowledge of resource types, a budget is associated with each resource domain set, and is simply a floating point number, and the unit is arbitrary. For example, for a power resource domain the unit could be Watts, and the original budget could be 50W. For each resource, Flamingo also needs the resource cost for operating in each state (HDD_{cost}), in the case of power these can be taken from the data sheet, e.g. spinning up may be 20W, active may be 10W and standby may be 1W. The *current cost* is the sum for all HDDs for them to operate in their current operating state. If a resource domain is *hard* then the current cost must not be higher than the budget, as this can cause long or short term failure. A $soft$ resource domain can be violated, but this will impact performance rather than failure rates. For each resource domain it is possible to set an upper bound that is used to control the search space when exploring changing the resource provisioning. By default, when exploring the design space Flamingo will look to increase a resource by the minimum that will allow at least one drive to transition to a different operating state. The minimum increase can also be specified. For example, a power domain may have an upper bound of 500W and a minimum increase of 25W.

Hierarchical resource domains can easily be expressed. For example, there could be a backplane that has 10 trays with 16 HDDs attached to it. A power domain can be created containing all 160 HDDs with a power budget. Then a power domain can also be created for each of the 10 trays. The sum of the tray budgets can exceed the budget for the backplane, but the backplane budget will never be exceeded.

Some resources are not necessarily additive, for example vibration. Using resource domains and budgets we have been able to handle these by emulating counting semaphores. The budget is used to capture the number of HDDs that are allowed in a particular state, and the HDD costs are set to zero or one. Using overlapping resource domains then also allows us to specify complex relationships. One set of resource constraints could be used to enforce that no neighboring HDDs can spin up concurrently, while a second one says that in a single tray only

4 can spin up concurrently. Flamingo will enforce both in its designs.

Finally, Flamingo also requires target performance characteristics; in particular data durability, physical servicing window, rack deployment lifetime, lower bound on bandwidth for file transfer, level of IO concurrency, capacity overhead for failure resilience and a fairness goal expressed as the trade-off in access latency versus throughout.

Simplifying assumptions The rack description allows arbitrary racks to be described. However, Flamingo makes two assumptions about the resource domains. First, for each resource defined *every* HDD in the rack must be specified in a $resource_{domain}$ description for that resource. For example, if power is a resource then each HDD must appear in at least one $resource_{power}$ definition. Second, each resource domain definition for a resource must include the same number of HDDs and be provisioned with the same budget. In the previous example of a tray and backplane power domain with different number of HDDs, this can be simply encoded by naming the resource domains differently, e.g. $power_{tray}$ and $power_{backplane}$. Finally, we assume that there is only one class of storage device specified. Flamingo can support other classes of storage device beyond HDD, provided they can be expressed as having multiple operating states over different resources. Flamingo could be extended to handle different storage device classes in the same rack, but this would increase the state space that Flamingo needs to handle. We believe these assumptions are reasonable and hold for all cold storage hardware that we are aware of, including Pelican and OCP Cold Storage. They simplify the data layout and in many cases reduce the number of inter-group constraints, improving concurrency and reducing overhead for the IO scheduler.

3.2 Flamingo Design

We now describe three important aspects of the core Flamingo design: the exploration of rack configurations, the data layout and the IO scheduler configuration.

3.2.1 Exploring rack configurations

The offline tool has three phases, the first explores the design space for resource provisioning in the rack. This is achieved by taking a configuration consisting of the rack description and a set of resource constraints and slowly relaxing the resource constraints. Each time a resource constraint is relaxed a new configuration is created which consist of the original rack description with the new set of resource constraints.

The intuition is that, if there are q resources, then there is a large q-dimensional space representing the set of all configurations. However, many of these configurations will vary resources that are not impacting the performance and can therefore be ignored. Hence, there is a surface being defined in the q-dimensional space of interesting configurations that can impact performance, and Flamingo is determining the configurations that lie on that surface. This can be a large space, for example a simple Pelican has $q = 4$ and, given multiple operating states for the HDDs, the total number of potential configurations is in the millions. However, the number of useful configurations will be considerably smaller.

Flamingo achieves this by determining the bottleneck resource for a given configuration. To calculate the bottleneck resource Flamingo calculates the number of HDDs in the rack (N) and, for each hard resource r, Flamingo determines the number of HDDs in each resource domain set for r, (N_r), and the per-resource domain budget (r_{budget}). Both N_r and r_{budget} will be the same for all resource domain sets for r. We define $cost_{highest}$ as the highest cost HDD operating state and the lowest as $cost_{lowest}$. The number of HDDs, (m_r), that can be in the highest operating state in each single resource domain is:

$$m_r = \left\lfloor \frac{r_{budget} - cost_{lowest} N_r}{cost_{highest} - cost_{lowest}} \right\rfloor \tag{1}$$

Across the entire rack the number of HDDs, (M_r), that can be operating in their highest cost operating state for the resource is:

$$M_r = (N/N_r) \times m_r \tag{2}$$

Flamingo generates for each resource r the value M_r. Given two resources, say $r = power$ and $r = cooling$, then *power* is more *restrictive* than *cooling* if $M_{power} < M_{cooling}$. To determine the bottleneck resource, the resources are ordered from most to least restrictive using their M_r values. The most restrictive resource is the *bottleneck resource*. The maximum number of HDDs that can be concurrently in their highest cost operating state M is then simply $M = M_{bottleneckresource}$. If there are two or more resources with equal M_r values then it is recorded that there are multiple bottleneck resources.

Once a bottleneck resource has been identified, the budget associated with the bottleneck resource is increased by δ. δ is the maximum of the smallest additional cost that will allow a single HDD in the bottleneck resource domain to transition to the next highest cost operating state and the specified minimum increase for the resource domain. The budget is then increased on the bottleneck domain by δ to create a new configuration.

If there is more than one bottleneck resource, then a new configuration is created where exactly one resource is selected to be relaxed. These configurations are then all used independently to recursively generate more configurations. The configuration exploration terminates when $M = N$, in other words, represents a fully provisioned rack or the bottleneck resource has reached the upper bound specified for it and cannot be increased. If the bottleneck resource cannot be increased it does not matter if other resources could be increased, they cannot yield better performance.

The number of configurations considered is dependent on the number of resources and the range over which the resources operate. Generating the configurations is fast, taking on the order of seconds on a high end CPU. Once all the configurations have been generated the storage stack parameters need to be calculated, which can happen in parallel for each configuration.

3.2.2 Data Layout

For each configuration Flamingo next needs to synthesize the data layout, and this involves two stages:

Groups and erasure coding Flamingo computes *groups* of HDDs such that each HDD belongs to a single group and there are sufficient resources across all resource domains to allow all the HDDs in the group to concurrently transition to their active state. We make a simplifying assumption that all groups are the same size, n. A file is stored on a subset of the HDDs in a single group, with k data and r redundant fragments generated for each file using erasure coding [28, 12].

The first stage is to calculate the group size. Flamingo does this by initially generating a set of candidate group sizes. Files are stored in a single group, therefore n should be large enough to store all fragments of a file even in presence of HDD failures, but small enough to maximize the number of groups that can be concurrently spun up. Because all HDDs in a group need to be able to spin up concurrently, $\lfloor M/n \rfloor$ groups can be simultaneously activated. To maximize resource utilization, we first enforce that $M \bmod n = 0$. For example, if $M = 96$ then both $n = 21$ and $n = 24$ allow the same number of concurrently active groups: 4, but only $n = 24$ fulfills $96 \bmod n = 0$. For $M = 96$, this restricts the possible group sizes to $n = \{1, 2, 3, 4, 8, 12, 16, 24, 32, 48, 96\}$. We refer to this as the *candidate set*. If the set is empty, then Flamingo stops processing the configuration and generates an error.

Flamingo then determines a set of values for erasure coding parameters k and r. The choice of values are a function of *(i)* the required data durability, *(ii)* the component failure rates, *(iii)* the storage capacity redundancy overhead *i.e.,* $\frac{r}{k+r}$, *(iv)* the interval between physically servicing a rack, and *(v)* the lower bound on per-file read or write throughput. The first four parameters are used in a simple failure model to generate a set of possible $k+r$ values. The fifth parameter is then used as a threshold for values of $k+r$, removing combinations that would yield too low throughput, so we look for $k \times HDD_{bandwidth} \geq target$. The result is an ordered list consisting of $k+r$ pairs that provide the specified durability ranked by the storage capacity overhead ($\frac{r}{k+r}$). If the set is empty, then an error is raised and Flamingo stops processing this configuration. The same model is also used to calculate f, an estimate of the maximum number of HDDs expected to fail during a rack service interval. This is calculated assuming that failure recovery is performed at the rack level which can be done by the Flamingo storage stack. However, if failure recovery is handled at a higher level across storage racks, then f can be configured to always be zero.

Given the candidate set of possible group sizes, the ranked $(k+r)$ list and f, Flamingo needs to select the lowest value for n from the candidate set, such that $k + r + f \leq n$. This maximizes the number of concurrently active groups and therefore the number of concurrent IO requests that can be serviced in parallel. So, given the previous candidate groups sizes, if the smallest value of $(k,r) = (15,3)$ and $f = 2$ then $n = 24$ will be selected. If M/n is less than the specified concurrent IO request target, Flamingo stops processing the configuration.

The Flamingo storage stack attempts to distribute the stored data in a group uniformly across all the HDDs in a group. When a group is accessed all n HDDs are concurrently migrated to the new state, rather than k. The reason to spin up $k+r$ is to allow us to read the data when the first k HDDs are ready to be accessed. The Flamingo runtime spins up the entire n (e.g. $k+r+f$) HDDs opportunistically, because if another request arrives for the group we are able to service it without waiting for potentially another drive to spin up.

Mapping HDDs to groups Once n has been determined, Flamingo next needs to form l, where $l = N/n$, groups and assign each HDD to exactly one group. The assignment is static, and transitioning *any* HDD in a group to a new state that would violate any hard resource constraint means the entire group cannot transition.

The assignment must also try to maximize IO request concurrency, which means maximizing the number of groups that can concurrently transition into active, where the upper bound is M/n. However, ensuring a mapping that achieves this is non-trivial because each HDD assigned to a group potentially conflicts with other groups in all its domains. This will lead to inefficient data layouts, in which every group conflicts with $l-1$ groups, achieving very low IO request concurrency e.g. one.

The number of possible assignments grows exponentially with the number of HDDs. To make this tractable, we use a custom designed solver that restricts the search space and selects the best group assignment according to a set of performance-related characteristics and heuristics. The solver exploits the observation that many resource domains are not composed of arbitrary HDDs but are rather defined by their physical location in the rack. For instance, the power domain would correspond to a backplane. The solver derives a coordinate system that captures this physical layout from the rack description and assigns a d-dimensional coordinate to each HDD, where d is the number of resource domain types.

The solver tries to form groups of HDDs that are close to each other in the coordinate space and do not conflict in any resource domain. It does this by initially generating different ordered vectors of the HDDs. This is achieved by changing the starting coordinate and ranking the coordinates on different dimensions. Hence, if each HDD has an (x,y) coordinate, one ranking would be generated by ordering on x then y and another one would be generated ranking y then x. The ordering function is dimension specific, so it can generate smallest to largest on x, but for coordinates where x is equal, rank largest to smallest on y. This generates multiple orderings of the HDDs. For each ordered vector created Flamingo greedily attempts to assign HDDs to groups, using a number of different heuristics to control into which group the next HDD is mapped. This is deterministic, no randomization is used. Intuitively, this finds good assignments because the group structure exploits the physical symmetry of the rack topology, forming sets of groups that conflict in all domains and are independent from the rest of the rack.

For each iteration, if the solver finds a solution where all HDDs are successfully assigned to groups such that all the HDDs in each group can concurrently transition operating states, then Flamingo needs to measure the quality of each solution. The metric of importance is the level of IO request concurrency that can be achieved by the data layout. An efficient solution will always allow any arbitrary selected M/n groups to be concurrently in their highest operating state.

Even with the custom solver this metric will need to be calculated potentially thousands of times per configuration. Hence, Flamingo uses a number of fast-to-compute heuristics. First, Flamingo determines if the groups are symmetric. We take each resource constraint and replace the HDD identifier in the definitions with the group identifier. For each group we then look at each resource domain in which it is present, and count the number of other unique groups that are present in each. We refer to these groups as conflicting groups. If, across all groups, the cardinality of the conflicting groups is the same, then the groups are symmetric. Each group impacts the same number of other groups. Further, the expected upper bound on the number of groups that should conflict with each group can be calculated.

Flamingo then uses a sub-sampling of the space to check configurations, and in particular explores sample sets consisting of less than or equal to M/l groups, checking if they can be successfully concurrently transitioned. The sub-sampling also estimates a lower bound on the number of groups that can be active (e.g. spinning) and another group transitioned into an active state. The expected number is determined as a function of M and again sub-sampling is used to estimate the lower bound. The number of samples can be varied *per configuration*.

If the ranking is shown to have no examples that violate the expected performance for these heuristics, then it is marked *efficient* and the solver stops. Otherwise, the solver records the quality of the metrics and continues to iterate through rankings. If all rankings have been checked and no efficient solutions found, then the solver selects the best solution found but marks the result *inefficient*. The output of the solver is a set of HDD to group mappings which define the data layout.

3.2.3 IO scheduler

Once data layout is complete the IO scheduler configuration needs generating. The IO scheduler in the storage stack receives IO requests and controls the order in which they are executed. It also controls when groups transition states. If it has a request for a group that is currently not active, it will ensure that the group becomes active and then issues the request to be serviced. It has to ensure that during operation the order in which groups transition between states does not violate the resource constraints. In order to do this, the IO scheduler needs to understand the relationship between groups, and we achieve this using a set of constraints between groups. The inter-group constraints capture the resource sharing relationships between groups, and allow the IO scheduler to determine which groups can concurrently be spinning.

To generate these IO scheduler *group constraints* Flamingo translates the resource constraints from being HDD based to group based. Each HDD identifier in each resource constraint is replaced with the HDD's group identifier and a weight, w_{id} initially set to one. For each resource constraint, all references to same group identifier are combined into a single entry with w_{id} being set to the number of references. The budget and associated per state costs for the original resource constraints are kept. If there are multiple group constraints which have *exactly* the same groups represented, the one with the most restrictive budget is kept. Flamingo outputs the set of group constraints.

The online IO scheduler in the storage stack uses the group constraints to control which groups can be spun up. It maintains a per-group queue for IO requests that are yet to be issued and an operating state for each group,

Rack	#HDDs	#HDDs/server	#domains	avg. HDDs/domain
OCP	240	240	73	15
Pelican	1152	576	1111	10
Rack_A	1152	576	1039	22
Rack_B	1152	576	1087	11
Rack_C	1152	576	1063	14
Rack_D	960	480	942	9
Rack_E	1920	960	1883	9

Table 1: Core properties of the seed racks.

which maps onto the HDD states, e.g. standby, spinning up, and spinning. The IO scheduler also maintains for each group constraint a balance, equal to the sum of $cost_{state} \times w_{id}$ for each group. In general, a group can transition to a new state if, for all group constraints, the change in balance is within the group constraint budget.

The IO scheduler is invoked each time a IO request is received, or an IO or group state transition completes. It needs to determine if there is a request that can be now serviced or if a group transition needs to occur to service queued requests.

The choice of which group or groups to transition is a function of the per group queue depth and the current queuing delay for the head request. There is a trade-off between latency and throughput, there is a throughput penalty for changing group state, but there is a latency penalty of making requests queue for longer. The performance characteristics specified control this trade-off. If the IO scheduler decides that a group g needs to transition state, the IO scheduler iterates over the groups and, using the group constraints, greedily identifies sets of groups that could be transitioned to free the resources to allow g to transition. If none or insufficient resources are found, then the scheduler waits for in-flight requests or group transitions to complete. If there are a number of sets of groups, then the scheduler selects the groups to transition based on their queue depth and head request delay. When it has selected a group or groups to transition, if there are spare resources in any group constraints, the IO scheduler is invoked again to allow further groups to transition state.

4 Evaluation

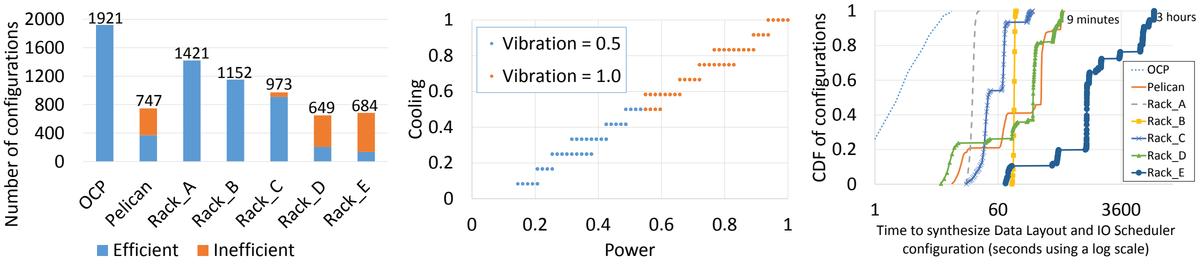
We now evaluate the performance of Flamingo using seven seed rack configurations, including Pelican and the Facebook/OCP Cold Storage Design [20]. The OCP Cold Storage Rack contains two independent servers and 16 Open Vault chassis, each filled with two trays of 15 HDDs with sufficient power and cooling to support one active drive and 14 in standby. The tray is a vibration domain, and each server is connected to 8 chassis using SAS containing a combined 240 HDDs and independent of the other server in the rack. Hence, this rack configuration is a half rack consisting of a server and 240 HDDs. Details of the software stack have not been released, but a Pelican-like storage stack is needed as most HDDs will be in standby. The other five racks are based on other cold storage designs and we refer to them as Rack_A to Rack_E. Table 1 summarizes the number of HDDs, the number of resource domains and the average HDDs per resource domain for each of them. All the designs have multiple bandwidth resource domains, to capture the bandwidth from the HDDs to the server, as well as power, cooling and vibration domains. Racks A to E are all credible physical hardware design points for cold storage which vary the power, cooling, and HDD density (hence vibration and HDD-to-server bandwidth). We have built out Pelican and Rack_D. We put no upper bounds or increment limits on the resource domains for any resources in any rack.

Flamingo uses a rack-scale discrete event simulator to estimate the performance of rack with the synthesized data layout and IO scheduler. The simulator is based on the discrete event simulator used to evaluate Pelican, which we have extended to support arbitrary physical rack topologies and to use the constraint-aware IO scheduler described. It models HDDs, network bandwidth and the server-to-HDD interconnect, and is configured with mount, unmount and spin up latency distributions from measurements of real archive class HDDs and has been cross validated against real rack-scale storage designs (for example the prototype Pelican [8]).

In the experiments we used a cluster of servers, each with two Intel Xeon E5-2665 2.4Ghz processors and 128 GB of DRAM. For each configuration we do a parameter sweep over a range of possible workload characteristics. A sequence of client read requests for 1 GB files is generated using a Poisson process with an average arrival rate $\lambda = 0.0625$ to 5. Beyond $\lambda = 5$ the network bandwidth becomes the bottleneck for all racks. The read requests are randomly distributed across all the files stored in the rack. We simulate 24 hours, and gather statistics for the last 12 hours when the simulation has reached a steady state. We believe this workload allows comprehensive comparison of the rack configurations.

4.1 Flamingo performance

First we evaluate the performance of Flamingo exploring the resource design space and creating the configurations from the initial rack description. For each of the seven racks, the time to generate the derived configurations is less than three seconds on a single server. Figure 3(a) shows the total number of configurations derived for each rack. Across each of the racks there is wide variance in the number of configurations derived, 649 to 1,921. The



(a) Number of configurations.

(b) Configurations for Pelican.

(c) Execution time.

Figure 3: Base performance of Flamingo.

number of configurations is a function of the resource domains and which ones are the bottleneck resource in each configuration. Across all racks 7,547 configurations are created.

Figure 3(a) also shows for each rack the fraction of configurations for which the generated data layout was considered efficient or inefficient by Flamingo. If Flamingo finds a configuration in which *(i)* all HDDs are allocated to a group, and *(ii)* all HDDs in a single group can be migrated from standby to active concurrently, it uses the fast heuristics as described to determine if the solution is efficient or inefficient. If these two conditions do not hold then the configuration is marked as having no solution, however for all 7,547 configurations a layout (efficient or inefficient) was found.

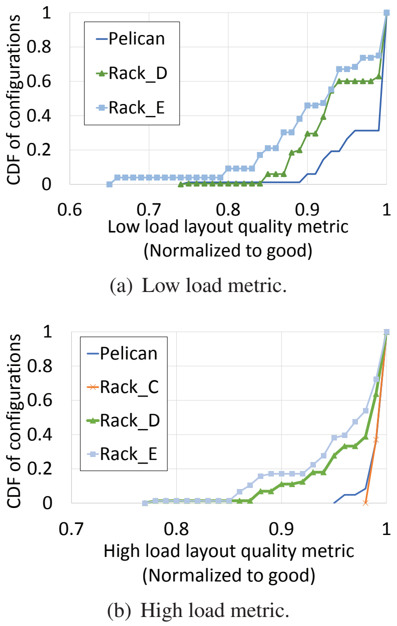
Figure 3(b) shows the fraction of power, cooling and vibration provisioned in each configuration derived from the Pelican rack. Each point represents a configuration and power and cooling are shown on the two axes, normalized to being fully provisioned. Hence a value of (1,1) means that the resource is sufficiently provisioned to have all HDDs in the rack in their most resource-consuming operating state. The vibration domain is expressed using the color of the point, again normalized to fully provisioned. Although showing only three resources, Figure 3(b) demonstrates how Flamingo traverses the design space, incrementing the bottleneck resource each time. For each configuration we increment the bottleneck resource by the smallest unit that will allow a single HDD to be in a more expensive operating state. However, this does not necessarily mean that the bottleneck resource changes from the previous configuration. In Figure 3(b) the impact of this can be seen where there are multiple power configurations for each step in the cooling.

Execution time Next, we consider the execution time of Flamingo. The solver used to synthesize the data layout and IO scheduler for each configuration runs as an independent single threaded process for each configuration. Flamingo runs one instance of the solver on each core of each server it is run on. Figure 3(c) shows a CDF of derived racks versus time taken to generate the data layout and the IO scheduler configuration for each configuration. The time taken is a function of the complexity of the configuration and the number of HDDs, and for all except those for Rack_E, none takes more than 9 minutes. In the worst case, for a configuration derived from Rack_E it takes 3 hours and the median for this rack is 20 minutes. The time taken for Flamingo is heavily dominated by the number of HDDs; as the number of HDDs increases the size of the state space to search increases faster then linearly. Table 1 shows Rack_E has 1,920 HDDs, almost a factor of two larger than the other racks. Our solver is deterministic and can report as it executes both the current best found solution and the fraction of the search space it has explored.

Once the data layout and IO scheduler parameters have been created, Flamingo runs the simulator to estimate the performance of each configuration. The time taken by the simulator is a function of the workloads evaluated. The workloads used in this paper allow a comprehensive exploration of the relative performance and across all 7,547 configurations we observed a mean execution time of 45 minutes per configuration, with a maximum of 83 minutes. As with the parameter generation, the simulations can be run concurrently.

4.2 Data layout quality

Next we quantify the quality of the data layout generated for each configuration. Flamingo considers a layout as efficient or inefficient, and stops searching once it finds one it considers efficient. Analytically it is impossible to determine if a layout is optimal at these scales, so instead we use two metrics. The first metric is the number of groups that can be concurrently spun up, which is a good indicator of performance under low load. For a configuration we can determine the bottleneck resource, and using that we can calculate an upper bound on the number of groups that should be able to be concurrently active in their highest state (m). We then generate a ran-



(a) Low load metric.

(b) High load metric.

Figure 4: Quality of data layout.

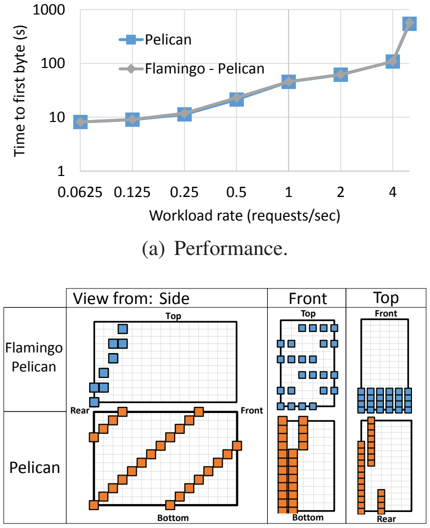
dom test ordering of the n groups in the configuration. For each configuration we greedily try to spin up the first k groups, where $k = 1, 2, 3, \ldots, m$. If we are able to migrate the HDDs in the k groups from standby to active concurrently without violating any resource constraints, we remove the group at position k in the test ordering and try again. Eventually, there are no further groups to remove from the test ordering and $k < m$, or $k = m$. We repeat this 250,000 times ensuring a unique ordering for each trial and record k and normalize it to m. We refer to this as the low load quality metric and reflects the level of concurrency achievable under low load.

The second metric is the number of groups that can be concurrently active and still allow an additional group to become active. This is a good indicator of performance under high load. We use the same process to calculate this metric, except instead of concurrently migrating the HDDs in all group from standby to active, we leave $k-1$ active and try to transition the kth group to the spinning up state. Again, we can calculate the value of m for this metric using the bottleneck resource. We refer to this as the high load quality metric. If, for all 250,000 trials, both metrics are one then the data layout is considered *good* otherwise it is considered *bad*. These metrics are not used by the Flamingo solver as they take many hours to compute for each *single* solution, and need to be computed for all the large number of solutions considered.

Table 2 compares using these metrics to the fast heuristics used by Flamingo showing the total number of configurations, the number of these configurations that Flamingo said it could generate an efficient layout, and then the number of false positives and false negatives. A *false positive* is a configuration marked efficient by Flamingo but bad by the metrics. A *false negative* is marked inefficient by Flamingo but good by the metrics. Three racks: OCP, Rack_A and Rack_B, have efficient and good layouts for all configurations.

Rack	Configurations	Efficient	False Positive	False Negative
OCP	1921	1921	0	0
Pelican	747	369	0	0
Rack_A	1421	1421	0	0
Rack_B	1152	1152	0	0
Rack_C	973	909	361	0
Rack_D	649	205	0	9
Rack_E	684	135	39	39

Table 2: Quality of Flamingo's data layout heuristics.



(a) Performance.

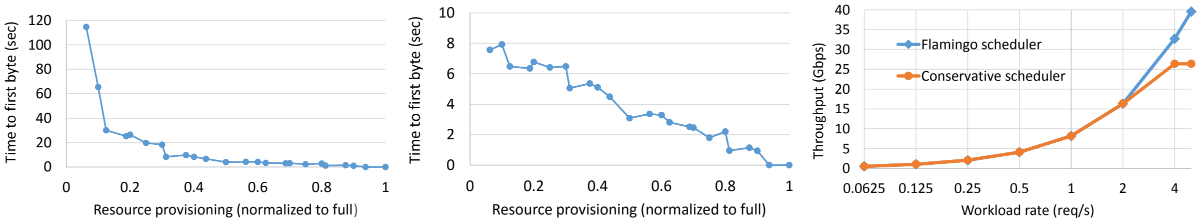
(b) Group layout.

Figure 5: Flamingo vs Pelican.

In order to understand further the quality of inefficient configurations, as well as the false positives and negatives, Figure 4 shows a CDF of configurations versus both quality metrics when the metrics are not one (OCP, Rack_A and Rack_B omitted). The low load metric is not 1 for only three racks, and in all cases the median is above 0.9. Under the high load metric all solutions are at 0.75 or higher for the four racks. This shows that even when a rack is not efficient, the quality of the solutions is high.

4.3 Storage performance

The last set of experiments evaluate the rack performance when using the IO scheduler and the data layout synthesized by Flamingo. First we compare how Flamingo performs to a manually-designed solution. To do this we



(a) Latency, high workload rate - Rack_E

(b) Latency, low workload rate - Rack_E

(c) Throughput - OCP rack

Figure 6: Performance of generated layouts and IO scheduler.

take the rack configuration which is equivalent to the Pelican rack used in [8] and compare its performance with the Flamingo generated data layout and IO scheduler constraints.

We first consider the time to first byte, which is the time between a request being issued by a client and the first data byte being sent to the client. This includes all queuing and spin up delays. Figure 5(a) shows the time to first byte as a function of the workload rate. Flamingo-Pelican is using the data layout and IO scheduler configuration synthesized by Flamingo and Pelican is the manually designed layout and IO scheduler. The time to first byte for Pelican and Flamingo-Pelican is virtually identical across all workload rates. This is true for other properties measured by the simulator, including throughput, which are omitted due to space constraints.

The fact that Flamingo-Pelican matches the original performance is interesting. The data layout differs significantly from the original Pelican layout. Figure 5(b) shows a representation of the rack as a 3D array of HDDs shown from the front, side and top comparing the layout of group zero for each. Each colored square represents a HDD, and other groups can be thought of as rotations of this group. Flamingo groups cover different individual resource domains compared to the original Pelican.

The next experiment explores the ability of Flamingo to exploit the resources provisioned in each rack. Ideally, the performance should increase as the provisioning of the bottleneck resource increases. For each of the 7,547 configurations for the seven racks we measure the time to first byte as a function of resource provisioning. Figure 6(a) shows the time to first byte versus the resource provisioning. Due to lack of space, we use a single (high) workload rate of 4 requests per second and only show results for one randomly selected rack (Rack_E). The other racks show similar trends in the results. To quantify the resource provisioning we use its M value normalized by the total number of HDDs in the rack representing a fully provisioned rack. Recall that M is the maximum number of HDDs that can be concurrently in their highest cost operating state, and is a function of the bottleneck resource. While deriving configurations, Flamingo increases the bottleneck resource budget by a value δ which is potentially less than the cost of allowing a HDD to operate in the highest cost operating state, hence several configurations can share the same M value.

From Figure 6(a) we see that the time to first byte generally decreases as provisioning increases, meaning that Flamingo is able to adapt to the increased resource provisioning, achieving better performance with more resources. The performance improvement is not monotonic: in some cases, the resource provisioning increase does not decrease the time to first byte. This happens because Flamingo attempts to optimize for general performance across multiple metrics, rather than just time to first byte. Figure 6(a) also shows that the decrease in time to first byte is not linear as the provisioning is increased. When resources are scarce, even a slight increase in provisioning leads to significantly better performance. For example, increasing the provisioning from 0.06 to 0.1 leads to a time to first byte decreased by nearly 80% on average for Rack_E. We observe this trend for all seven racks, meaning relatively low provisioned racks can achieve a performance close to fully provisioned ones. Intuitively, this happens because for the given workload, resource provisioning within the rack is not necessarily the bottleneck. At some point, the performance becomes limited by external factors such as the bandwidth from the rack to the data center fabric (in this case 40 Gbps). Notably, the exact benefit of increasing resources is very different for each initial rack description, e.g. for Rack_A, the time to first byte decreases by 80% only when resource provisioning reaches 0.68.

To illustrate this further we use a low workload rate of 0.0625 requests per second. Figure 6(b) shows the time to first byte versus the resource provisioning for Rack_E. For this low workload rate, the IO scheduler is unable to do extensive batching of requests and needs to frequently transition between groups. The rack bandwidth is not the bottleneck and the IO scheduler can benefit from more resources in the rack to increase concurrency of group transitioning. As a result, the time to first byte decreases almost linearly as provisioning increases. Resource provisioning depends on multiple factors internal *and* external to the rack. Tools like Flamingo provide great benefit when co-designing a rack and storage stack for a particular workload.

The final experiment evaluates the benefit for the IO scheduler to dynamically manage the available resources. Pelican made the simplifying assumption that HDDs could have two states; standby and active. This leaves some resources unused which means that it will be able to keep fewer groups concurrently active, but has the benefit of being much simpler and we refer to this as a conservative IO scheduler. Allowing an arbitrary number of states with differentiated costs requires the IO scheduler to track transitions between each state for all HDDs, and ensuring that budgets will not be violated by each transition to a new state. We compare the conservative and the Flamingo schedulers using the OCP rack. For this default configuration power is the bottleneck resource, with sufficient provisioning to allow two groups to spin up concurrently. Figure 6(c) shows the throughput as a function of the workload rate. For workloads with higher request rates of 2 or more requests/second, the Flamingo IO scheduler outperforms the conservative one. It does this because, at the higher loads, it can keep more groups concurrently spinning; it is able to keep up to three groups concurrently spinning as opposed to two for the conservative scheduler, allowing one more requests to be processed in parallel. For lower workload rates, the performance is dominated by the number of groups that can spin up concurrently as the IO scheduler needs to frequently transition between groups, so the Flamingo IO scheduler offers no additional performance. It should be noted that if the HDDs can operate in lower power RPM states which offer faster transitioning to active, the benefit of the finer-grained resource management in the Flamingo IO scheduler would enable increased performance for all workload rates.

5 Related Work

Flamingo addresses the challenges of designing rack-scale systems for near-line storage. To reduce costs physical resources are typically constrained. The storage stack needs to maximize performance without violating the constraints making data layout and IO scheduling key. In contrast, traditional storage is provisioned for peak performance. There have been proposals for systems like MAID [10], as well as other power efficient storage systems [22, 18, 4, 24, 32], that allow idle disks to spin down. Data layout and mechanisms to handle spun down disks is important in all their designs. Pergamum [22] used NVRAM to handle meta-data and other small writes, effectively providing a write-back cache used when the disks are spun down. Hibernator [32] supports low RPM disk modes and dynamically determines the proportion of disks in each mode in function of the workload. Rabbit [4], Sierra [24] and PARAID [27] achieve power-proportionality through careful data layout schemes, but in these systems fine-grained provisioning of physical resources is not done at design time.

There has been work on automatic configuration of RAID storage [21, 30, 29, 3, 1, 6, 7], for example to design RAID configuration that meet workload availability requirements [3, 1, 6, 9]. These use a solver that takes declarative specifications of workload requirements and device capabilities, formulates constraint representation of each design problem, and uses optimization techniques to explore the search space of possible solutions computing the best RAID level for each logical unit of data on disk. Designs often include an online data migration policy between RAID levels [30, 7]. Flamingo is designed to optimize the physical resource utilization in the rack, working at a larger scale and explicitly handling a large number of constrained resources.

Tools to manage the design and administration of enterprise [26, 5], cluster [15] and wide-area [13] storage that optimize for data availability, durability and capital cost as primary metrics offline but do not consider fine-grained resource management or online IO scheduling.

Flamingo provides quantitative answers to questions about hypothetical workload or resource changes and their impact on performance. This is similar to prior work [25, 23, 11]. For example, [23] evaluates different storage provisioning schemes, which helps understanding trade-offs. In contrast, Flamingo complements the analysis by creating the data layout and IO scheduling policies for each configuration.

More generally, [14] proposes automatically generating data layout for data-parallel languages. Remy [31], given network characteristics and transport protocol targets, synthesizes a network congestion control algorithm. Flamingo has the same high-level goal: to make systems less brittle.

6 Conclusion

Flamingo is designed to simplify the development of rack-scale near-line storage. Flamingo has two high-level goals: first to synthesize the data layout and IO scheduler parameters for a generic storage stack for cloud near-line storage racks. The second aspect is that Flamingo supports the co-design of rack hardware and software, by allowing an efficient exploration of the impact of varying the resources provisioned within the rack.

Acknowledgments We would like to thank Aaron Ogus, the reviewers and our shepherd Mustafa Uysal for their feedback.

References

[1] ALVAREZ, G. A., BOROWSKY, E., GO, S., ROMER, T. H., BECKER-SZENDY, R., GOLDING, R., MERCHANT, A., SPASOJEVIC, M., VEITCH, A., AND WILKES, J. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst. 19*, 4 (Nov. 2001), 483–518.

[2] Amazon glacier. `http://aws.amazon.com/glacier/`, August 2012.

[3] AMIRI, K., AND WILKES, J. Automatic Design of Storage Systems To Meet Availability Requirements. Tech. Rep. HPL-SSP-96-17, Computer Systems Laboratory, Hewlett-Packard Laboratories, August 1996.

[4] AMUR, H., CIPAR, J., GUPTA, V., GANGER, G. R., KOZUCH, M. A., AND SCHWAN, K. Robust and Flexible Power-proportional Storage. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC '10.

[5] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. C. Hippodrome: Running circles around storage administration. In *FAST* (2002), vol. 2, pp. 175–188.

[6] ANDERSON, E., SPENCE, S., SWAMINATHAN, R., KALLAHALLA, M., AND WANG, Q. Quickly Finding Near-optimal Storage Designs. *ACM Trans. Comput. Syst. 23*, 4 (Nov. 2005), 337–374.

[7] ANDERSON, E., SWAMINATHAN, R., VEITCH, A. C., ALVAREZ, G. A., AND WILKES, J. Selecting raid levels for disk arrays. In *FAST* (2002), vol. 2, Citeseer, pp. 189–201.

[8] BALAKRISHNAN, S., BLACK, R., DONNELLY, A., ENGLAND, P., GLASS, A., HARPER, D., LEGTCHENKO, S., OGUS, A., PETERSON, E., AND ROWSTRON, A. Pelican: A Building Block for Exascale Cold Data Storage. In *OSDI* (Oct. 2014).

[9] BOROWSKY, E., GOLDING, R., MERCHANT, A., SCHREIER, L., SHRIVER, E., SPASOJEVIC, M., AND WILKES, J. Using attribute-managed storage to achieve qos. In *Building QoS into distributed systems*. Springer, 1997, pp. 203–206.

[10] COLARELLI, D., AND GRUNWALD, D. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* (2002), SC '02, IEEE Computer Society Press, pp. 1–11.

[11] EL MALEK, M. A., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M., PRASAD, M., SALMON, O., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Early Experiences on the Journey Towards Self-* Storage. *IEEE Data Eng. Bulletin 29* (2006).

[12] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., YEKHANIN, S., ET AL. Erasure coding in windows azure storage.

[13] KEETON, K., SANTOS, C. A., BEYER, D., CHASE, J. S., AND WILKES, J. Designing for disasters. In *FAST* (2004), vol. 4, pp. 59–62.

[14] KENNEDY, K., AND KREMER, U. Automatic Data Layout for Distributed-memory Machines. *ACM Trans. Program. Lang. Syst. 20*, 4 (July 1998), 869–916.

[15] MADHYASTHA, H. V., MCCULLOUGH, J., PORTER, G., KAPOOR, R., SAVAGE, S., SNOEREN, A. C., AND VAHDAT, A. scc: cluster storage provisioning informed by application characteristics and slas. In *FAST* (2012), p. 23.

[16] MARCH, A. Storage pod 4.0: Direct wire drives - faster, simpler, and less expensive. `http://blog.backblaze.com/2014/03/19/backblaze-storage-pod-4/`, March 2014.

[17] MORGAN, T. P. Facebook loads up innovative cold storage datacenter. `http://tinyurl.com/mtc95ve`, October 2013.

[18] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write off-loading: Practical power management for enterprise storage. *Trans. Storage 4*, 3 (Nov. 2008), 10:1–10:23.

[19] NEWSON, P. Whitepaper: Google cloud storage nearline. `https://cloud.google.com/files/GoogleCloudStorageNearline.pdf`, March 2015.

[20] OPEN COMPUTE STORAGE. `http://www.opencompute.org/projects/storage/`.

[21] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1988), SIGMOD '88, ACM.

[22] STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. Pergamum: Replacing Tape with Energy Efficient, Reliable, Disk-based Archival Storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), FAST'08.

[23] STRUNK, J. D., THERESKA, E., FALOUTSOS, C., AND GANGER, G. R. Using utility to provision storage systems. In *FAST* (2008), vol. 8, pp. 1–16.

[24] THERESKA, E., DONNELLY, A., AND NARAYANAN, D. Sierra: Practical power-proportionality for data center storage. In *Proceedings of the Sixth Conference on Computer Systems* (2011), EuroSys '11.

[25] THERESKA, E., NARAYANAN, D., AND GANGER, G. R. Towards Self-predicting Systems: What if You Could Ask "What-if"? *Knowl. Eng. Rev. 21*, 3 (Sept. 2006), 261–267.

[26] WARD, J., O'SULLIVAN, M., SHAHOUMIAN, T., AND WILKES, J. Appia: Automatic storage area network fabric design. In *FAST* (2002), vol. 2, p. 15.

[27] WEDDLE, C., OLDHAM, M., QIAN, J., WANG, A.-I. A., REIHER, P., AND KUENNING, G. Paraid: A gear-shifting power-aware raid. *ACM Transactions on Storage (TOS) 3*, 3 (2007), 13.

[28] WICKER, S. B., AND BHARGAVA, V. K. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[29] WILKES, J. Traveling to Rome: A Retrospective on the Journey. *SIGOPS Oper. Syst. Rev. 43*, 1 (Jan. 2009), 10–15.

[30] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID Hierarchical Storage System. *ACM Trans. Comput. Syst. 14*, 1 (Feb. 1996), 108–136.

[31] WINSTEIN, K., AND BALAKRISHNAN, H. TCP Ex Machina: Computer-generated Congestion Control. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (2013), SIGCOMM '13.

[32] ZHU, Q., CHEN, Z., TAN, L., ZHOU, Y., KEETON, K., AND WILKES, J. Hibernator: helping disk arrays sleep through the winter. In *ACM SIGOPS Operating Systems Review* (2005), vol. 39, ACM, pp. 177–190.

PCAP: Performance-Aware Power Capping for the Disk Drive in the Cloud

Mohammed G. Khatib and Zvonimir Bandic
WDC Research
{mohammed.khatib,zvonimir.bandic}@hgst.com

Abstract

Power efficiency is pressing in today's cloud systems. Datacenter architects are responding with various strategies, including capping the power available to computing systems. Throttling bandwidth has been proposed to cap the power usage of the disk drive. This work revisits throttling and addresses its shortcomings. We show that, contrary to the common belief, the disk's power usage does not always increase as the disk's throughput increases. Furthermore, throttling unnecessarily sacrifices I/O response times by idling the disk. We propose a technique that resizes the queues of the disk to cap its power. Resizing queues not only imposes no delays on servicing requests, but also enables performance differentiation.

We present the design and implementation of PCAP, an agile performance-aware power capping system for the disk drive. PCAP dynamically resizes the disk's queues to cap power. It operates in two performance-aware modes, throughput and tail-latency, making it viable for cloud systems with service-level differentiation. We evaluate PCAP for different workloads and disk drives. Our experiments show that PCAP reduces power by up to 22%. Further, under PCAP, 60% of the requests observe service times below 100 ms compared to just 10% under throttling. PCAP also reduces worst-case latency by 50% and increases throughput by 32% relative to throttling.

1 Introduction

The widespread adoption of on-line services has been fueling the demand for more and denser datacenters. The design of such warehouse-sized computing systems [12] is not at all trivial. Architects have to deal not only with computing, storage and networking equipment, but also with cooling and power infrastructures [13]. In fact, power and energy are first-order concerns for architects as new high-performing hardware is likely to require more power, while the cost of hardware has remained stable. With these trends continuing, Barroso [11] argues that the cost of the energy to operate a server during its lifetime could surpass the cost of the hardware itself.

Power is more concerning since the cost of building a datacenter is mainly dictated by the costs of its power infrastructure. These costs typically range between \$10 and \$20 per deployed Watt of peak critical power [29]. Hence, a datacenter with a provisioned 10 MW peak power capacity costs \$100M to \$200M (excluding cooling and ancillary costs), a significant amount of money. Contrast the \$10/W building cost with an average of \$0.80/Watt-hour for electricity in the U.S. Still, while energy costs vary with the usage of the datacenter, building costs are fixed and based on the *peak* power capacity. Consequently, it becomes very important to fully utilize a datacenter's power capacity. If a facility is operated at 85% of its maximum capacity, the cost of building the facility surpasses the energy costs for *ten years* of operation [29].

Recent studies have addressed maximizing the power utilization in datacenters [26, 36]. Researchers have characterized the power usage at different levels in the datacenter (e.g., machine and cluster) and investigated power provisioning strategies. One especially promising strategy is *power over-subscription* [12], that oversubscribes a datacenter with more machines (and thus more work) to ensure near 100% power utilization. The incentive to fully utilize the power budget is, however, offset by the risk of overloading the power trains and infrastructure of the datacenter. Such overloading could result in long service downtimes (due to power outages) and/or costly contractual fines (due to service agreement violations). To prevent overloading, power capping techniques are deployed as a safety mechanism, thus allowing maximum utilization while preventing costly penalties.

Power capping is a mechanism that ensures that the power drawn by a datacenter stays below a predefined power limit or cap. At the core of power capping is a monitoring loop, which takes in power readings, and computes the amount of power capping needed. Capping itself is done in a variety of techniques depending on the scale and type of the hardware component under question. On a datacenter level, capping is an aggregate number that trickles down to clusters, racks, machines and components. Suspending low-priority tasks in a cluster and adapting the clock frequency of a CPU component are two example techniques.

This work focuses on capping the power usage of the storage component of the datacenter. We address the 3.5-inch high-capacity enterprise hard disk drives (HDDs) common in today's cloud deployments. This paper tackles the question of: *How can the* HDD *power consumption be capped in a performance-aware manner?*

To this end, we revisit the throttling technique proposed for power capping [40] and address its shortcomings in a new technique we propose. We show that throttling *unnecessarily* sacrifices timing performance to cap power. Throttling limits disk throughput by *stopping servicing and delaying all outstanding requests*. It is predicated on the assumption that low throughputs result in less power usage by the disk. Our power measurements reveal that, contrary to the common belief, the power usage of the disk does not always increase with the increase in throughput but declines for high throughputs. We find *no strict positive correlation between the power usage and the throughput of the disk.*

To enable power capping for the disk drive, we propose a technique that resizes the I/O queues. We show that resizing queues not only reduces the impact on performance, unlike throttling, but also enables two different performance-oriented operation modes: throughput and tail-latency. This is important given the various services offered by today's datacenters. For instance, web search is sensitive to latency, whereas Map-reduce is throughput-oriented [22, 35]. By I/O queues we mean both the disk's queue as well as its respective OS queue. We investigate the interplay between both queues and their influence on the disk's power usage.

We present PCAP, an agile performance-aware power capping system for the disk drive. PCAP dynamically adapts the queues of a disk drive to cap its power. It performs power capping in two different operation modes: throughput and tail-latency. Under PCAP, 60% of the requests exhibit latencies less than 100 ms as opposed to just 10% under throttling. Also, PCAP reduces worst-case latency by 50% and increases throughput by 32% compared to throttling. PCAP has three goals:

1. Agile power capping that reacts quickly to workload variations to prevent power overshooting as well as performance degradation.
2. Graceful power control to prevent oscillations in power and better adhere to service level agreements.
3. Maximized disk performance for enhanced performance of applications.

This paper makes the following contributions:

- Revisiting the throttling technique for HDDs and studying the throughput–power relationship (section 4).
- Investigating the impact of the HDD's and OS queues on the HDD's power and performance (section 5).
- Designing and evaluating the PCAP system that is agile, graceful, and performance-aware (section 6).

This paper is structured as follows. The next section offers a brief refresher of the basics of HDDs. Section 3 evaluates the merit of power capping for HDDs and presents our experimental setup. Section 4 revisits throttling and its impact on power. Section 5 investigates the influence of the queue size on HDD's power consumption. Section 6 presents the design of PCAP and Section 7 evaluates it. Section 8 studies PCAP for different workloads. Section 9 discusses related work and Section 10 concludes.

2 Background

2.1 Power capping vs. Energy efficiency

This work focuses on power capping to maximize the utilization in the datacenter, where peak power predominates costs of ownership. We do not address energy efficiency, where machines are powered down in underutilized datacenters to save energy. While the latter received ample attention in the literature [43, 44, 45, 48, 41], the former received relatively little [40, 26, 36].

Addressing power capping, we measure power dissipation. Power is the rate at which electricity is consumed. It is measured at an instant in time as Watts (W). To contrast, energy is a total quantity and is power integrated over time. It is measured as Wh (Watt-hour), or joules. We focus on power usage here.

2.2 HDD power capping

The active read/write mode of the HDD is of a primary interest for power capping. This is because the HDD draws most of the power in the active mode (e.g., compare 11 W during activity to 6 W during idleness). Also, in cloud systems, HDDs spend most of the time in the active mode [17]. Generally speaking, power capping may transition the HDD between the active mode and one or more of its low power modes to reduce the average power drawn in a period of time. Throttling for instance transitions the disk between the active and idle modes. This transitioning comes at a performance penalty, which scales with the depth and frequency the low-power mode being visited. In contrast, in this work we avoid transitioning between power modes and propose the adjustment of the queue size to achieve power capping for the disk drive in the active mode.

2.3 HDD's IOQ and NCQ queues

Any block storage device, that is managed by an Operating System (OS), has a respective queue as a part of the OS [14]. The queue serves as space for the I/O scheduler to reorder I/O requests for increased throughputs. For example, the Linux OS maintains a queue depth of 128 requests by default (in the current Ubuntu distributions). Requests are reordered to optimize for sequentiality on the HDD. The queue size is adjustable with a minimum size of 4. We refer to this queue as IOQ in this work.

NCQ stands for Native Command Queuing [5]. It is an extension to the Serial ATA protocol that allows the HDD to internally optimize the order in which requests are serviced. For instance, the HDD may reorder requests depending on its rotational positioning in order to serve all of them with fewer rotations and thus less time. NCQ typically ranges from 1 to 32, where NCQ=1 means disabled NCQ.

2.4 Scope of this work

We focus on the storage component of the datacenter. Making storage power-aware enables better overall power provisioning and capping. The share in power consumption due to storage varies. For example, in storage-heavy systems, such as the HGST Active Archive System [3], 80% of the power is due to the 600 HDDs it hosts, whereas in a computing-heavy system, such as the Dell PowerEdge R720 Server, 5-10%. We propose a technique to cap power at the disk level. Other techniques exist that may operate at different levels. We envision our technique complementing other techniques to achieve datacenter-wide power capping. Our technique has potentially wide applicability, since it (1) has no influence on data availability, (2) works under heavy workloads (i.e., no idle periods), (3) has no impact on HDD reliability, and (4) enables fine-tuned Watt-order capping. It offers three key properties: sensitive to performance, non-invasive to the I/O software stack, and simple to understand and implement (see Section 6).

3 The case for HDD power capping

In this section, we address a merit question: *How much power can be saved by power capping the* HDD*?*

To this end, we quantify the range of power an HDD draws when servicing requests, called dynamic power. We discuss the setup we prepared for our studies first.

3.1 Hardware setup

We chose a JBOD setup (Just a Bunch of Disks) to host a set of HDDs, which are exercised and their power is measured. The JBOD setup consists of a Dell PowerEdge R720 Server connected via LSI 9207-8e HBA to a Supermicro JBOD. It holds 16 3.5" SATA HDDs. We selected HGST Ultrastar 7K4000 of 4 TB capacity [2], commonly found in cloud storage systems today.

Besides the HGST Ultrastar 7K4000, we have obtained a sample HDD from two other HDD vendors. We selected a Seagate 4TB HDD [6] and a WD 4TB HDD [8]. All disks have the same capacity and number of platters, since they share the same storage density (i.e., same generation). We use different disks to ensure the commonality of our observations as well as the applicability of our techniques across different vendors, generations and technologies.

We profile power using WattsUp .NET power meters [7]. We use one meter for the JBOD and another for the server. We interpose between the power supply and the mains. Since the JBOD is dual corded for high availability, we connect both to an electric strip which in turn goes into the power meter. The meters are connected via USB to the server, on which we collect power read-outs for later analysis. The meters sample power once per second. This rate should be enough for power capping, since capping is performed on higher time scales [12].

3.2 Software

Our server runs a 64-bit 12.02 Ubuntu Server distribution with the 3.0.8 Linux kernel. No OS data were stored on the disks in the JBODs. Instead, we used a separate disk for that purpose, which is housed in the server itself. Unless pointed out otherwise, the default settings of the I/O stack were kept intact. For example, the file systems on our disks (`XFS` in this case) were formatted with the default settings. The I/O scheduler was kept at the `deadline` default scheduler.

We used existing Linux utilities and implemented our own when needed. Utilities were used to generate workloads and collect logs of timing performance and power. We installed a WattsUp utility that communicates with the power meter and logs power read-outs. As for benchmarking, we use the FIO benchmark [10] to generate different workloads. We use FIO for design space exploration. To generate real workloads, we use MongoDB [4]. Because the usage of a benchmark varies depending on the goal of the experiment, we defer talking about the setup to each individual discussion of our studies. For timing performance profiling, we use the `iostat` Linux utility and benchmark-specific statistics. The collected performance and power logs are then fed

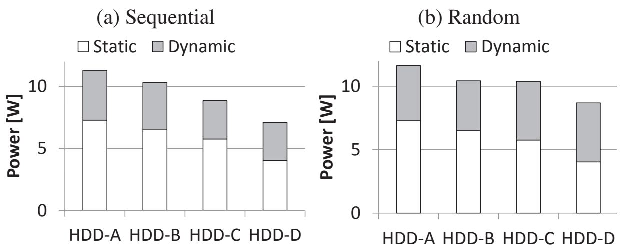


Figure 1: The dynamic and static power components measured for the four sample HDDs

to a custom-built Python program for analysis. Unless otherwise mentioned, we always collect measurements on the application level for end-to-end performance.

3.3 HDD's dynamic power

We measured the dynamic power for the four different types of HDD we have; the static power was isolated in separate measurements with no I/O load. Using FIO, we generated sequential and random workloads to measure the maximum power for each individual HDD. Under sequential workloads, the maximum power corresponds to the maximum attainable throughput, which is approximately 170 MB/s. In contrast, the maximum power under random workloads is attained by ensuring maximum seek distance between consecutive requests (Section 4). Figure 1 shows HDD's power broken down into static and dynamic components. The static component is related to the spindle and the electronics, whereas the dynamic component is related to the head arm and read/write channel. Across all HDDs, the figure shows that the dynamic power ranges up to 4 W and 5 W under sequential and random workloads, respectively. Hence, for our JBOD system, which can hold up to 24 HDDs, power capping exhibits a range up to 96 W and 120 W, respectively. As such, *HDDs enjoy a sizable dynamic power range for power capping to conserve power.*

4 HDD's throughput throttling

Throttling has been proposed as a technique to cap HDD's power [40]. This section investigates the relationship between the power and throughput in HDDs. We implemented a kernel module that enables us to throttle throughput under sequential as well as random workloads, called `dm-throttle`. The module is based on the device-mapper layer of the Linux kernel and is 700 lines of C code. It accepts as an input the desired throughput cap in KB per second or IOs per second for sequential and random workloads, respectively. The throughput cap can be modified at run-time via the `/proc/` file system, where statistics are accessed too.

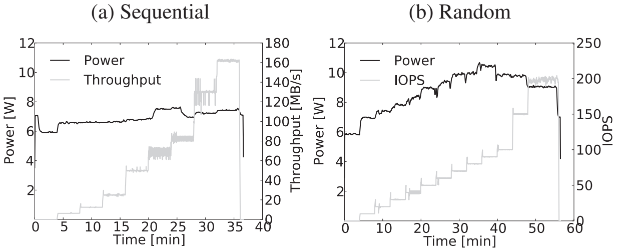


Figure 2: The power–throughput relationship under sequential and random workloads

We set up two instances of FIO to generate sequential and random workloads for 40 and 50 minutes, respectively. We used multiple threads in the random workload generation to attain maximum disk throughput. During the run of each FIO instance, we varied the throttling throughput of `dm-throttle` and measured the power and the effective throughput. Throughput was throttled at several points (6.25, 12.5, 25, 50, 75, 100, 150, 200 MB/s) for sequential workloads, and (10−100, 150, 200 IOPS) for random workloads. In these experiments, we used one HDD from the 16 (identical) HDDs. We singled out its power after subtracting the idle power of the JBOD (incl. power supplies, fans and adapters).

Figures 2a and 2b show the throughput–power relationship under sequential and random workloads, respectively. Figure 2a shows that the HDD draws more power (the black curve) as throughput increases (the gray curve). The HDD draws 8 W of power at the maximum throughput (170 MB/s). And its power range, that scales with the throughput, is 7 − 8 W. Another 0.6 W is added to the dynamic range due to channel activity when exercising the disk with some workload. This effect can be seen in Figure 2a during the second four minutes of the experiment. In separate measurements, we found that an additional 1 W of power is drained when the heads are loaded to counteract the drag. As such, the total dynamic range for such disks is 5.5 − 8 W, which is in agreement with the figures for HDD-C in Figure 1a.

Figure 2b shows that under random workloads power increases with the throughput up to a certain point, 90 IOPS in this case. After that, power starts decreasing for higher throughputs with a noticeable drop at the maximum throughput of 200 IOPS, thanks to better scheduling in the disk (see the next section). The figure highlights the fact that different throughputs can be attained for the same amount of power drawn. Also, the dynamic power range is wider under random workloads compared to sequential workloads, between 7 − 10.5 W versus 7 − 8 W (excluding the power due to channel activity and head positioning). As such the *effective* power ranges for our JBOD are up to 24 W and 84 W under sequential and random workloads, respectively.

Summarizing, we make two key observations that guide the rest of this work:

Observation 1: *HDDs exhibit a dynamic power range that is wider under random workloads compared to sequential workloads.*

Observation 2: *HDDs can offer different throughputs and service times for an equivalent amount of power under random workloads.*

The two observations lead us to investigate a power-capping technique under random workloads (Observation 1) that is performance-aware (Observation 2). We focus on random workloads in the rest of this paper. If required, power-capping by throttling should be a sufficient technique for sequential workloads, thanks to the positive correlation between power and throughput *under sequential workloads*. Next, we investigate the reason for the decline in the HDD's power consumption at high throughputs, which motivates the choice for using the queue size to control power.

5 Power–Queue relationship

This section explains the dynamics of the HDD's head arm. We motivate our choice for the queue size to control power. We then investigate the influence of the queue size on the HDD's power, throughput, and service time.

5.1 Causality

Under random workloads, the HDD's head moves across the surface of the platters to service requests from different locations. The head motion is characterized as random, since the head spends most of the time seeking instead of accessing bits (compare 5 ms seek time to 0.04 ms 4 KB read time). Moving the head dissipates power by its VCM (voice-coil motor). Depending on the physical distance separating any two consecutive requests, the head may need to accelerate and subsequently decelerate. Acceleration and deceleration require a relatively large amount of power (similar to accelerating a car from standstill). *A few (random) requests have a long physical distance in between, requiring acceleration and deceleration. Conversely, the more the requests dispatched to the disk, the shorter the separating distance and thus the less the power due to reduced acceleration, if any.* At higher loads more requests are dispatched to the disk simultaneously, allowing the disk to better schedule and reduce distances and thus accelerations resulting in less power. In Figure 2b, the disk consumes less power at low throughputs ($<$ 90 IOPS) too but for a different reason. At low throughputs, the disk is underutilized and spends more than 45% of the time in the idle power mode, resulting in power savings that outweigh the increase in power due to (occasional) accelerations.

5.2 Characterizing the relationship

This section studies both the IOQ (scheduler) queue and the NCQ queue described in Section 2.3. We investigate their interplay and influence on power and performance. We seek to answer the following two questions:

1. For a fixed NCQ queue size, what is the relationship between the IOQ queue size and the HDD's power, throughput and service time?

2. For a fixed IOQ queue size, what is the relationship between the NCQ queue size and the HDD's power, throughput and service time?

Methodology We studied a single HDD in our JBOD system from Section 3.1. We carried out two sets of experiments to confirm trends: once with FIO and another with MongoDB [4]. We generated random 4KB requests with FIO using `aiolib`. We used enough threads to mimic real systems with multiple outstanding requests. For the MongoDB setup, we stored 100 databases on the disk, each of which is approximately 10 GB in size. The files of every two consecutive databases (e.g., 1-st and 2-nd) were separated by a 10-GB dummy file on the disk, so that 2.4 TB was consumed from our 4 TB HDD. The disk was formatted with the `XFS` file system using the default settings. We used YCSB [19] to exercise 10 databases, the 10-th, 20-th, 30-th, up to the 100-th. One YCSB instance of 10 threads was used per database to increase throughput. We benchmarked for different queue sizes. The IOQ queue was varied in the range (4, 8, 16, 32, 64, 128), whereas the range for the NCQ queue was (1, 2, 4, 8, 16, 32). To resize a queue to a value, say SIZE, we used the following commands:

- IOQ queue: `echo SIZE > /sys/block/sdc/queue/nr_requests`
- NCQ queue: `hdparm -Q SIZE /dev/sdc`

Results Figure 3a shows HDD's power versus the size of the IOQ queue. Power decreases as the IOQ queue size increases. A large IOQ queue enables better scheduling and thus reduces randomness in requests arriving to the disk (which in turn reduces accelerations). A trend exists where power reduction exhibits diminishing returns at large queues, since only the power due to the head's VCM is affected, whereas other static power components remain intact (Amdahl's law). The figure confirms the same trend for different sizes of the NCQ queue, but at different power levels.

Figure 3b shows HDD's power versus the size of the NCQ queue. Unlike for the IOQ queue, two opposing trends exist. In fact, the size of the IOQ queue plays a major role here. We can explain the figure by observing trends at small and large IOQ queue sizes (e.g., 4 and

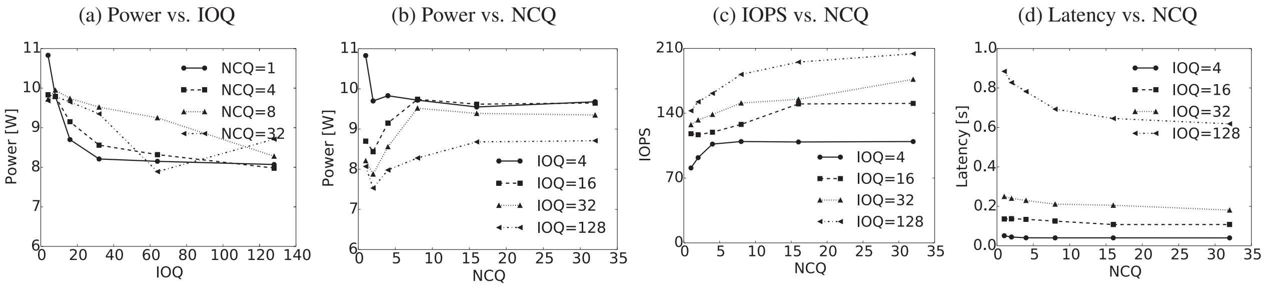


Figure 3: The influence of the queue size for both IOQ and NCQ on the HDD's power, throughput and service time

32, respectively). At small sizes, power decreases as the NCQ queue size increases, because the requests arriving at the disk sill exhibit randomness, leaving room for better scheduling by NCQ. Recall that better scheduling reduces acceleration and thus leads to lower power consumption. Conversely, at large sizes of the IOQ queue, power increases as the NCQ queue size increases, since randomness is already reduced by the I/O scheduler and thus even higher throughputs are attained at large NCQ queue sizes (200 IOPS versus 100 IOPS on the bottom curve of Figure 3c). High throughputs involve more channel activity, which draws more power.

As for the timing performance, Figure 3c shows the relationship between throughput, in IOPS, and the NCQ queue size. Expectedly, throughput increases at large queues, since more time is spent on accessing bits rather than seeking. We observed similar trends for throughput versus the IOQ queue. We do not show it for space reasons. One observation is that the HDD's throughput is mainly limited by the IOQ size. That is, increasing NCQ beyond IOQ size does not result in increased throughput, since NCQ scheduling is limited by the number of requests it sees at a time, which is in turn bounded by the IOQ size.

Figure 3d shows the relationship between the HDD service time, measured by `iostat`, and the NCQ queue size. Surprisingly, the service time decreases for large NCQ queue sizes, although larger queuing delays are incurred. This suggests that the saving in rotational positioning time due to NCQ scheduling outweighs the queuing delay of large queues. This improvement is more pronounced for large numbers of arriving requests as shown by the top curve in the figure for an IOQ size of 128. Conversely but expectedly, we observed a linear increase in service time as the IOQ queue size increases. We do not show it for space reasons.

Summarizing, HDD power decreases with the increase in the size of the IOQ (scheduler) queue. Both throughput and service time expectedly increase. On the other hand, while throughput increases with the increase of the NCQ queue size, power and service time have unusual trends. We make two new observations:

Observation 3: *The power drawn by the HDD exhibits opposing trends with respect to the NCQ queue size. These trends are influenced by the size of the IOQ scheduler queue.*

Observation 4: *The HDD's service time decreases with the increase in the size of the NCQ queue, thanks to improved in-disk scheduling.*

The interplay between the two queues leads to the following observation:

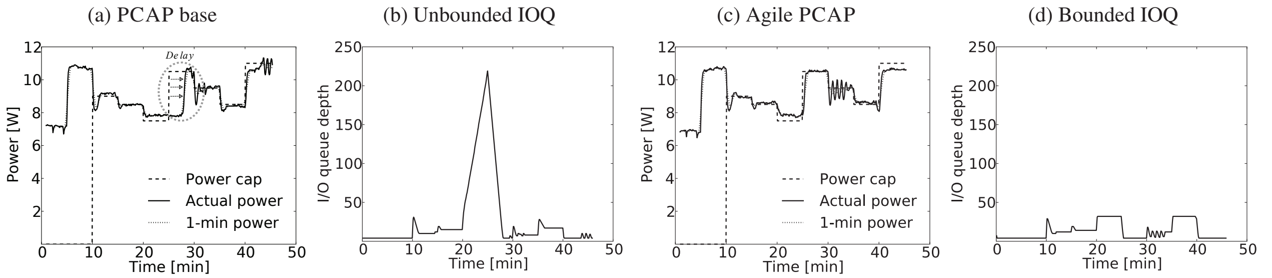
Observation 5: *The HDD's dynamic power range can be fully navigated by varying the sizes of the NCQ and I/O scheduler queues.*

6 PCAP design

In a dynamic datacenter environment, where power requirements and workloads change constantly, a control system is required. Such a system ensures compliance to power caps when power is constrained and enables better performance when more power is available. We present the design of PCAP and techniques to make it graceful, agile and performance-aware. PCAP's design is based on the observations made previously.

6.1 Base design

At its core, PCAP has a control loop that triggers every period, T. It computes the running average of the power readings over the past T time units and decides on the amount of power capping needed. PCAP is a proportional controller. To attain better performance and ensure stability, we improve upon the basic proportional controller in four ways. (1) PCAP increases and decreases power using models derived from the observations of Section 5. (2) It uses different gains when increasing and decreasing power. (3) PCAP bounds the ranges of its control signals (i.e., queue sizes) and (4) employs a hysteresis around its target output to prevent oscillations due to measurement errors. Power is increased or decreased by adjusting the size of the IOQ and NCQ queues gradually, one step per period. Queue adjustment is based on the relationships investigated in Section 5.2. PCAP uses two factors, α_{up} and α_{dn}, to increase power (or allow


(a) PCAP base

(b) Unbounded IOQ

(c) Agile PCAP

(d) Bounded IOQ

Figure 4: Power capping with PCAP and the corresponding queue sizes for the (a & b) base and (c & d) agile designs

better performance) and to decrease power, respectively. We use two factors, since the decrease in power must be done aggressively to avoid penalties, whereas the increase in power is done incrementally and cautiously. Consequently, α_{dn} is greater than α_{up}. Based on either factor, the queue size is adjusted proportionally to how far the target power, P_t, is from the current power, P_c. We use the following equations to calculate the change in the queue size, ΔQ, to enable *graceful control*:

$$\Delta Q_{IOQ} = \frac{|P_t - P_c|}{\Delta P_{IOQ}} \cdot 2\alpha_{dn} \quad (1)$$

$$\Delta Q_{NCQ} = \frac{|P_t - P_c|}{\Delta P_{NCQ}} \cdot \alpha_{dn} \quad (2)$$

ΔP_{IOQ} and ΔP_{NCQ} are the maximum change in power attainable by varying the IOQ and NCQ queues, respectively. We multiply α_{dn} by 2 for the IOQ to attain measurable changes in power. Changing queue sizes to allow for power increase follows Equations 1 and 2 after replacing α_{dn} with α_{up}. To account for measurement errors and reduce oscillations, an error margin of ε is tolerated around the target power. That is no further power-capping actions are taken, if the current power is within a range of [-ε, +ε] of the target power. For our experiments, we settled on the settings for PCAP's parameters shown in Table 1. These are, however, configurable depending on the conditions of the datacenter as well as its operation goals.

We implemented a prototype of PCAP in Python. It is 300 lines of code. PCAP runs in the background. New power targets, if any, are echoed into a text file which PCAP reads in at the beginning of every period, T. After that, it executes the control loop explained before until the target power is attained. Figure 4a shows the activity of PCAP on a single HDD over a 45-minute period of time. The figure shows three curves, the target power cap (dashed), the instantaneous power (solid), and the 1-min average power (dotted). We generate a random workload for the entire period. Initially, we leave the disk idle for 5 minutes and then generate a workload with no power-capping for another 5 minutes. The HDD draws approximately 8 W and 11 W, respectively. At minute 10, the power cap is set to 9 W and PCAP adjusts queues to reduce HDD's power by 2 W to below 9 W. At minute 15, the power cap is again lowered to 8.5 W and PCAP lowers power accordingly. At minute 20, PCAP is unable to lower the power below 7.5 W, since it is outside the dynamic range of the HDD. We keep capping power at different levels for the rest of the experiment and PCAP reacts accordingly. Figure 4b shows the change in the queue sizes to attain power-capping.

The figure shows a delay at minute 25 in responding to raising the power cap from 7.5 W to 10.5 W. We study this sluggishness in the next section. Datacenter operators may by more interested in long-term smoothed averages for which contractual agreements may be in place. For example, the 1-minute power curve in the figure inhibits oscillations, unlike the instantaneous power curve, so that power violations should be of no concern in this case. We discuss performance in Section 6.3.

Table 1: PCAP's parameters and their settings

Parameter	Setting	Description
T	5 s	control loop period
α_{up}	2	factor used for power increase
α_{dn}	8	factor used for power decrease
ε	0.2 W	control tolerance factor
ΔP_{IOQ}	2 W	max power change with IOQ queue
ΔP_{NCQ}	2 W	max power change with NCQ queue
$[Q^L_{IOQ}, Q^U_{IOQ}]$	$[4, 32]$	IOQ queue range
$[Q^L_{NCQ}, Q^U_{NCQ}]$	$[1, 8]$	NCQ queue range
Q^ρ_{IOQ}	128	IOQ setting for maximizing throughput
Q^ρ_{NCQ}	32	NCQ setting for maximizing throughput

6.2 Agile capping

The oval in Figure 4a highlights an inefficiency in the base design of PCAP. It manifests as delays in increasing power when the power cap is lifted up. This inefficiency results in low throughputs and long service times, since queues take some time to adjust accordingly as shown at minute 25, calling for better agility.

We examined the cause of the delay in adjusting queue sizes. Figure 4b shows that the IOQ queue reached high sizes during the tracking of the previous very low power target (7.5 W). This is because the base design keeps in-

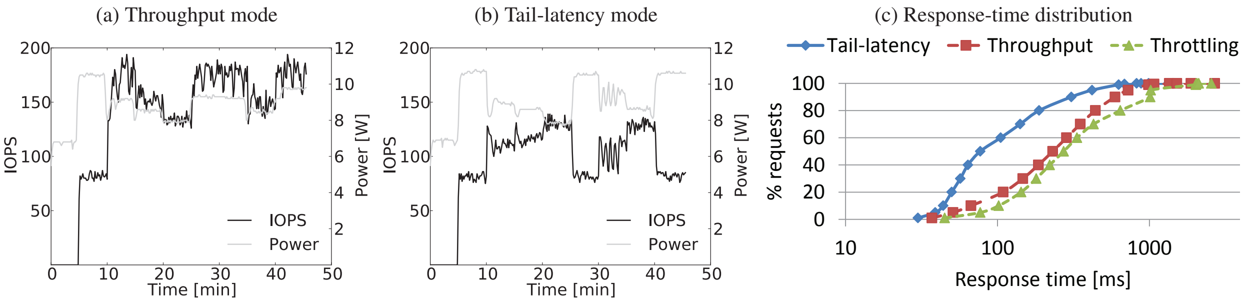


Figure 5: Throughput and service time under the throughput and tail-latency performance modes of PCAP

creasing the size of the IOQ queue until the power target is reached. In this particular case, IOQ queue size reached 219. Similarly, the NCQ queue can increase significantly. Later, when new and higher power targets are set as in minute 25, PCAP takes long time to reduce the queue size to low values since this happens gradually. The sluggishness results in a relatively long time period of lost performance.

To improve agility, we leverage an observation made from Figure 3a namely, power exhibits diminishing returns at high queue sizes, so that their respective power savings are marginal. As such, we introduce upper bounds on the sizes of both queues, Q^U_{IOQ} and Q^U_{NCQ}. The bounds limit the increase in queue sizes and enable shorter times to navigate queue ranges. Similarly, we introduce lower bounds. Figure 4c shows the performance of PCAP with bounds. Thanks to its queue-depth bounding, PCAP avoids infinitely and hopelessly attempting to cap power. Figure 4d confirms that the queue sizes never exceed the bounds. The bounds are shown in Table 1 and were set with values of the knees of the curves of Figures 3a and 3b. Figure 4c confirms *PCAP's agility*.

6.3 Performance awareness

Resizing queues impacts the performance of the HDD. We distinguish between two types of timing performance: (1) throughput and (2) tail-latency. In the throughput mode, PCAP attempts to increase throughput while adhering to the power cap. This mode enhances the average latency. In the tail-latency mode, PCAP attempts to reduce the high-percentiles latencies or alternatively shorten the tail of the latency distribution. In practice, the designer can set performance targets along the power target to reach compromises between the two.

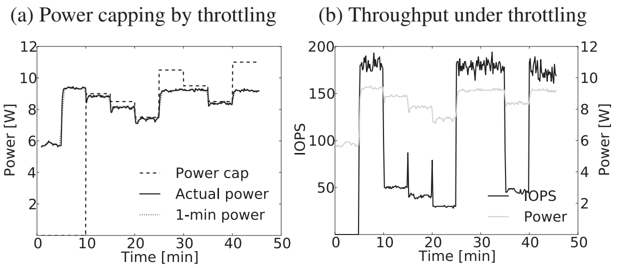
As discussed in Section 5.2, throughput increases by increasing the size of both IOQ and NCQ queues. Power decreases with the increase in the IOQ queue size, whereas it increases for large NCQ queue sizes as shown in Figure 3c. PCAP uses models of these relationships to increase throughput while capping power. In contrast, the tail-latency decreases (i.e., high-percentile latencies decrease) for small IOQ queue sizes and large NCQ queues as shown in Figure 3d. We also incorporate models of this relationships in PCAP to reduce tail latencies while capping power.

The solution for agility of the previous section is in conflict with maximizing throughput (in PCAP's throughput mode). This is because the low upper bound on the size of both queues limits the maximum attained throughput. Compare 150 IOPS at (NCQ=8, IOQ=32) to 200 IOPS at (NCQ=32, IOQ=128) in Figure 3c, a 25% potential loss in throughput. To prevent this loss, we redesigned PCAP such that when it reaches the "agility" upper bounds, it snaps to predefined queue settings to maximize throughput in the throughput mode. The corresponding queue parameters are Q^ρ_{IOQ} and Q^ρ_{NCQ} (see Table 1). Similarly, PCAP snaps back from these settings to the upper bounds in the downtrend. That way, agility is still preserved while throughput is maximized. This scheme has no effect in the tail-latency mode, since small queues are required.

Figures 5a and 5b show the throughput of a single HDD when running PCAP in the throughput and tail-latency modes, respectively. Throughputs up to 200 IOPS are attained in the throughput mode, which is higher than the 140-IOPS maximum throughput attained in the tail-latency mode. The high throughput comes at the cost of long response times, resulting in a long tail in the distribution of the response time as Figure 5c shows. The figure plots the corresponding cumulative distribution function for the response time measured at the client side for both PCAP's modes. Tail-latency attains shorter maximum latencies, compare 1.3 s to 2.7 s maximum latency. Further, 60% of the requests observe latencies shorter than 100 ms in the tail-latency mode as opposed to just 20% in the throughput mode. We discuss the curve corresponding to throttling in Section 7.1.

7 PCAP's performance

This section compares PCAP to throttling and then studies the influence on I/O concurrency on PCAP's ability to cap power.



(a) Power capping by throttling

(b) Throughput under throttling

Figure 6: Using throttling to cap (a) HDD's power usage, while (b) maximizing throughput

7.1 PCAP versus throttling

Both queue resizing and throughput throttling can be used for capping power as evidenced in Figures 5a and 2a, respectively. Section 5.1 presented a qualitative argument as for why to use queue resizing to control power. This section presents a quantitative argument. We compare the timing performance of the HDD when its power is capped by either queue resizing (i.e., PCAP) or throttling. Precisely, we answer the question of: *For the same power cap, how do the throughput and the service time compare under throttling and* PCAP*?*

We repeated the same experiments of Section 6.3 while throttling the throughput (see Figure 2a) to cap the power within the desired levels. We used our `dm-throttle` module for throttling to reproduce Figure 4c. To take advantage of the HDD's low power consumption at high throughputs (Figure 2a), we used throttling only for power caps strictly below 9.5 W (which corresponds to the power drained at maximum throughputs). For higher power caps, we disabled throttling to maximize the performance of the disk. As such, `dm-throttle` was used to cap power selectively while avoiding hurting performance whenever possible, hence offering the best-case performance of throttling.

Figure 6a shows the power consumption of the HDD. Throttling keeps the power below the power cap, which is shown in dashed black. The power curve separates from the power cap by a large margin during some periods, such as minutes 25–30 and 40–45. These periods correspond to power capping with no throttling deployed, since the power cap is above 9.5 W. Figure 6b shows the throughput of the disk over the time period of the experiment. It confirms maximum throughputs around 185 IOPS during periods of no throttling. Comparing Figure 6b to Figures 5a and 5b we can visually see that throttling attains lower throughputs than the throughput mode of PCAP, whereas it outperforms the tail-latency mode of PCAP. While throttling attains an average of 117 IOPS, PCAP attains 154 IOPS (32% more) and 102 IOPS (15% less) in the throughput and tail-latency modes, respectively. Figure 5c shows the cumulative distribution of the response time for the requests of the previous experiment. Overall, throttling attains worse latency than the two performance modes of PCAP. Just 10% of the requests exhibit latencies under 100 ms, whereas the maximum latency is 2.5 s. This is however expected, since throttling delays requests and maintains default queue settings that are deep and optimized towards throughput. By treating response time and throughput differently, PCAP outperforms throttling on both performance goals. 60% of the requests exhibit response times below 100 ms and an increase of 32% in throughput is attainable with PCAP.

PCAP relies on concurrency in I/O requests to attain capping as we shall see next.

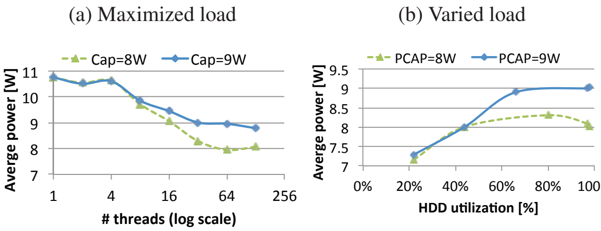
7.2 Influence of concurrency on PCAP

We carried out two sets of experiments to study PCAP's performance on a single HDD. In the first set, we varied the number of concurrent threads, while maximizing the load per thread so that the HDD utilization is 100% all the time. In the second set, we fixed the number of threads and varied the load per thread to attain different utilization levels.

Our goal is to study the relationship between the effective queue size (i.e., the actual number of outstanding requests at any time instance) and PCAP ability to cap the HDD's power. In the first experiment, since utilization is maximized the effective queue size matches the number of threads. In the second experiment, the effective queue size varies, since the load varies.

Maximized load In the first experiment, we varied the number of threads in the range (1, 2, 4, 8, 16, 32, 64, 128). We ran PCAP in the throughput mode and in the tail-latency mode, and chose different power caps. Figure 7a shows the average power measured over the period of the experiment for each setting. The figure shows that for a few threads the average power is 10.7 W, far above the power targets (8 W and 9 W). That is PCAP cannot cap power at concurrency levels below 4 threads (i.e., queue sizes under 4), since little opportunity exists for reordering requests and saving power. At concurrency levels above 4, the two curves start declining and departing from each other. This signifies that PCAP starts achieving some capping but cannot attain the target cap (i.e., no perfect capping). Finally, it attains the 9 W cap at 32 threads, whereas the 8 W cap is attained at 64.

We repeated the same experiment while running PCAP in the tail-latency mode. We studied for three power targets including 10 W, since the HDD enjoys larger dynamic range for small queues. We observed the same trend of attaining capping at higher concurrency levels. The power caps were attained at smaller number of threads compared to the case with the throughput


(a) Maximized load

(b) Varied load

Figure 7: The influence of I/O concurrency on PCAP's performance in capping power for a single HDD

mode. For example, 9 W and 8 W were attained with as little as 16 (vs. 32) and 32 (vs. 64) threads, respectively. In addition, for the higher power target of 10 W, PCAP attains the cap starting from 2 threads.

Varied load In this experiment, we pushed load to the HDD progressively while fixing the number of threads. We used 32 threads to ensure power capping with PCAP. Without capping, the load varies between 32 IOPS to 200 IOPS. Figure 7b plots the average measured power versus the utilization level reported by `iostat`. The figure shows that the power is lower than both targets at low utilization, since the HDD spends a sizable fraction of time idling, thanks to the low load. Observe in such a scenario, throttling is happening "naturally". At higher utilizations, PCAP perfectly caps the power to below 9 W, whereas the 8 W cap is violated for utilization levels between 40–95%. To explain these results, we examined the effective queue size (from `iostat`) versus utilization. We found that the overshooting over 8 W is due to queue sizes below 16, where PCAP cannot achieve perfect capping (see Figure 7a). At higher utilization levels, large queues build up which allows PCAP to restore power to below (or close to) the target as shown for 9 W.

Discussion In summary, PCAP attains perfect power capping for high concurrency levels, whereas reductions in power usage are possible for lower levels. The effective concurrency for perfect capping tends to increase as the desired power cap decreases. We find that PCAP becomes effective starting from 32 threads. We also find that PCAP is ineffective below 4. We believe this should not be an immediate concern to the applicability of PCAP, since real systems are usually multi-threaded for performance reasons. As such, chances that little concurrency appears in practice are little. That said, throttling can be used in such cases.

8 Putting it all together

This section evaluates PCAP's performance under different workloads. Then, we evaluate it when capping power at the system level for an array of HDDs.

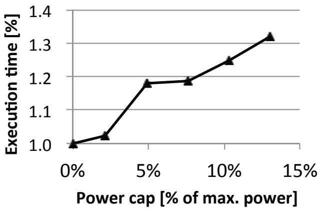


Figure 8: The increase in execution time for a 500 MB batch job for different power caps

8.1 PCAP under batch workloads

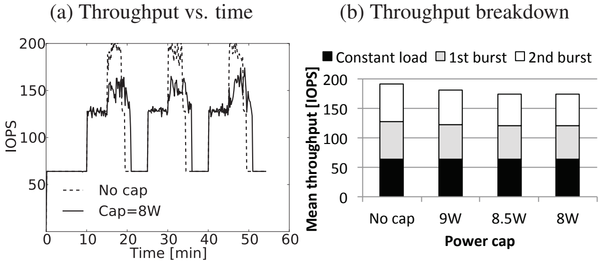
This section studies the impact of power capping by PCAP on the performance. We assume a batch job that reads data chunks from random locations on a single HDD. The HDD always reads at its maximum throughput and reads a total of 500 MB. We investigate the total execution time of the job when capping the HDD power at different levels. Since the job is batch, where throughput is important, we run PCAP in the throughput mode. And we vary the power cap in the range [8-9] W with a step of 0.25 W. We study the tail-latency mode later.

Figure 8 plots the execution time of the batch job versus the power cap. We normalize the figure to the case without power capping, where the average power is 9.2 W and the total execution time is 10.5 minutes. The figure shows that the execution time increases by 33% when capping at 13% (8 W), which is the maximum attainable cap. Also, the figure highlights a nonlinear relationship between the power cap and performance. For example, the increase in response time between power caps 2% and 5% is larger than that between 5% and 8% (16% vs. 2%). We confirmed this relationship by repeating the experiments and also examining the effective throughput of the application. We found that the throughput is 195 IOPS at 2% and drops to 167 IOPS and 165 IOPS at 5% and 8%, respectively. We study for bursty workloads next.

8.2 PCAP under bursty workloads

This section studies PCAP under workloads that vary in intensity and exhibit trough as well as bursty periods. This experiment seeks primarily to show that the increase in a job's execution time due to power capping (as shown in the previous section), can be absorbed in the subsequent trough periods. As such, long execution times do not necessarily always manifest. Still, power is capped.

We searched for real traces to replay but were challenged with two issues. The first issues was scaling the address space to reflect the growth in disk capacity. The second issue was scaling the arrival times. We obtained Microsoft traces [37] and implemented a replay tool. Although these traces confirm the randomness in the I/O


(a) Throughput vs. time

(b) Throughput breakdown

Figure 9: (a) PCAP performance under bursty workloads. The increase in execution time is absorbed, thanks to the trough periods. (b) Throughput is reduced due to capping and workload components share the cut.

pattern, we found that the address space is in the range of 256 GB (vs. our 4 TB HDDs). More importantly, we found that the traces were taken from a mostly-idle system, so that high power is not an issue but energy efficiency (see Section 2.1) for which indeed some techniques were proposed such as write offloading [37]. For example, we needed to scale traces by a factor up to 100 in order to see I/O activity, which in turn was not realistic because the inherent burstiness of the trace disappeared. We resorted to emulating a real workload.

Our workload is 55-minute long and consists of three parts. The first part is a constant part which continuously pushes load to the underlying HDD at a rate of 64 IOPS. The second and third parts are bursts which read 40 MB and 20 MB worth of random data, respectively. They are repeated 3 times throughout the workload separated by trough periods, each is 10-minute long. The third burst always starts 5 minutes after the second one. We repeated the experiment 4 times for different power caps: no cap, 9 W, 8.5 W, and 8 W. PCAP runs in the throughput mode.

Figure 9a shows the throughput in IOPS for the entire experiment. We show for the two extreme power settings: no cap and 8.0 W to keep the figure readable. Two observations can be made. First, the throughput during bursts decreases for power-capped scenarios, whereas it remains unaffected otherwise since the actual power is below the cap. The reduced throughput results in longer times to finish the burst jobs, which are perfectly absorbed in the subsequent trough periods. Secondly, both capping scenarios finish at the exact time of 55 minute. Note that in cases were no trough periods exist, longer execution times cannot be hidden and the discussion reduces to that of the previous section.

Figure 9b shows the split of the HDD's throughput across the three workload components. We show the split for the four power-capping settings. The throughput drops from 190 IOPS to 170 IOPS. The two bursty parts share the cut in throughput, 11% and 17%, respectively.

In summary, power capping impacts the performance of the HDD. In real scenarios, where the total execution time matters, a system like PCAP can incorporate performance as a target to optimize for while performing power capping. The resultant increase in response time manifests in high intensity workloads such as batch jobs, whereas it decreases in varying workloads.

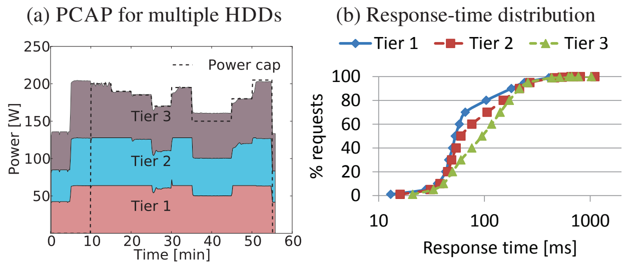
8.3 PCAP/S: System-level power capping

This section demonstrates the application of power capping on larger scales with multiple HDDs. We present PCAP/S, PCAP for a system of HDDs.

PCAP/S builds on PCAP. It borrows the main control loop of PCAP and works with multiple HDDs. PCAP/S elects HDDs for power capping in a performance-aware way. To mimic real systems, we assume that HDDs are split into tiers, which represent different service level objectives (SLOs) as in service-oriented datacenters [35, 1] for instance. PCAP/S' power capping policy splits into two parts: (1) JBOD-level and (2) HDD-level. We chose for a priority-based strategy for the JBOD-level policy. It elects HDDs from the lower-priority tiers first when power must be reduced, whereas higher-priority tiers are elected first when more power is available. HDDs within the same tier are treated equally with regards to power increase or decrease. This policy strives to reduce performance penalty for higher-priority tiers. The HDD-level part, on the other hand, is exactly that of Section 6, which resizes queues to attain power capping per HDD.

We studied power capping for the array of HDDs for our JBOD from Section 3.1. We split the 16 HDDs in each JBOD into three different tiers. The first tier enjoys the best performance. Both Tier 1 and Tier 2 contain 5 HDDs each, whereas Tier 3 has 6. The JBOD itself consumes 80 W of power when unloaded (i.e., contains no HDDs). We set PCAP/S' period, $T = 10$ s and error margin, $\varepsilon = 1$ W.

We applied random workloads generated by FIO to the three tiers over a 55-minute period of time. Different power caps were used: (200, 190, 185, 170, 195, 150, 150, 180, 205 W.) PCAP/S was run in the latency mode. Figure 10a shows the total power differentiated into three tiers. Power capping starts at minute 10 with a power cap of 200 W (excl. the static power). PCAP/S reduces the power below the cap by reducing the consumption of Tier 3, the lowest-priority tier. At minute 20, however, the power cap is set at 185 W, which is larger than the maximum saving attainable by Tier 3. Therefore, Tier 2 takes a cut in power here too, but at a lesser degree than Tier 3. At minute 25, the power cap is set at 170 W, and Tiers 1–3 contribute to the reduction. When power is raised later at minute 30, Tier 1 regains its maximum power budget, whereas Tier 3 still observes a cut in power. At minute 35, a relatively very low power cap of 150 W is set, which is beyond the capping capability



(a) PCAP for multiple HDDs

(b) Response-time distribution

Figure 10: Using PCAP/S to (a) cap the power usage of our JBOD for (b) three tiers of performance

of PCAP/S. Here, PCAP/S does its best by maximizing the reduction on all the tiers, while being off by 10 W. As such, it caps power by up to 22% in this particular case.

Figure 10b shows the distribution of the service time for the three workloads applied on the three tiers, respectively. It shows that 80% of the requests observe latencies less than 104, 151, and 172 milliseconds on the three tiers, respectively. Also, 80%, 70% and 55% observe latencies under 100 ms, respectively. Thanks to the priority-based JBOD-level policy, higher-priority tiers suffer less performance penalties.

8.4 Discussion

PCAP works under mixed workloads as well as under random workloads observing the fact that mixed workloads result in a random pattern on the disk. The capping percentage will be affected as detailed in the experiments in Section 7.2. As for pure sequential workloads throttling can be used. Extending PCAP to detect sequential patterns and use our `dm-throttle` from Section 4 should be straightforward.

Queue resizing by PCAP is based on observations inherent to the fundamental architecture of the HDD (Section 5.1) as opposed to specifics of an HDD model or brand. Therefore, we expect that no per-device calibration is needed but perhaps per family of devices; regular versus Helium HDDs. Figure 1 confirms the similarity of the dynamic power range for contemporary HDDs from different vendors.

9 Related Work

Numerous studies have addressed the power and energy efficiency of IT systems. Some studies focus on mobile systems [32, 33, 46], while others focus on datacenters. The latter studies look into energy efficiency as well as power controllability and accountability. Energy-efficiency studies closely examine power management approaches for processors [35, 39, 25], memory [23, 25] and the disk drive. Approaches for the disk drive include power cycling [20, 24, 45, 48, 15], dynamic RPM [27, 15], buffering and scheduling [18, 31], and the acoustic seek mode [16]. Other studies addressed modeling the energy consumption of the disk drive for holistic system efficiency [28, 9, 46, 47]. Newer technologies were also investigated. Researchers looked into newer technologies, such as SSDs to reduce data movement costs using their energy-efficient computation [42].

Recently, power has received increased attention in an attempt to reduce running and infrastructure costs [36, 26, 38, 34, 30]. Some authors investigated power accounting on a per-virtual machine [30, 38, 41]. Other authors proposed techniques for power capping for the processor [34, 33] and the main memory [21]. As for the disk drive, the throughput-throttling technique [40] and the acoustic seek mode [16] were proposed. While throttling works under sequential workloads, it incurs large performance penalties for random workloads. Likewise, acoustic seeks result in slow seeks, which impacts performance too.

This work complements the previous work and propose queue resizing to cap the disk drive's power consumption under random and mixed workloads. We investigated the relationship between the queue size and the power usage of the disk drive. We showed that queues can be resized to cap power yet in a performance-aware manner. We designed PCAP based on key observations of the queue–power relationship. PCAP is capable of capping for single- and multi-HDD systems. We made the case for PCAP's superiority over throttling.

10 Summary

We presented a technique to cap the power usage of 3.5-inch disk drives. The technique is based on queue resizing. We presented PCAP, an agile system to cap power for the disk drive. We evaluated PCAP performance on a system of 16 disks. We showed that PCAP outperforms throttling. In our experiments, 60% of the requests exhibit response times below 100 ms and an increase of 32% in throughput is attainable with PCAP. We also showed that PCAP caps power for tiered storage systems and offers performance-differentiation on larger scales.

Acknowledgments

The authors wish to thank Adam Manzanares for his thoughtful comments on early drafts of this paper. We also thank Jim Donaldson for helping with setting up the measurement apparatus. Our shepherd, Ken Salem, and the anonymous FAST reviewers helped improving the clarity of the manuscript with their detailed comments.

References

[1] Amazon S3. http://aws.amazon.com/s3/.

[2] HGST 3.5-inch Enterprise Hard Drive. http://www.hgst.com/hard-drives/enterprise-hard-drives/enterprise-sata-drives/ultrastar-7k4000.

[3] Hgst active archive system. http://www.hgst.com/products/systems/hgst-active-archive-system.

[4] mongoDB. http://www.mongodb.org/.

[5] Native Command Queuing. https://www.sata-io.org/native-command-queuing.

[6] Seagate Enterprise Capacity 3.5 HDD. http://www.seagate.com/internal-hard-drives/enterprise-hard-drives/hdd/enterprise-capacity-3-5-hdd/.

[7] Wattsup .Net power meter. https://www.wattsupmeters.com/secure/products.php?pn=0&wai=0&more=2.

[8] WD Enterprise Re 3.5 HDD. http://www.wdc.com/en/products/products.aspx?id=580.

[9] ALLALOUF, M., ARBITMAN, Y., FACTOR, M., KAT, R. I., METH, K., AND NAOR, D. Storage modeling for power estimation. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (New York, NY, USA, 2009), SYSTOR'09, ACM, pp. 3:1–3:10.

[10] AXBOE, J. FIO benchmark. http://freecode.com/projects/fio.

[11] BARROSO, L. A. The price of performance. *ACM Queue 3*, 7 (Sept. 2005), 48–53.

[12] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition*, vol. 8. 2013.

[13] BARROSO, L. A., DEAN, J., AND HÖLZLE, U. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro 23*, 2 (Mar. 2003), 22–28.

[14] BOVET, D., AND CESATI, M. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[15] CARRERA, E. V., PINHEIRO, E., AND BIANCHINI, R. Conserving disk energy in network servers. In *Proceedings of the 17th Annual International Conference on Supercomputing* (New York, NY, USA, 2003), ICS'03, ACM, pp. 86–97.

[16] CHEN, D., GOLDBERG, G., KAHN, R., KAT, R. I., AND METH, K. Leveraging disk drive acoustic modes for power management. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies* (Washington, DC, USA, 2010), MSST'10, IEEE Computer Society, pp. 1–9.

[17] CHEN, Y., ALSPAUGH, S., BORTHAKUR, D., AND KATZ, R. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *Proceedings of the 7th ACM european conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 43–56.

[18] CHOI, J., WON, Y., AND NAM, S. W. Power conscious disk scheduling for multimedia data retrieval. In *Proceedings of the 2nd International Conference on Advances in Information Systems* (2002), ADVIS'02, pp. 336–345.

[19] 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* (New York, NY, USA, 2010), SoCC'10, ACM, pp. 143–154.

[20] CRAVEN, M., AND AMER, A. Predictive Reduction of Power and Latency (PuRPLe). In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies* (2005), MSST'05, pp. 237–244.

[21] DAVID, H., GORBATOV, E., HANEBUTTE, U. R., KHANNA, R., AND LE, C. RAPL: Memory power estimation and capping. In *Proceedings of 2010 ACM/IEEE International Symposium on Low-Power Electronics and Design* (Aug 2010), ISLPED'10, pp. 189–194.

[22] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Communications of the ACM 56* (2013), 74–80.

[23] DENG, Q., MEISNER, D., RAMOS, L., WENISCH, T. F., AND BIANCHINI, R. Memscale: Active low-power modes for main memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 225–238.

[24] DOUGLIS, F., KRISHNAN, P., AND MARSH, B. Thwarting the power-hungry disk. In *Proceedings of the USENIX Winter 1994 Technical Conference* (Berkeley, CA, USA, 1994), WTEC'94, USENIX Association, pp. 293–306.

[25] FAN, X., ELLIS, C. S., AND LEBECK, A. R. The synergy between power-aware memory systems and processor voltage scaling. In *Proceedings of the Third International Conference on Power - Aware Computer Systems* (Berlin, Heidelberg, 2004), PACS'03, Springer-Verlag, pp. 164–179.

[26] FAN, X., WEBER, W.-D., AND BARROSO, L. A. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2007), ISCA'07, ACM, pp. 13–23.

[27] GURUMURTHI, S., SIVASUBRAMANIAM, A., KANDEMIR, M., AND FRANKE, H. DRPM: dynamic speed control for power management in server class disks. In *Proceedings of the 30th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2003), ISCA'03, ACM, pp. 169–181.

[28] HYLICK, A., SOHAN, R., RICE, A., AND JONES, B. An analysis of hard drive energy consumption. In *Proceedings of the 16th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS'08* (2008), IEEE Computer Society, pp. 103–112.

[29] IV, W. T., SEADER, J., AND BRILL, K. Tier classifications define site infrastructure performance. *The Uptime Institute, White Paper* (2006).

[30] KANSAL, A., ZHAO, F., LIU, J., KOTHARI, N., AND BHATTACHARYA, A. A. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC'10, ACM, pp. 39–50.

[31] KHATIB, M. G., VAN DER ZWAAG, B. J., HARTEL, P. H., AND SMIT, G. J. M. Interposing Flash between Disk and DRAM to Save Energy for Streaming Workloads. In *IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia, 2007. ESTIMedia 2007* (Oct 2007), pp. 7–12.

[32] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 209–222.

[33] LI, J., BADAM, A., CHANDRA, R., SWANSON, S., WORTHINGTON, B., AND ZHANG, Q. On the Energy Overhead of Mobile Storage Systems. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, 2014), FAST'14, USENIX Association, pp. 105–118.

[34] Lim, H., Kansal, A., and Liu, J. Power budgeting for virtualized data centers. In *Proceedings of the 2011 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2011), ATC'11, USENIX Association, pp. 59–72.

[35] Lo, D., Cheng, L., Govindaraju, R., Barroso, L. A., and Kozyrakis, C. Towards Energy Proportionality for Large-scale Latency-critical Workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA'14, IEEE Press, pp. 301–312.

[36] Meisner, D., Sadler, C. M., Barroso, L. A., Weber, W.-D., and Wenisch, T. F. Power management of online data-intensive services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2011), ISCA'11, ACM, pp. 319–330.

[37] Narayanan, D., Donnelly, A., and Rowstron, A. Write off-loading: Practical power management for enterprise storage. *Trans. Storage 4*, 3 (Nov. 2008), 10:1–10:23.

[38] Raghavendra, R., Ranganathan, P., Talwar, V., Wang, Z., and Zhu, X. No "power" struggles: Coordinated multi-level power management for the data center. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 48–59.

[39] Rajamani, K., Lefurgy, C., Ghiasi, S., Rubio, J., Hanson, H., and Keller, T. Power management solutions for computer systems and datacenters. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design* (Aug 2008), ISLPED'08, pp. 135–136.

[40] Stoess, J., Lang, C., and Bellosa, F. Energy management for hypervisor-based virtual machines. In *Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2007), ATC'07, USENIX Association, pp. 1:1–1:14.

[41] Thereska, E., Donnelly, A., and Narayanan, D. Sierra: practical power-proportionality for data center storage. In *Proceedings of the sixth conference on Computer systems* (New York, NY, USA, 2011), EuroSys'11, ACM, pp. 169–182.

[42] Tiwari, D., Vazhkudai, S. S., Kim, Y., Ma, X., Boboila, S., and Desnoyers, P. J. Reducing Data Movement Costs Using Energy-Efficient, Active Computation on SSD. In *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems* (Berkeley, CA, 2012), USENIX.

[43] Verma, A., Koller, R., Useche, L., and Rangaswami, R. SRCMap: energy proportional storage using dynamic consolidation. In *Proceedings of the 8th USENIX conference on File and storage technologies* (Berkeley, CA, USA, 2010), FAST'10, USENIX Association, pp. 267–280.

[44] Weddle, C., Oldham, M., Qian, J., Wang, A.-I. A., Reiher, P., and Kuenning, G. PARAID: A gear-shifting power-aware RAID. *Trans. Storage 3*, 3 (Oct 2007).

[45] Xu, L., Cipar, J., Krevat, E., Tumanov, A., Gupta, N., Kozuch, M. A., and Ganger, G. R. SpringFS: Bridging Agility and Performance in Elastic Distributed Storage. In *Proceedings of the 12th USENIX conference on File and storage technologies* (Berkeley, CA, USA, 2014), FAST'14, USENIX Association, pp. 243–256.

[46] Zedlewski, J., Sobti, S., Garg, N., Zheng, F., Krishnamurthy, A., and Wang, R. Modeling hard-disk power consumption. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), FAST'03, USENIX Association, pp. 217–230.

[47] Zhang, Y., Gurumurthi, S., and Stan, M. R. SODA: Sensitivity Based Optimization of Disk Architecture. In *Proceedings of the 44th Annual Design Automation Conference* (New York, NY, USA, 2007), DAC'07, ACM, pp. 865–870.

[48] Zhu, Q., Chen, Z., Tan, L., Zhou, Y., Keeton, K., and Wilkes, J. Hibernator: helping disk arrays sleep through the winter. In *Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), SOSP'05, ACM, pp. 177–190.

Mitigating Sync Amplification for Copy-on-write Virtual Disk

Qingshu Chen, Liang Liang, Yubin Xia‡, Haibo Chen‡*, Hyunsoo Kim†
Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
† Samsung Electronics Co., Ltd.

Abstract

Copy-on-write virtual disks (e.g., *qcow2* images) provide many useful features like snapshot, de-duplication, and full-disk encryption. However, our study uncovers that they introduce additional metadata for block organization and notably more disk sync operations (e.g., more than 3X for *qcow2* and 4X for *VMDK* images). To mitigate such sync amplification, we propose three optimizations, namely *per virtual disk internal journaling*, *dual-mode journaling*, and *adaptive-preallocation*, which eliminate the extra sync operations while preserving those features in a consistent way. Our evaluation shows that the three optimizations result in up to 110% performance speedup for *varmail* and 50% for *TPCC*.

1 Introduction

One major benefit of virtualization in the cloud environment is the convenience of using image files as virtual disks for virtual machines. For example, by using the copy-on-write (CoW) feature provided by virtual disks in the *qcow2* format, a cloud administrator can provide an image file as a read-only *base file*, and then overlay small files atop the *base file* for virtual machines [16]. This can significantly ease tasks like VM deployment, backup, and snapshot, and bring features such as image size growing, data de-duplication [7, 9], and full-disk encryption. Thus, CoW virtual disks have been widely used in major cloud infrastructures like OpenStack.

However, the convenience also comes at a cost. We observe that with such features being enabled, the performance of some I/O intensive workloads may degrade notably. For example, running *varmail* on virtual disks with the *qcow2* format only gets half the throughput of running on the *raw* formats. Our analysis reveals that the major reason is a dramatic increase of sync operations (i.e., *sync amplification*) under *qcow2*, which is more than 3X compared to the *raw* format.

The extra sync operations are used to keep the consistency of the virtual disk. A CoW image (e.g., *qcow2*) contains much additional metadata for block organization, such as the mapping from virtual block numbers to physical block numbers, which should be kept consistent to prevent data loss or even disk corruption. Thus, the *qcow2* manager heavily uses the *fdatasync* system call to ensure the order of disk writes. This, however, causes notable performance slowdown as a sync operation is expensive [17, 1]. Further, since a sync operation triggers disk flushes that force all data in the write cache to be written to the disk [25], it reduces the effectiveness of the write cache in I/O scheduling and write absorption. For SSD, sync operations can result in additional writes and subsequent garbage collection. Our evaluation shows that SSD has a 76% performance speedup for random write workload after disabling write cache flushing. A workload with frequent syncs may also interfere with other concurrent tasks. Our experiment shows that sequential writes suffer from 54.5% performance slowdown if another application calls *fdatasync* every 10 milliseconds. Besides, we found that other virtual disk formats like VMDK share similar *sync amplification* issues.

One way to mitigate the *sync amplification* problem is disabling the sync operations. For example, the VirtualBox (version 4.3.10) just ignores the guest sync requests for high performance [23]. Besides, VMware Workstation (version 11) provides an option to enable write cache [24], which ignores guest sync operations as well. This, however, is at the risk of data inconsistency or even corruption upon crashes [6].

To enjoy the rich features of CoW virtual disks with low overhead, this paper describes three optimizations, *per virtual disk internal journaling*, *dual-mode journaling* and *adaptive preallocation*, to mitigate *sync amplification* while preserving metadata consistency.

Per virtual disk journaling leverages the journaling mechanism to guarantee the consistency of virtual disks. *Qcow2* requires multiple syncs to enforce ordering, which is too strong according to our observation. To address this issue, we implement an internal journal for each virtual disk, where metadata/data updates are first logged in a transaction, which needs only one sync operation to put them to disk consistently. Such a journaling mechanism, however, requires data to be written twice, which is a waste of disk bandwidth. We further introduce *dual-mode journaling* which monitors each modification to the virtual disk and only logs metadata (i.e., *reference table*, *lookup table*) when there is no data overwriting.

[*‡] Corresponding authors

Adaptive preallocation allocates extra blocks for a virtual disk image when the disk is growing. The preallocated blocks can be used directly in the following expansion of virtual disks. This saves the image manager from requesting the host file system for more free blocks, and thus avoids extra flush operations.

We have implemented the optimizations for *qcow2* in QEMU-2.1.2. Our optimizations result in up to 50% performance speedup for *varmail* and 30% speedup for *tpcc* for a mixture of different workloads. When we run *varmail* and *tpcc* with a random workload, they can achieve 110% and 50% speedup, respectively.

2 Background and Motivation

We use the *qcow2* format as an example to describe the organization of a VM image and the causes of *sync amplification*.

2.1 The *qcow2* Format

A *qcow2* virtual disk contains an image header, a two-level *lookup table*, a *reference table*, and data clusters, as shown in Figure 1. The image header resembles the superblock of a file system, which contains the basic information of the image file such as the base address of the *lookup table* and the *reference table*. The image file is organized at the granularity of *cluster*, and the size of the cluster is stored in the image header. The *lookup table* is used for address translation. A virtual block address (VBA) a in the guest VM is split into three parts, i.e., a=(a_1, a_2, a_3): a_1 is used as the L1 table's index to locate the corresponding L2 table; a_2 is used as the L2 table's index to locate the corresponding data cluster; a_3 is the offset in the data cluster. The *reference table* is used to track each cluster used by snapshots. The refcount in *reference table* is set to 1 for a newly allocated cluster, and its value grows when more snapshots use the cluster.

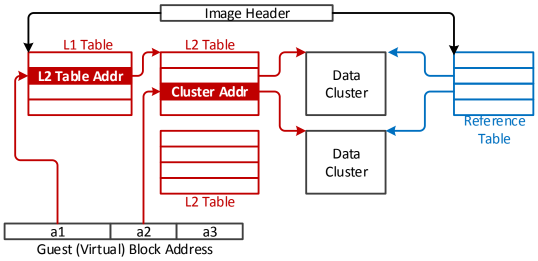


Figure 1: The orginization of *qcow2* disk format.

The process of writing some new data to a virtual disk includes following steps:

① Look up the L1 table to get the offset of the L2 table. ② If the L2 table is not allocated, then set the corresponding *reference table* entry to allocate a cluster for the L2 table, and initialize the new L2 table. ③ Update the L1 table entry to point to the new L2 table if a new L2 table is allocated. ④ Set the *reference table* to allocate a cluster for data. ⑤ Write the data to the new data cluster. ⑥ Update the L2 table entry to point to the new data cluster.

Note that, each step in the whole appending process should not be reordered; otherwise, it may cause metadata inconsistency.

2.2 Sync Amplification

The organization of *qcow2* format requires extra efforts to retain crash consistency such that the dependencies between the metadata and data are respected. For example, a data cluster should be flushed to disk before updating the lookup table; otherwise, the entry in the lookup table may point to some garbage data. The *reference table* should be updated before updating the *lookup table*; otherwise, the lookup table may point to some unallocated data cluster.

We use two simple benchmarks in QEMU-2.1.2 to compare the number of sync operations in the guest VM and the host: 1) "overwrite benchmark", which allocates blocks in advance in the disk image (i.e., the *qcow2* image size remains the same before and after the test); 2) "append benchmark", which allocates new blocks in the disk image during the test (i.e., the image size increases after the test). The test writes 64KB data and calls *fdatasync* every 50 iterations. We find that the virtual disks introduce more than 3X sync operations for *qcow2* and 4X for *VMDK* images, as shown in Figure 2.

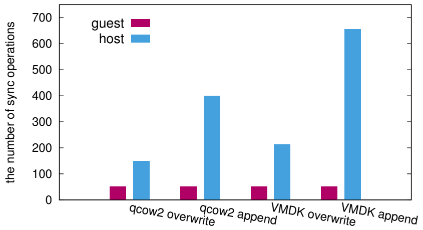


Figure 2: #sync operations observed from inside/outside of the VM.

As shown in Figure 3, a *fdatasync* of the user application can cause a transaction commit in the file system. This requires two flushes (in guest VM) to preserve its atomicity, which are then translated into two set of writes in QEMU. The first write puts the data and the journal metadata of the VM to the virtual disk, which in the worst case, causes its size to grow.

To grow the virtual disk in QEMU, a data block must be allocated, and the corresponding *reference table* block should be set strictly before other operations. This necessitates the first flush. After that, the L2 data block must be updated strictly before the remaining operations. This necessitates the second flush. (In some extreme cases where the L1 data block should be updated as well, it introduces even more flushes). The third flush is used to update the base image's *reference table*. When creating a new image based on the base image, the refcount in the *reference table* of the base image will be increased by one to indicate that another image uses the base image's data. When updating the new image, *qcow2* will copy data from the base image to a new place and do updates. The new image will use the COW data and will not access the old data in the base image, so the refcount in the base image should be decreased by one. The third flush is used to make the *reference table* of the base image durable. The fourth flush is introduced solely because of the suboptimal implementation in QEMU. The second write is the same as the first one, which needs four flushes. Consequently, we need around eight flushes for one guest *fdatasync* at most.

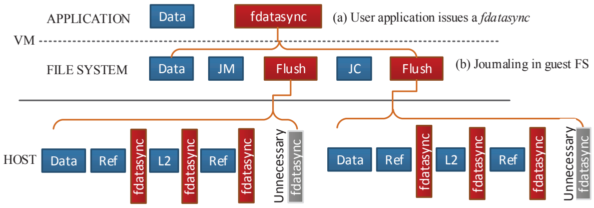


Figure 3: Illustration of sync amplification: it shows how the number of sync operations increases after the user issues *fdatasync*. The *fdatasync* with red color is necessary to impose the write ordering, while *fdatasync* with gray color are solely because of the flawed implementation of *qcow2*. J_M is for *journal of metadata*, J_C is for *journal of commit block*, *Ref* is for *reference table*, *L2* is for *L2 lookup table*.

2.3 Other Virtual Disk Formats

Other virtual disks have a similar structure to *qcow2* virtual disk. They have an image header to describe the basic information of the virtual disk, a block map table to translate virtual block address to physical block address, and many data blocks to store the guest VM's data. For example, the *grain directory* and *grain table* in the VMDK consist of a two-level table to do address translation. The VMDK also keeps two copies of the *grain directories* and *grain tables* on disk to improve the virtual disk's resilience. FVD [22] even has a bitmap for the copy-on-read feature. In summary, the organization of virtual disks will translate one update operation in the guest into several update operations in the host. Besides, the virtual disks should carefully schedule the update order to preserve crash consistency, which introduces more sync operations. Actually, our evaluation shows that VMDK has more severe sync amplification than qcow2, as shown in Figure 2.

3 Optimizations

3.1 Per Virtual Disk Internal Journaling

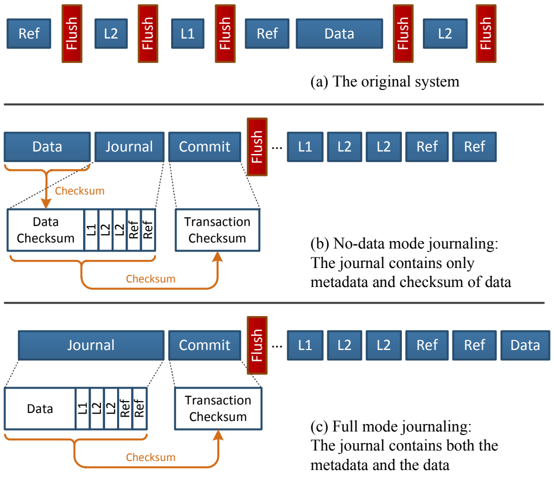
According to our analysis, we found that the cause of the extra sync operations is the overly-constrained semantics imposed during the modification of virtual disks. This is because the underlying file system cannot preserve the internal consistency of a virtual disk, in which certain data serves as metadata to support the rich features. As a result, the virtual disk has to impose strong ordering while being updated for the sake of crash consistency.

Based on this observation, we designed a *per virtual disk internal journaling* mechanism. As the name suggests, each virtual disk maintains an independent and internal journal, residing in a preallocated space of the virtual disk to which the journal belongs. The *per virtual disk internal journal* works in a manner similar to the normal file system journal, with the exception that it only logs the modification of the content of a single virtual disk.

On a virtual disk update, the metadata (e.g., *reference table* and *lookup table*) as well as the changed data are first logged into the preallocated virtual disk journaling area, which is treated as a transaction. At the end of this update, the journal commit block of a virtual disk is appended to the end of the transaction, indicating this transaction is committed. If any failures occur before the commit block is made durable, the whole transaction is canceled, and the virtual disk is still in a consistent state. We also leverage the checksum mechanism: by calculating a checksum of all the blocks in the transaction, we reduce the number of flushes to one per transaction.

Like other journaling mechanisms, we should install the modifications logged in the journal to their home place, i.e., checkpoint. To improve the performance of checkpoint, we delay the checkpoint process for batching and write absorbing. After a checkpoint, the area took up by this transaction can be reclaimed and reused in the future.

The *per virtual disk journaling* relaxes the original overly-constrained ordering requirement to an all-or-nothing manner, which reduces the number of flushes while retaining crash consistency.



(a) The original system

(b) No-data mode journaling: The journal contains only metadata and checksum of data

(c) Full mode journaling: The journal contains both the metadata and the data

Figure 4: The dual-mode journaling optimization: it shows the process of data updating on a *qcow2* virtual disk, as mentioned in section 2.1. (*L1 means level-1 lookup table, L2 means level-2 lookup table, and Ref means reference table*)

Dual-Mode Journaling: Journaling mechanism requires logged data to be written twice. This, under certain circumstances, severely degrades performance due to wasted disk bandwidth. To this end, we apply an optimization similar to Prabhakaran et al. [18] and Shen et al. [21]. This optimization, namely *dual-mode journaling*, drives the *per virtual disk internal journaling* to dynamically adapt between two modes according to the virtual disk's access pattern. Specifically, it only logs all the data when there is an overwriting of the original data; otherwise, only metadata (*reference table* and *lookup table*) is journaled.

It should be noted that *dual-mode journaling* differs from prior work [21, 18] in that it leverages different journaling mode for a single file, and thus it does not cause any inconsistency issues discussed in [21]. This is because our journaling mechanism only deals with a single file while prior work [21, 18] needs to deal with a complete file system. The two modes used are described as follows.

No-data Journaling Mode: Instead of writing the data into the journaling, the *no-data journaling mode* simply calculates a checksum of the data and puts only the checksum into the journal. The data is written to the "home" place. The process is shown in Figure 4-b.

The Overwriting Problem and Full Journaling Mode: The *no-data journaling mode*, however, can render the recovery process inconsistent upon data overwriting. This is because the correctness of recovery relies on the checksum of the blocks in the transaction, which does not necessarily reside in the journaling area since *no-data mode journaling* does not log data blocks. The content of those data blocks can be arbitrarily affected by the following overwriting operations; the whose checksum, of course, can be different from the time when it was committed.

Consequently, if a transaction is not fully checkpointed when its following transaction aborts, and at the same time the data of the previous transaction is partially overwritten, the recovery process will erroneously consider the already committed transaction as a broken one. Therefore, if a disk write transaction needs to overwrite the data which has not been checkpointed, the system will switch to full-mode journaling and put the entire data into the journal, as shown in Figure 4-c.

Crash Recovery: During crash recovery, we scan the journal to find the first transaction that has not been checkpointed. Then, we calculate the checksum of the journal data and other related data in this transaction. If the calculated checksum is the same as the checksum recorded in the current journal transaction, we apply the logged modifications for the data and metadata to their home place and do recovery for the next transaction. It the two checksums are not equal, it means that the transaction is not completely written, and we reach the end of the journal. We abort the transaction and finish crash recovery.

Other Implementation Issues: The current suboptimal implementation of *qcow2* image format involves some unnecessary sync operations (as shown in grey boxes in Figure 3). We just remove these sync operations without affecting the consistency.

With the above operations, we can reduce the number of syncs to one for each flush request from the guest VM. In another word, if the guest OS issues a flush operation, there will be only one sync on the host.

3.2 Adaptive Preallocation

We further diagnose the behavior of the host file system when handling guest VM's sync requests, and find that the actual number of disk flushes are usually more than the number of sync requests. This is because, if a write from the guest VM increases the size of its image file, the host file system will trigger a journal commit, which flushes the disk twice, the first for the data and the second for the journal commit block.

More specifically, Linux provides three syscalls (*msync*, *fsync*, *fdatasync*) for the sync operations. The *fsync* transfers all modified in-memory data in the file referred to by the file descriptor (fd) to the disk. The *fdatasync* is similar to *fsync*, but it does not flush metadata unless that metadata is needed in order to allow a subsequent data retrieval to be correctly handled (i.e., the file size). The *msync* is used for memory-mapped I/O. In *qcow2*, it uses *fdatasync* to make data persistent on the disk. Thus, if a guest VM's sync request does not increase the disk image's size, there will be only one flush operation for the data; but if the image size changes, the host file system will commit a transaction and cause an extra disk flush.

We leverage an adaptive preallocation approach to reducing the number of journal commits in the host. When the size of the image file needs to grow, we append more blocks than those actually required. A future virtual disk write operation which originally extends the image file can now be transformed into an "overwrite" operation. In this case, *fdatasync* will not force a journal commit in the host, which can reduce the latency of sync operation.

Specifically, we compare the position of the write operation with the image size. If the position of the write operation exceeds the image size, we will do the preallocation. Currently, the size of preallocated space is 1MB.

4 Evaluation

We implemented the optimizations in QEMU-2.1.2, which comprise 1300 LoC. This section presents our evaluation of the optimized *qcow2* from two aspects: consistency and performance.

We conducted the experiments on a machine with a 4-core Intel Xeon E3-1230 V2 CPU (3.3GHz) and 8GB memory. We use 1 TB WDC HDD and 120G Samsung 850 EVO SSD as the underlying storage devices. The host OS is Ubuntu 14.04; the guest OS is Ubuntu 12.04. Both guest and host use the ext4 file system. We use KVM [10] and configure each VM with 1 VCPU, 2GB memory, and 10GB disk. The cache mode of each VM is *writeback*, which is the default setting. It has good performance while being safe as long as the guest VM correctly flushes necessary disk caches [20].

4.1 Consistency

To validate the consistency properties, we implement a trace record and replay framework. We run a test workload in the guest and record the write operations and sync operations in the host. Then, we randomly choose a sync operation in the trace and replay all the I/O operations above this sync operation on a backup image (the image is a copy of the guest image before running the test workload). After that, we choose some write operations between this sync and the next sync operation, and apply these writes to the backup image Finally, we use the *qemu-img* tool to check the consistency of virtual disk image.

We record two traces, one for the *append* workload and the other for *append* + *overwrite* workload. In the *append* workload, we append 64k data and then call *fdatasync* in the VM. In the *append* + *overwrite* workload, we *append* 64k data, call *fdatasync*, overwrite the 64k data and then call *fdatasync*. We simulated 200 crash scenarios for each workload. We divide data in the virtual disk image into four types: guest data, metadata (i.e., Lookup table), journal data (i.e., the journal record for metadata's modification) and journal commit block. The 200 crash scenarios for each workload contain all four types of data loss. The result shows that optimized *qcow2* can recover correctly and get a consistent state for all test cases.

4.2 Performance

We first present the performance of synchronous overwrite and append workloads. For overwrite workload, we generate a base image and allocate space in advance, then overlay a small image atop the base image. For append workload, we do the experiment directly on a newly allocated image. We also run the *Filebench* [15] and *TPCC* [4] workload. *Filebench*'s *varmail* simulates a mail server workload and will frequently call sync operations. TPCC simulates a complete environment where a population of terminal operators executes against a database. We run the experiments under three configurations. For the first configuration (raw), we only boot one VM and run the tests. For the second configuration (seq), we boot two VMs, one for the experiments and the other is doing sequential I/O all the time. For the third configuration (ran), we also have two VMs, one for the experiments and the other is doing random I/O all the time.

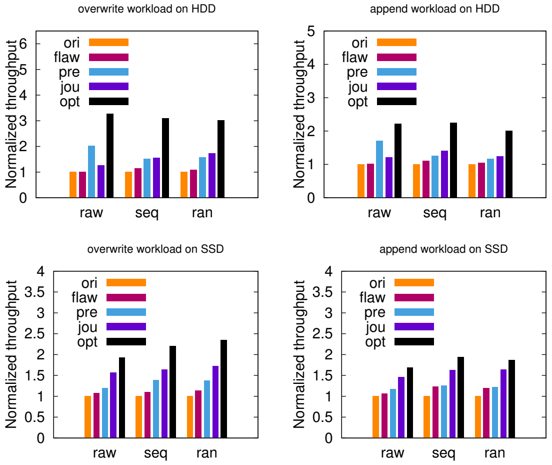


Figure 5: Micro-benchmark result on 2 storage media. *Each separate optimization as well as the overall result is showed. "ori" refers to the original system; "flaw" refers to the system which removes unnecessary sync operations caused by qcow2 flawed implementation; "pre" refers to the adaptive prealloacation; "jou" refers to the per virtual disk internal journal; "opt" refers to the overall result.*

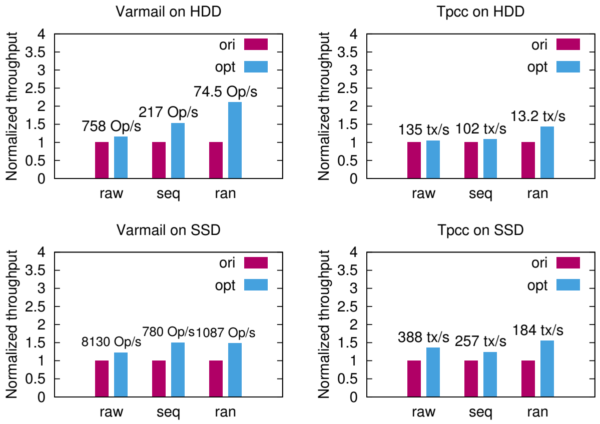


Figure 6: Macro-benchmark on 2 storage media.

Figure 5 and Figure 6 show the performance results on both HDD (hard disk driver) and SSD. For overwrite workload, our system improves the throughput by 200% for disk and 100% for SSD. For *varmail*, our system achieves 110% speedup when running *varmail* together with a random workload on HDD. The performance gain for *varmail* on SSD is about 50% when running *varmail* together with a sequential workload.

Figure 7 compares the TPCC transaction latency between our system and the original system. The results show that our system has lower latency, and the latency even decreased by 40% when TPCC is running together with a random workload.

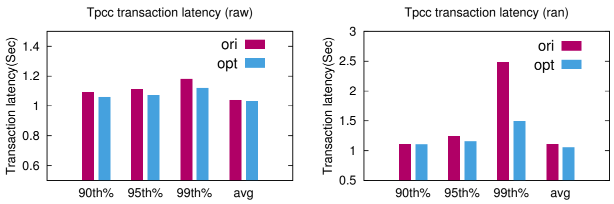


Figure 7: Latency for *TPCC*.

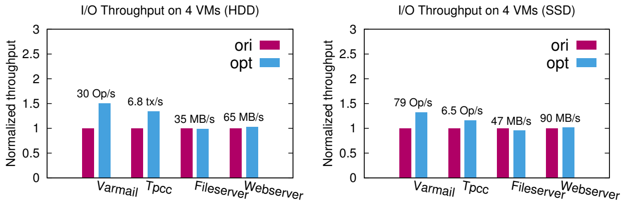


Figure 8: Evaluation for multiple VMs.

We also evaluate the multiple VM configurations. We run four VMs with *varmail*, *TPCC*, *fileserver* and *webserver* workloads respectively. *Varmail* and *TPCC* workloads issue syncs frequently, while *fileserver* and *webserver* have fewer sync operations.

Figure 8 shows the evaluation results for multiple VMs test. On HDD, the performance of *varmail* and *tpcc* improves 50% and 34%, respectively. On SSD, the performance gain is 30% and 20%, respectively. Besides, *fileserver* and *webserver* on the optimized system have similar performance to those on the original system. This is because fileserver and webserver have few sync operations and do not update *qcow2* metadata frequently.

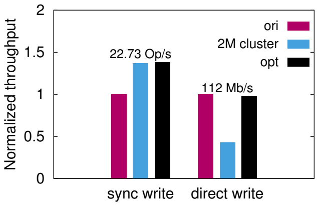


Figure 9: *"Sync write" means calling fdatasync after each write operation; "Direct write" means opening the file with O_DIRECT flag; "2M cluster" means the cluster size is 2M; The cluster size for "ori" and "opt" is 64k.*

Qcow2, VMDK and VDI support big clusters (1MB or more). Big clusters decrease the frequency of metadata operations and can mitigate the sync amplification. Figure 9 shows using big clusters may mitigate sync amplification as our approach for the *sync write* workload. However, it results in 50% performance degradation for the *direct write* workload. This is because bigger clusters increase the cost of allocating a new cluster [8]. Besides, bigger clusters reduce the compactness of sparse files. In contrast, our optimizations mitigate sync amplification without such side effects.

5 Related Work

Reducing sync operations has been intensively studied [1, 2, 21, 18, 12, 17]. For example, adaptive journaling [21, 18] is a common approach to reducing journaling incurred sync operations. OptFS [1] advocates separation of durability and ordering and split the sync operation into *dsync* and *osync* for file systems accordingly. However, it requires the underlying device to provide asynchronous durability notification interface. Our work focuses on the virtual disk image format and is transparent to the guest VM. NoFS [2] eliminates all the ordering points and uses the *backpointer* to maintain crash consistency; but it is hard to implement atomic operations, such as *rename*. Xsyncfs [17] also aimed to improve sync performance. It delays sync operations until an external observer reads corresponding data. By delaying the sync operations, there is more space for I/O scheduler to batch and absorb write operations.

Improving file system performance for the virtual machine is also a hot research topic [11, 14, 22, 19]. Le [11] analyzed the performance of nested file systems in virtual machines. Li [14] proposed to accelerate guest VM's sync operation by saving the syncing data in host memory and returning directly without writing to disk. FVD [22] is a high-performance virtual disk format. It supports many features, such as copy-on-read and prefetching. QED [19] is designed to avoid some of the pitfalls of qcow2 and is expected to be more performant than qcow2. All these work did not address the *sync amplification* problem.

Our work leverages several prior techniques such as checksum [3, 5] and pre-allocation [13], but applies them to solve a new problem.

6 Conclusion

This paper uncovered the *sync amplification* problem of copy-on-write virtual disks. It then described three optimizations to minimize sync operations while preserving crash consistency. The evaluation showed that the optimizations notably boost some I/O intensive workloads.

7 Acknowledgment

We thank our shepherd Nisha Talagala and the anonymous reviewers for their constructive comments. This work is supported in part by China National Natural Science Foundation (61572314, 61303011), a research grant from Samsung, Inc., National Youth Top-notch Talent Support Program of China, Zhangjiang Hi-Tech program (No. 201501-YP-B108-012), and Singapore NRF (CREATE E2S2).

References

[1] Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 228–243.

[2] Chidambaram, V., Sharma, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Consistency without ordering. In *FAST* (2012), p. 9.

[3] Cohen, F. A cryptographic checksum for integrity protection. *Computers & Security 6*, 6 (1987), 505–510.

[4] Difallah, D. E., Pavlo, A., Curino, C., and Cudre-Mauroux, P. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment 7*, 4 (2013), 277–288.

[5] Ext4 Wiki. Ext4 metadata checksums. `https://ext4.wiki.kernel.org/index.php/Ext4_Metadata_Checksums`.

[6] Forums, V. B. Problem for virtualbox doesn't flush/commit. `https://forums.virtualbox.org/viewtopic.php?t=13661`.

[7] Hajnoczi, S. cluster size parameter on qcow2 image format. `http://www.linux-kvm.org/images/d/d6/Kvm-forum-2013-toward_qcow2_deduplication.pdf`.

[8] Hajnoczi, S. cluster size parameter on qcow2 image format. `http://lists.gnu.org/archive/html/qemu-devel/2012-02/msg03164.html`.

[9] Jin, K., and Miller, E. L. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), ACM, p. 7.

[10] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[11] Le, D., Huang, H., and Wang, H. Understanding performance implications of nested file systems in a virtualized environment. In *FAST* (2012), p. 8.

[12] Lee, W., Lee, K., Son, H., Kim, W.-H., Nam, B., and Won, Y. Waldio: eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 235–247.

[13] Leung, M. Y., Lui, J., and Golubchik, L. Buffer and i/o resource pre-allocation for implementing batching and buffering techniques for video-on-demand systems. In *Data Engineering, 1997. Proceedings. 13th International Conference on* (1997), IEEE, pp. 344–353.

[14] Li, D., Liao, X., Jin, H., Zhou, B., and Zhang, Q. A new disk i/o model of virtualized cloud environment. *Parallel and Distributed Systems, IEEE Transactions on 24*, 6 (2013), 1129–1138.

[15] McDougall, R., and Mauro, J. Filebench. `http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench`, 2005.

[16] McLoughlin, M. The qcow2 image format. `https://people.gnome.org/~markmc/qcow-image-format.html`, 2011.

[17] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the sync. *ACM Transactions on Computer Systems (TOCS) 26*, 3 (2008), 6.

[18] Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference, General Track* (2005), pp. 105–120.

[19] QEMU Wiki. Qed specification. `http://wiki.qemu.org/Features/QED/Specification`.

[20] QEMU Wiki. Qemu emulator user documentation. `http://qemu.weilnetz.de/qemu-doc.html`.

[21] Shen, K., Park, S., and Zhu, M. Journaling of journal is (almost) free. In *FAST* (2014), pp. 287–293.

[22] Tang, C. Fvd: A high-performance virtual machine image format for cloud. In *USENIX Annual Technical Conference* (2011).

[23] Virtual Box Manual. Responding to guest ide/sata flush requests. `https://www.virtualbox.org/manual/ch12.html`.

[24] VMware Support. Performance best practices for vmware workstation. `www.vmware.com/pdf/ws7_performance.pdf`.

[25] wikipedia. diskbuffer. `https://en.wikipedia.org/wiki/Disk_buffer`.

Uncovering Bugs in Distributed Storage Systems during Testing (not in Production!)

Pantazis Deligiannis[†1], Matt McCutchen[◇1], Paul Thomson[†1], Shuo Chen[⋆]
Alastair F. Donaldson[†], John Erickson[⋆], Cheng Huang[⋆], Akash Lal[⋆]
Rashmi Mudduluru[⋆], Shaz Qadeer[⋆], Wolfram Schulte[⋆]

[†]*Imperial College London,* [◇]*Massachusetts Institute of Technology,* [⋆]*Microsoft*

Abstract

Testing distributed systems is challenging due to multiple sources of nondeterminism. Conventional testing techniques, such as unit, integration and stress testing, are ineffective in preventing serious but subtle bugs from reaching production. Formal techniques, such as TLA+, can only verify high-level specifications of systems at the level of logic-based models, and fall short of checking the actual executable code. In this paper, we present a new methodology for testing distributed systems. Our approach applies advanced systematic testing techniques to thoroughly check that the executable code adheres to its high-level specifications, which significantly improves coverage of important system behaviors.

Our methodology has been applied to three distributed storage systems in the Microsoft Azure cloud computing platform. In the process, numerous bugs were identified, reproduced, confirmed and fixed. These bugs required a subtle combination of concurrency and failures, making them extremely difficult to find with conventional testing techniques. An important advantage of our approach is that a bug is uncovered in a small setting and witnessed by a full system trace, which dramatically increases the productivity of debugging.

1 Introduction

Distributed systems are notoriously hard to design, implement and test [6, 31, 20, 27, 35]. This challenge is due to many sources of *nondeterminism* [7, 23, 32], such as unexpected node failures, the asynchronous interaction between system components, data losses due to unreliable communication channels, the use of multithreaded code to exploit multicore machines, and interaction with clients. All these sources of nondeterminism can easily create *Heisenbugs* [16, 38], corner-case bugs that are difficult to detect, diagnose and fix. These bugs might hide inside a code path that can only be triggered by a specific interleaving of concurrent events and only manifest under extremely rare conditions [16, 38], but the consequences can be catastrophic [1, 44].

Developers of production distributed systems use many testing techniques, such as unit testing, integration testing, stress testing, and fault injection. In spite of extensive use of these testing methods, many bugs that arise from subtle combinations of concurrency and failure events are missed during testing and get exposed only in production. However, allowing serious bugs to reach production can cost organizations a lot of money [42] and lead to customer dissatisfaction [1, 44].

We interviewed technical leaders and senior managers in Microsoft Azure regarding the top problems in distributed system development. The consensus was that one of the most critical problems today is how to improve *testing coverage* so that bugs can be uncovered *during testing* and *not in production*. The need for better testing techniques is not specific to Microsoft; other companies, such as Amazon and Google, have acknowledged [7, 39] that testing methodologies have to improve to be able to reason about the correctness of increasingly more complex distributed systems that are used in production.

Recently, the Amazon Web Services (AWS) team used formal methods “to prevent serious but subtle bugs from reaching production” [39]. The gist of their approach is to extract the high-level logic from a production system, represent this logic as specifications in the expressive TLA+ [29] language, and finally verify the specifications using a model checker. While highly effective, as demonstrated by its use in AWS, this approach falls short of “verifying that executable code correctly implements the high-level specification”' [39], and the AWS team admits that it is “not aware of any such tools that can handle distributed systems as large and complex as those being built at Amazon” [39].

We have found that checking high-level specifications is necessary but not sufficient, due to the gap between

[1] Part of the work was done while interning at Microsoft.

the specification and the executable code. Our goal is to bridge this gap. We propose a new methodology that validates high-level specifications directly on the executable code. Our methodology is different from prior approaches that required developers to either switch to an unfamiliar domain specific language [25, 10], or manually annotate and instrument their code [45]. Instead, we allow developers to systematically test *production code* by writing test harnesses in C#, a mainstream programming language. This significantly lowered the acceptance barrier for adoption by the Microsoft Azure team.

Our testing methodology is based on P# [9], an extension of C# that provides support for modeling, specification, and systematic testing of distributed systems written in the Microsoft .NET framework. To use P# for testing, the programmer has to augment the original system with three artifacts: a model of the nondeterministic execution environment of the system; a test harness that drives the system towards interesting behaviors; and safety or liveness specifications. P# then systematically exercises the test harness and validates program behaviors against the provided specifications.

The original P# paper [9] discussed language design issues and data race detection for programs written in P#, whereas this work focuses on using P# to test three distributed storage systems inside Microsoft: Azure Storage vNext; Live Table Migration; and Azure Service Fabric. We uncovered numerous bugs in these systems, including a subtle liveness bug that only intermittently manifested during stress testing for months without being fixed. Our testing approach uncovered this bug in a very small setting, which made it easy for developers to examine traces, identify, and fix the problem.

To summarize, our contributions are as follows:

- We present a new methodology for modeling, specifying properties of correctness, and systematically testing real distributed systems with P#.
- We discuss our experience of using P# to test three distributed storage systems built on top of Microsoft Azure, finding subtle bugs that could not be found with traditional testing techniques.
- We evaluate the cost and benefits of using our approach, and show that P# can detect bugs in a small setting and with easy to understand traces.

2 Testing Distributed Systems with P#

The goal of our work is to find bugs in distributed systems *before* they reach production. Typical distributed systems consist of multiple components that interact with each other via *message passing*. If messages—or unexpected failures and timeouts—are not handled properly, they can lead to subtle bugs. To expose these bugs, we use P# [9], an extension of the C# language that provides: (i) language support for *specifying* properties of *correctness*, and *modeling* the environment of distributed systems written in .NET; and (ii) a *systematic testing engine* that can explore interleavings between *distributed events*, such as the nondeterministic order of message deliveries, client requests, failures and timeouts.

Modeling using P# involves three core activities. First, the developer must modify the original system so that messages are not sent through the real network, but are instead dispatched through the `PSharp.Send(...)` method. Such modification does not need to be invasive, as it can be performed using virtual method dispatch, a C# language feature widely used for testing. Second, the developer must write a P# *test harness* that drives the system towards interesting behaviors by nondeterministically triggering various events (see §2.3). The harness is essentially a model of the environment of the system. The purpose of these first two activities is to explicitly declare all sources of nondeterminism in the system using P#. Finally, the developer must specify the criteria for correctness of an execution of the system-under-test. Specifications in P# can encode either *safety* or *liveness* [28] properties (see §2.4 and §2.5).

During testing, the P# runtime is aware of all sources of nondeterminism that were declared during modeling, and exploits this knowledge to create a scheduling point each time a nondeterministic choice has to be taken. The P# testing engine will serialize (in a single-box) the system, and repeatedly execute it from start to completion, each time exploring a potentially different set of nondeterministic choices, until it either reaches a user-supplied bound (e.g. in number of executions or time), or it hits a safety or liveness property violation. This testing process is fully automatic and has no false-positives (assuming an accurate test harness). After a bug is discovered, P# generates a trace that represents the buggy schedule, which can then be replayed to reproduce the bug. In contrast to logs typically generated during production, the P# trace provides a global order of all communication events, and thus is easier to debug.

Due to the highly asynchronous nature of distributed systems, the number of possible states that these systems can reach is exponentially large. Tools such as MoDist [48] and dBug [45] focus on testing unmodified distributed systems, but this can easily lead to state-space explosion when trying to exhaustively explore the entire state-space of a production-scale distributed storage system, such as the Azure Storage vNext. On the other hand, techniques such as TLA+ [29] have been successfully used in industry to verify specifications of complex distributed systems [39], but they are unable to verify the actual implementation.

Server

```
receive msg {
  case ClientReq:
    this.Data = message.Val;
    // Replicate data to all nodes
    foreach (sn in this.Nodes)
      sn.send(ReplReq, this.Data);
  case Sync:
    Node node = message.Id;
    Log log = message.Log;
    doSync(node, log);
}
```

```
doSync (Node sn, Log log) {
  // If the storage log is not
  // up-to-date, replicate
  if (!isUpToDate(log))
    sn.send(ReplReq, this.Data);
  else {
    this.NumReplicas++;
    if (this.NumReplicas == 3)
      this.Client.send(Ack);
  }
}
```

Storage Node

```
receive msg {
  case ReplReq:
    // Store received data
    store(message.Val);
  case Timeout:
    // Send server the log
    // upon timeout
    this.Server.send(Sync,
      this.Id, this.Log);
}
```

Client

```
while (hasNextRequest()) {
  this.Server.send(ClientReq,
    this.DataToReplicate);
  receive(Ack); // Wait for ack
}
```

Timer

```
// Send timeout to node when
// countdown reaches 0
if (this.Countdown == 0)
  this.SN.send(Timeout);
```

Figure 1: Pseudocode of a simple distributed storage system that replicates data sent by a client.

In this work, we are proposing a solution between the above two extremes: test the real implementation of one or more components against a modeled in P# environment. The benefit of our approach is that it can detect bugs in the *actual* implementation by exploring a much reduced state-space. Note that testing with P# does not come for free; developers have to invest effort and time into building a test harness using P#. However, developers already spend significant time in building test suites for distributed systems prior to deployment. The P# approach augments this effort; by investing time in modeling the environment, it offers dividends by finding more bugs (see §6). In principle, our methodology is *not* specific to P# and the .NET framework, and can be used in combination with any other programming framework that has equivalent capabilities.

2.1 The P# programming model

P# programs consist of multiple *state machines* that communicate with each other *asynchronously* by exchanging *events*. In the case of distributed systems, P# events can be used to model regular messages between system components, failures or timeouts. A P# machine declaration is similar to a C# class declaration, but a machine also contains an event queue, and one or more *states*. Each state can register *actions* to handle incoming events.

P# machines run concurrently with each other, each executing an event handling loop that dequeues the next event from the queue and handles it by invoking the registered action. An action might transition the machine to a new state, create a new machine, send an event to a machine, access a field or call a method. In P#, a send operation is *non-blocking*; the event is simply enqueued into the queue of the target machine, which will dequeue and handle the event concurrently. All this functionality is provided in a lightweight runtime library, built on top of Microsoft's Task Parallel Library [33].

2.2 An example distributed system

Figure 1 presents the pseudocode of a simple distributed storage system that was contrived for the purposes of explaining our testing methodology. The system consists of a client, a server and three storage nodes (SNs). The client sends the server a `ClientReq` message that contains data to be replicated, and then waits to get an acknowledgement before sending the next request. When the server receives `ClientReq`, it first stores the data locally (in the `Data` field), and then broadcasts a `ReplReq` message to all SNs. When an SN receives `ReplReq`, it handles the message by storing the received data locally (using the `store` method). Each SN has a timer installed, which sends periodic `Timeout` messages. Upon receiving `Timeout`, an SN sends a `Sync` message to the server that contains the storage log. The server handles `Sync` by calling the `isUpToDate` method to check if the SN log is up-to-date. If it is not, the server sends a repeat `ReplReq` message to the outdated SN. If the SN log is up-to-date, then the server increments a replica counter by one. Finally, when there are three replicas available, the server sends an `Ack` message to the client.

There are two bugs in the above example. The first bug is that the server does not keep track of unique replicas. The replica counter increments upon each up-to-date `Sync`, even if the syncing SN is already considered a replica. This means that the server might send `Ack` when fewer than three replicas exist, which is erroneous behavior. The second bug is that the server does not reset the replica counter to 0 upon sending an `Ack`. This means that when the client sends another `ClientReq`, it will never receive `Ack`, and thus block indefinitely.

2.3 Modeling the example system

To systematically test the example of Figure 1, the developer must first create a P# test harness, and then specify the correctness properties of the system. Figure 2 illustrates a test harness that can find the two bugs discussed in §2.2. Each box in the figure represents a concurrently running P# machine, while an arrow represents an event being sent from one machine to another. We use three types of boxes: (i) a box with rounded corners and thick border denotes a real component wrapped inside a P# machine; (ii) a box with thin border denotes a modeled

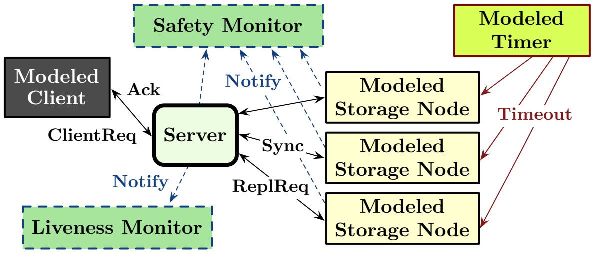


Figure 2: P# test harness for the Figure 1 example.

component in P#; and (iii) a box with dashed border denotes a special P# machine used for safety or liveness checking (see §2.4 and §2.5).

We do not model the server component since we want to test its actual implementation. The server is wrapped inside a P# machine, which is responsible for: (i) sending events via the `PSharp.Send(...)` method, instead of the real network; and (ii) delivering received events to the wrapped component. We model the SNs so that they store data in memory rather than on disk (which could be inefficient during testing). We also model the client so that it can drive the system by repeatedly sending a nondeterministically generated `ClientReq`, and then waiting for an `Ack` message. Finally, we model the timer so that P# takes control of all time-related nondeterminism in the system. This allows the P# runtime to control when a `Timeout` event will be sent to the SNs during testing, and systematically explore different schedules.

P# uses object-oriented language features such as interfaces and virtual method dispatch to connect the real code with the modeled code. Developers in industry are used to working with such features, and heavily employ them in testing production systems. In our experience, this significantly lowers the bar for engineering teams inside Microsoft to embrace P# for testing.

In §2.4 and §2.5, we discuss how safety and liveness specifications can be expressed in P# to check if the example system is correct. The details of how P# was used to model and test real distributed storage systems in Microsoft are covered in §3, §4 and §5. Interested readers can also refer to the P# GitHub repository[2] to find a manual and samples (e.g. Paxos [30] and Raft [40]).

2.4 Specifying safety properties in P#

Safety property specifications generalize the notion of source code assertions; a safety violation is a finite trace leading to an erroneous state. P# supports the usual assertions for specifying safety properties that are local to a P# machine, and also provides a way to specify global assertions by using a *safety monitor* [10], a special P# machine that can receive, but not send, events.

A safety monitor maintains local state that is modified in response to events received from ordinary (non-monitor) machines. This local state is used to maintain a history of the computation that is relevant to the property being specified. An erroneous global behavior is flagged via an assertion on the private state of the safety monitor. Thus, a monitor cleanly separates instrumentation state required for specification (inside the monitor) from program state (outside the monitor).

The first bug in the example of §2.2 is a safety bug. To find it, the developer can write a safety monitor (see Figure 2) that contains a map from unique SN ids to a Boolean value, which denotes if the SN is a replica or not. Each time an SN replicates the latest data, it notifies the monitor to update the map. Each time the server issues an `Ack`, it also notifies the monitor. If the monitor detects that an `Ack` was sent without three replicas actually existing, a safety violation is triggered.

2.5 Specifying liveness properties in P#

Liveness property specifications generalize nontermination; a liveness violation is an infinite trace that exhibits lack of progress. Typically, a liveness property is specified via a temporal logic formula [41, 29]. We take a different approach and allow the developers to write a *liveness monitor* [10]. Similar to a safety monitor, a liveness monitor can receive, but not send, events.

A liveness monitor contains two special types of states: *hot* and *cold*. A hot state denotes a point in the execution where progress is required but has not happened yet; e.g. a node has failed but a new one has not launched yet. A liveness monitor enters a hot state when it is notified that the system must make progress. The liveness monitor leaves the hot state and enters the cold state when it is notified that the system has progressed. An infinite execution is erroneous if the liveness monitor is in the hot state for an infinitely long period of time. Our liveness monitors can encode arbitrary temporal logic properties.

A liveness violation is witnessed by an *infinite* execution in which all concurrently executing P# machines are *fairly* scheduled. Since it is impossible to generate an infinite execution by executing a program for a finite amount of time, our implementation of liveness checking in P# approximates an infinite execution using several heuristics. In this work, we consider an execution longer than a large user-supplied bound as an “infinite” execution [25, 37]. Note that checking for fairness is not relevant when using this heuristic, due to our pragmatic use of a large bound.

The second bug in the example of §2.2 is a liveness bug. To detect it, the developer can write a liveness monitor (see Figure 2) that transitions from a hot state, which

[2] `https://github.com/p-org/PSharp`

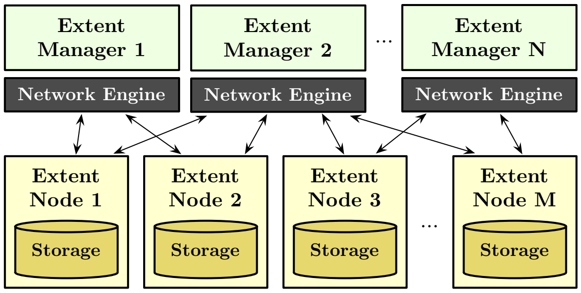


Figure 3: Top-level components for extent management in Microsoft Azure Storage vNext.

denotes that the client sent a `ClientReq` and waits for an `Ack`, to a cold state, which denotes that the server has sent an `Ack` in response to the last `ClientReq`. Each time a server receives a `ClientReq`, it notifies the monitor to transition to the hot state. Each time the server issues an `Ack`, it notifies the monitor to transition to the cold state. If the monitor is in a hot state when the bounded infinite execution terminates, a liveness violation is triggered.

3 Case Study: Azure Storage vNext

Microsoft Azure Storage is a cloud storage system that provides customers the ability to store seemingly limitless amounts of data. It has grown from tens of petabytes in 2010 to exabytes in 2015, with the total number of objects stored exceeding 60 trillion [17].

Azure Storage vNext is the next generation storage system currently being developed for Microsoft Azure, where the primary design target is to scale the storage capacity by more than 100×. Similar to the current system, vNext employs containers, called *extents*, to store data. Extents can be several gigabytes each, consisting of many data blocks, and are replicated over multiple *Extent Nodes* (ENs). However, in contrast to the current system, which uses a Paxos-based, centralized mapping from extents to ENs [5], vNext achieves scalability by using a *distributed mapping*. In vNext, extents are divided into partitions, with each partition managed by a lightweight *Extent Manager* (ExtMgr). This partitioning is illustrated in Figure 3.

One of the responsibilities of an ExtMgr is to ensure that every managed extent maintains enough *replicas* in the system. To achieve this, an ExtMgr receives frequent periodic *heartbeat* messages from every EN that it manages. EN failure is detected by missing heartbeats. An ExtMgr also receives less frequent, but still periodic, *synchronization reports* from every EN. The sync reports list all the extents (and associated metadata) stored on the EN. Based on these two types of messages, an ExtMgr identifies which ENs have failed, and which extents are affected by the failure and are missing replicas as a result. The ExtMgr then schedules tasks to repair the affected extents and distributes the tasks to the remaining ENs. The ENs then repair the extents from the existing replicas and lazily update the ExtMgr via their next periodic sync reports. All the communications between an ExtMgr and the ENs occur via network engines installed in each component of vNext (see Figure 3).

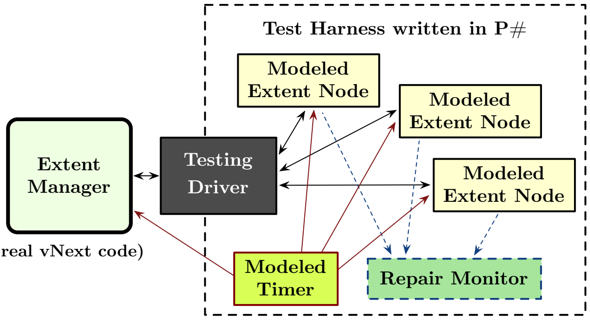


Figure 4: Real Extent Manager with its P# test harness (each box represents one P# machine).

To ensure correctness, the developers of vNext have instrumented extensive, multiple levels of testing:

1. *Unit testing*, in which emulated heartbeats and sync reports are sent to an ExtMgr. These tests check that the messages are processed as expected.

2. *Integration testing*, in which an ExtMgr is launched together with multiple ENs. An EN failure is subsequently injected. These tests check that the affected extents are eventually repaired.

3. *Stress testing*, in which an ExtMgr is launched together with multiple ENs and many extents. The test keeps repeating the following process: injects an EN failure, launches a new EN and checks that the affected extents are eventually repaired.

Despite the extensive testing efforts, the vNext developers were plagued for months by an elusive bug in the ExtMgr logic. All the unit test suites and integration test suites successfully passed on each test run. However, the stress test suite failed *from time to time* after very long executions; in these cases, certain replicas of some extents failed without subsequently being repaired.

This bug proved difficult to identify, reproduce and troubleshoot. First, an extent never being repaired is *not* a property that can be easily checked. Second, the bug appeared to manifest only in very long executions. Finally, by the time that the bug did manifest, very long execution traces had been collected, which made manual inspection tedious and ineffective.

To uncover the elusive extent repair bug in Azure Storage vNext, its developers wrote a test harness using P#. The developers expected that it was more likely for the

```
// wrapping the target vNext component in a P# machine
class ExtentManagerMachine : Machine {
  private ExtentManager ExtMgr; // real vNext code

  void Init() {
    ExtMgr = new ExtentManager();
    ExtMgr.NetEngine = new ModelNetEngine(); // model network
    ExtMgr.DisableTimer(); // disable internal timer
  }

  [OnEvent(ExtentNodeMessageEvent, DeliverMessage)]
  void DeliverMessage(ExtentNodeMessage msg) {
    // relay messages from Extent Node to Extent Manager
    ExtMgr.ProcessMessage(msg);
  }

  [OnEvent(TimerTickEvent, ProcessExtentRepair)]
  void ProcessExtentRepair() {
    // extent repair loop driven by external timer
    ExtMgr.ProcessEvent(new ExtentRepairEvent());
  }
}
```

Figure 5: The real Extent Manager is wrapped inside the `ExtentManager` P# machine.

bug to occur in the ExtMgr logic, rather than in the EN logic. Hence, they focused on testing the real ExtMgr using modeled ENs. The test harness for vNext consists of the following P# machines (as shown in Figure 4):

ExtentManager acts as a thin wrapper machine for the real ExtMgr component in vNext (see §3.1).

ExtentNode is a simple model of an EN (see §3.2).

Timer exploits the nondeterministic choice generation available in P# to model timeouts (see §3.3).

TestingDriver is responsible for driving testing scenarios, relaying messages between machines, and injecting failures (see §3.4).

RepairMonitor collects EN-related state to check if the desired liveness property is satisfied (see §3.5).

3.1 The ExtentManager machine

The real ExtMgr in vNext, which is our system-under-test, is wrapped inside the `ExtentManager` machine, as illustrated in the code snippet of Figure 5.

Real Extent Manager. The real ExtMgr (see Figure 6) contains two data structures: `ExtentCenter` and `ExtentNodeMap`. The `ExtentCenter` maps extents to their hosting ENs. It is updated upon receiving a periodic sync report from an EN. Recall that a sync report from a particular EN lists all the extents stored at the EN. Its purpose is to update the ExtMgr's possibly out-of-date view of the EN with the ground truth. The `ExtentNodeMap` maps ENs to their latest heartbeat times.

ExtMgr internally runs a periodic *EN expiration loop* that is responsible for removing ENs that have been missing heartbeats for an extended period of time, as well as cleaning up the corresponding extent records in `ExtentCenter`. In addition, ExtMgr runs a periodic *extent repair loop* that examines all the `ExtentCenter` records, identifies extents with missing replicas, schedules extent repair tasks and sends them to the ENs.

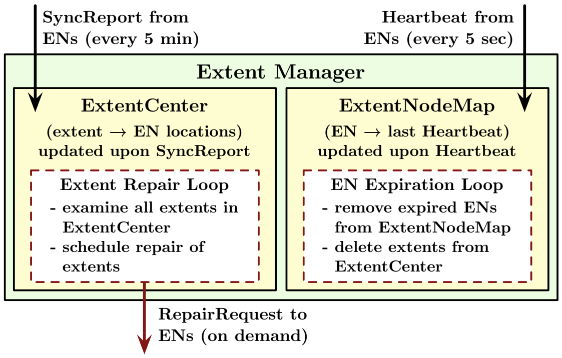


Figure 6: Internal components of the real Extent Manager in Microsoft Azure Storage vNext.

```
// network interface in vNext
class NetworkEngine {
  public virtual void SendMessage(Socket s, Message msg);
}

// modeled engine for intercepting Extent Manager messages
class ModelNetEngine : NetworkEngine {
  public override void SendMessage(Socket s, Message msg) {
    // intercept and relay Extent Manager messages
    PSharp.Send(TestingDriver, new ExtMgrMsgEvent(), s, msg);
  }
}
```

Figure 7: Modeled vNext network engine.

Intercepting network messages. The real ExtMgr uses a network engine to asynchronously send messages to ENs. The P# test harness models the original network engine in vNext by overriding the original implementation. The modeled network engine (see Figure 7) intercepts all outbound messages from the ExtMgr, and invokes `PSharp.Send(...)` to asynchronously relay the messages to `TestingDriver` machine, which is responsible for dispatching the messages to the corresponding ENs. This modeled network engine replaces the real network engine in the wrapped ExtMgr (see Figure 5).

Intercepting all network messages and dispatching them through P# is important for two reasons. First, it allows P# to systematically explore the interleavings between asynchronous event handlers in the system. Second, the modeled network engine could leverage the support for controlled nondeterministic choices in P#, and choose to drop the messages in a nondeterministic fashion, in case emulating message loss is desirable (not shown in this example).

Messages coming from `ExtentNode` machines do *not* go through the modeled network engine; they are instead delivered to the `ExtentManager` machine and trigger an action that invokes the messages on the wrapped

```
// modeling Extent Node in P#
class ExtentNodeMachine : Machine {
  // leverage real vNext component whenever appropriate
  private ExtentNode.ExtentCenter ExtCtr;

  [OnEvent(ExtentCopyResponseEvent, ProcessCopyResponse)]
  void ProcessCopyResponse(ExtentCopyResponse response) {
    // extent copy response from source replica
    if (IsCopySucceeded(response)) {
      var rec = GetExtentRecord(response);
      ExtCtr.AddOrUpdate(rec); // update ExtentCenter
    }
  }

  // extent node sync logic
  [OnEvent(TimerTickEvent, ProcessExtentNodeSync)]
  void ProcessExtentNodeSync() {
    var sync = ExtCtr.GetSyncReport(); // prepare sync report
    PSharp.Send(ExtentManagerMachine,
      new ExtentNodeMessageEvent(), sync);
  }

  // extent node failure logic
  [OnEvent(FailureEvent, ProcessFailure)]
  void ProcessFailure() {
    // notifies the monitor that this EN failed
    PSharp.Notify<RepairMonitor>(new ENFailedEvent(), this);
    PSharp.Halt(); // terminate this P# machine
  }
}
```

Figure 8: Code snippet of the modeled EN.

ExtMgr with `ExtMgr.ProcessMessage` (see Figure 5). The benefit of this approach is that the real ExtMgr can be tested without modifying the original communication-related code; the ExtMgr is simply unaware of the P# test harness and behaves as if it is running in a real distributed environment and communicating with real ENs.

3.2 The ExtentNode machine

The `ExtentNode` machine is a simplified version of the original EN. The machine omits most of the complex details of a real EN, and only models the necessary logic for testing. This modeled logic includes: repairing an extent from its replica, and sending sync reports and heartbeat messages periodically to `ExtentManager`.

The P# test harness leverages components of the real vNext system whenever it is appropriate. For example, `ExtentNode` re-uses the `ExtentCenter` data structure, which is used inside a real EN for extent bookkeeping. In the modeled extent repair logic, `ExtentNode` takes action upon receiving an extent repair request from the `ExtentManager` machine. It sends a copy request to a source `ExtentNode` machine where a replica is stored. After receiving an `ExtentCopyRespEvent` event from the source machine, it updates the `ExtentCenter`, as illustrated in Figure 8.

In the modeled EN sync logic, the machine is driven by an external timer modeled in P# (see §3.3). It prepares a sync report with `extCtr.GetSyncReport(...)`, and then asynchronously sends the report to `ExtentManager` using `PSharp.Send(...)`. The `ExtentNode` machine also includes failure-related logic (see §3.4).

```
// modeling timer expiration in P#
class Timer : Machine {
  Machine Target; // target machine

  [OnEvent(RepeatedEvent, GenerateTimerTick)]
  void GenerateTimerTick() {
    // nondeterministic choice controlled by P#
    if (PSharp.Nondet())
      PSharp.Send(Target, new TimerTickEvent());
    PSharp.Send(this, new RepeatedEvent()); // loop
  }
}
```

Figure 9: Modeling timer expiration using P#.

```
// machine for driving testing scenarios in vNext
class TestingDriver : Machine {
  private HashSet<Machine> ExtentNodes; // EN machines

  void InjectNodeFailure() {
    // nondeterministically choose an EN using P#
    var node = (Machine)PSharp.Nondet(ExtentNodes);
    // fail chosen EN
    PSharp.Send(node, new FailureEvent());
  }
}
```

Figure 10: Code snippet of the TestingDriver machine.

3.3 The Timer machine

System correctness should *not* hinge on the frequency of any individual timer. Hence, it makes sense to delegate all nondeterminism due to timing-related events to P#. To achieve this, all the timers inside ExtMgr are disabled (see Figure 5), and the EN expiration loop and the extent repair loop are driven instead by timers modeled in P#, an approach also used in previous work [10].[3] Similarly, `ExtentNode` machines do *not* have internal timers either. Their periodic heartbeats and sync reports are also driven by timers modeled in P#.

Figure 9 shows the `Timer` machine in the test harness. `Timer` invokes the P# method `Nondet()`, which generates a nondeterministic choice controlled by the P# runtime. Using `Nondet()` allows the machine to nondeterministically send a timeout event to its target (the `ExtentManager` or `ExtentNode` machines). The P# testing engine has the freedom to schedule arbitrary interleavings between these timeout events and all other regular system events.

3.4 The TestingDriver machine

The `TestingDriver` machine drives two testing scenarios. In the first scenario, `TestingDriver` launches one `ExtentManager` and three `ExtentNode` machines, with a single extent on one of the ENs. It then waits for the extent to be replicated at the remaining ENs. In the second testing scenario, `TestingDriver` fails one of the

[3] We had to make a minor change to the real ExtMgr code to facilitate modeling: we added the `DisableTimer` method, which disables the real ExtMgr timer so that it can be replaced with the P# timer.

ExtentNode machines and launches a new one. It then waits for the extent to be repaired on the newly launched ExtentNode machine.

Figure 10 illustrates how TestingDriver leverages P# to inject nondeterministic failures. It uses Nondet() to nondeterministically choose an ExtentNode machine, and then sends a FailureEvent to the chosen machine to emulate an EN failure. As shown in the earlier Figure 8, the chosen ExtentNode machine processes the FailureEvent, notifies the liveness monitor of its failure (see §3.5) and terminates itself by invoking the P# method Halt(). P# not only enumerates interleavings of asynchronous event handlers, but also the values returned by calls to Nondet(), thus enumerating different failure scenarios.

3.5 The RepairMonitor liveness monitor

RepairMonitor is a P# liveness monitor (see §2.5) that transitions between a cold and a hot state. Whenever an EN fails, the monitor is notified with an ENFailedEvent event. As soon as the number of extent replicas falls below a specified target (three replicas in the current P# test harness), the monitor transitions into the hot *repairing* state, waiting for all missing replicas to be repaired. Whenever an extent replica is repaired, RepairMonitor is notified with an ExtentRepairedEvent event. When the replica number reaches again the target, the monitor transitions into the cold *repaired* state, as illustrated in the code snippet of Figure 11.

In the extent repair testing scenarios, RepairMonitor checks that it should *always eventually* end up in the cold state. Otherwise, RepairMonitor is stuck in the hot state for *infinitely* long. This indicates that the corresponding execution sequence results in an extent replica never being repaired, which is a liveness bug.

3.6 Liveness Bug in Azure Storage vNext

It took less than ten seconds for the P# testing engine to report the first occurrence of a liveness bug in vNext (see §6). Upon examining the debug trace, the developers of vNext were able to quickly confirm the bug.

The original P# trace did not include sufficient details to allow the developers to identify the root cause of the problem. Fortunately, running the test harness took very little time, so the developers were able to quickly iterate and add more refined debugging outputs in each iteration. After several iterations, the developers were able to pinpoint the exact culprit and immediately propose a solution for fixing the bug. Once the proposed solution was implemented, the developers ran the test harness again. No bugs were found during 100,000 executions, a process that only took a few minutes.

```
class RepairMonitor : Monitor {
 private HashSet<Machine> ExtentNodesWithReplica;

 // cold state: repaired
 cold state Repaired {
  [OnEvent(ENFailedEvent, ProcessENFailure)]
  void ProcessENFailure(ExtentNodeMachine en) {
    ExtentNodesWithReplica.Remove(en);
    if (ReplicaCount < Harness.REPLICA_COUNT_TARGET)
     jumpto Repairing;
  }
 }

 // hot state: repairing
 hot state Repairing {
  [OnEvent(ExtentRepairedEvent, ProcessRepairCompletion)]
  void ProcessRepairCompletion(ExtentNodeMachine en) {
   ExtentNodesWithReplica.Add(en);
   if (ReplicaCount == Harness.REPLICA_COUNT_TARGET)
    jumpto Repaired;
  }
 }
}
```

Figure 11: The RepairMonitor liveness monitor.

The liveness bug occurs in the second testing scenario, where the TestingDriver machine fails one of the ExtentNode machines and launches a new one. RepairMonitor transitions to the hot repairing state and is stuck in the state for infinitely long.

The following is one particular execution sequence resulting in this liveness bug: (i) EN_0 fails and is detected by the EN expiration loop; (ii) EN_0 is removed from ExtentNodeMap; (iii) ExtentCenter is updated and the replica count drops from 3 (which is the target) to 2; (iv) ExtMgr receives a sync report from EN_0; (v) the extent center is updated and the replica count increases again from 2 to 3. This is problematic since the replica count is equal to the target, which means that the extent repair loop will never schedule any repair task. At the same time, there are only two *true* replicas in the system, which is one less than the target. This execution sequence leads to one missing replica; repeating this process two more times would result in all replicas missing, but ExtMgr would still think that all replicas are healthy. If released to production, this bug would have caused a very serious incident of customer data unavailability.

The culprit is in step (iv), where ExtMgr receives a sync report from EN_0 after deleting the EN. This interleaving is exposed quickly by P#'s testing engine that has the control to arbitrarily interleave events. It may also occur, albeit much less frequently, during stress testing due to messages being delayed in the network. This explains why the bug only occurs from time to time during stress testing and requires long executions to manifest. In contrast, P# allows the bug to manifest quickly, the developers to iterate rapidly, the culprit to be identified promptly, and the fix to be tested effectively and thoroughly, all of which have the potential to vastly increase the productivity of distributed storage system development.

4 Case Study: Live Table Migration

Live Table Migration (MigratingTable) is a library designed to transparently migrate a key-value data set between two Microsoft Azure tables [24] (called the *old* table and the *new* table, or collectively the *backend* tables) while an application is accessing this data set. The MigratingTable testing effort differs from the vNext effort in two significant ways: the P# test harness was developed along with the system rather than later; and it checks complete compliance with an interface specification rather than just a single liveness property. Indeed, the P# test caught bugs throughout the development process (see §6).

During migration, each application process creates its own MigratingTable instance (MT) that refers to the same backend tables (BTs). The application performs all data access via the MT, which provides an interface named `IChainTable` similar to that of an Azure table (the MT assumes that the BTs provide the same interface via an adapter). A *migrator* job moves the data in the background. In the meantime, each *logical* read and write operation issued to an MT is implemented via a sequence of *backend* operations on the BTs according to a custom protocol. The protocol is designed to guarantee that the output of the logical operations complies with the `IChainTable` specification, as if all the operations were performed on a single *virtual table* (VT). The goal of using P# was to systematically test this property.

There are two main challenges behind testing MigratingTable: (i) the system is highly concurrent; and (ii) the logical operations accept many parameters that affect the behavior in different ways. The developers could have chosen specific input sequences, but they were not confident that these sequences would cover the combinations of parameters that might trigger bugs. Instead, they used the P# `Nondet()` method to choose all of the parameters independently within certain limits. They issued the same operations to the MTs and to a *reference table* (RT) running a reference implementation of the `IChainTable` specification, and compared the output. This reference implementation was reused for the BTs, since the goal was not to test the real Azure tables.

The complete test environment is shown in Figure 12. It consists of a `Tables` machine, which contains the BTs and RT, and serializes all operations on these tables; a set of `Service` machines that contain identically configured MTs; and a `Migrator` machine that performs the background migration. Each `Service` machine issues a random sequence of logical operations to its MT, which performs the backend operations on the BTs via P# events. The developers instrumented the MTs to report the *linearization point* of each logical operation, i.e., the time at which it takes effect on the VT, so the test harness could perform the operation on the RT at the same time. (More precisely, after processing each backend operation, the `Tables` machine enters a P# state that blocks all further work until the MT reports whether the backend operation was the linearization point and, if so, the logical operation has been completed on the RT. This way, the rest of the system never observes the RT to be out of sync with the VT.) For further implementation details of MigratingTable and its test harness, we refer the reader to the source code repository [36].

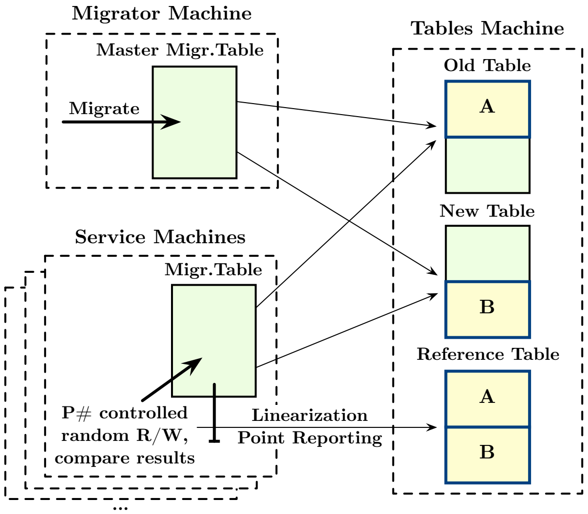


Figure 12: The test environment of MigratingTable (each box with a dotted line represents one P# machine).

5 Case Study: Azure Service Fabric

Azure Service Fabric (or Fabric for short) is a platform and API for creating reliable services that execute on a cluster of machines. The developer writes a service that receives requests (e.g. HTTP requests from a client) and mutates its state based on these requests. To make a user service reliable, Fabric launches several replicas of the service, where each replica runs as a separate process on a different node in the cluster. One replica is selected to be the *primary* which serves client requests; the remaining replicas are *secondaries*. The primary forwards state-mutating operations to the secondaries so that all replicas eventually have the same state. If the primary fails, Fabric elects one of the secondaries to be the new primary, and then launches a new secondary, which will receive a copy of the state of the new primary in order to "catch up" with the other secondaries. Fabric services are complex asynchronous and distributed applications, and are thus challenging to test.

Our primary goal was to create a P# model of Fabric (where all the Fabric asynchrony is captured and controlled by the P# runtime) to allow thorough testing of

Fabric services. The model was written once to include all behaviors of Fabric, including simulating failures and recovery, so that it can be reused repeatedly to test many Fabric services. This was the largest modeling exercise among the case studies, but the cost was amortized across testing multiple services. This is analogous to prior work on device driver verification [2], where the cost of developing a model of the Windows kernel was amortized across testing multiple device drivers. Note that we model the lowest Fabric API layer (`Fabric.dll`), which is currently not documented for use externally; we target internally-developed services that use this API.

Using P# was very helpful in debugging the model itself. To systematically test the model, we wrote a simple service in P# that runs on our P# Fabric model. We tested a scenario where the primary replica fails at some nondeterministic point. One bug that we found occurred when the primary failed as a new secondary was about to receive a copy of the state; the secondary was then elected to be the new primary and yet, because the secondary stopped waiting for a copy of the state, it was then "promoted" to be an *active* secondary (one that has caught up with the other secondaries). This caused an assertion failure in our model, because only a secondary can be promoted to an active secondary, which allowed us to detect and fix this incorrect behavior.

The main system that we tested using our P# Fabric model is *CScale* [12], a big data stream processing system. Supporting CScale required significant additions to the model, making it much more feature-complete. CScale chains multiple Fabric services, which communicate via remote procedure calls (RPCs). To close the system, we modeled RPCs using `PSharp.Send(...)`. Thus, we converted a distributed system that uses both Fabric and its own network communication protocol into a closed, single-process system. A key challenge in our work was to thoroughly test CScale despite the fact that it uses various synchronous and asynchronous APIs besides RPCs. This work is still in-progress. However, we were able to find a `NullReferenceException` bug in CScale by running it against our Fabric model. The bug has been communicated to the developers of CScale, but we are still awaiting a confirmation.

6 Quantifying the Cost of Using P#

We report our experience of applying P# on the three case studies discussed in this paper. We aim to answer the following two questions:

1. How much human effort was spent in modeling the environment of a distributed system using P#?

2. How much computational time was spent in systematically testing a distributed system using P#?

	System		**P# Test Harness**			
System-under-test	**#LoC**	**#B**	**#LoC**	**#M**	**#ST**	**#AH**
vNext Extent Manager	19,775	1	684	5	11	17
MigratingTable	2,267	11	2,275	3	5	10
Fabric User Service	31,959	1⋆	6,534	13	21	87

Table 1: Statistics from modeling the environment of the three Microsoft Azure-based systems under test. The (⋆) denotes "awaiting confirmation".

6.1 Cost of environment modeling

Environment modeling is a core activity of using P#. It is required for *closing* a system to make it amenable to systematic testing. Table 1 presents program statistics for the three case studies. The columns under "System" refer to the real system-under-test, while the columns under "P# Test Harness" refer to the test harness written in P#. We report: lines of code for the system-under-test (#LoC); number of bugs found in the system-under-test (#B); lines of P# code for the test harness (#LoC); number of machines (#M); number of state transitions (#ST); and number of action handlers (#AH).

Modeling the environment of the Extent Manager in the Azure Storage vNext system required approximately two person weeks of part-time developing. The P# test harness for this system is the smallest (in lines of code) from the three case studies. This was because this modeling exercise aimed to reproduce the particular liveness bug that was haunting the developers of vNext.

Developing both MigratingTable and its P# test harness took approximately five person weeks. The harness was developed in parallel with the actual system. This differs from the other two case studies, where the modeling activity occurred independently and after the development process.

Modeling Fabric required approximately five person months, an effort undertaken by the authors of P#. In contrast the other two systems discussed in this paper were modeled and tested by their corresponding developers. Although modeling Fabric required a significant amount of time, it is a one-time effort, which only needs incremental refinement with each release.

6.2 Cost of systematic testing

Using P# we managed to uncover 8 serious bugs in our case studies. As discussed earlier in the paper, these bugs were hard to find with traditional testing techniques, but P# managed to uncover and reproduce them in a small setting. According to the developers, the P# traces were useful, as it allowed them to understand the bugs and fix them in a timely manner. After all the discovered bugs

		P# Random Scheduler			P# Priority-based Scheduler		
CS	**Bug Identifier**	**BF?**	**Time to bug (s)**	**#NDC**	**BF?**	**Time to bug (s)**	**#NDC**
1	ExtentNodeLivenessViolation	✓	10.56	9,000	✓	10.77	9,000
2	QueryAtomicFilterShadowing	✓	157.22	165	✓	350.46	108
2	QueryStreamedLock	✓	2,121.45	181	✓	6.58	220
2	QueryStreamedBackUpNewStream	✗	-	-	✓	5.95	232
2	DeleteNoLeaveTombstonesEtag	✗	-	-	✓	4.69	272
2	DeletePrimaryKey	✓	2.72	168	✓	2.37	171
2	EnsurePartitionSwitchedFromPopulated	✓	25.17	85	✓	1.57	136
2	TombstoneOutputETag	✓	8.25	305	✓	3.40	242
2	QueryStreamedFilterShadowing	◇	0.55	79	◇	0.41	79
∗2	MigrateSkipPreferOld	✗	-	-	◇	1.13	115
∗2	MigrateSkipUseNewWithTombstones	✗	-	-	◇	1.16	120
∗2	InsertBehindMigrator	◇	0.32	47	◇	0.31	47

Table 2: Results from running the P# random and priority-based systematic testing schedulers for 100,000 executions. We report: whether the bug was found (BF?) (✓ means it was found, ◇ means it was found only using a custom test case, and ✗ means it was not found); time in seconds to find the bug (Time to Bug); and number of nondeterministic choices made in the first execution that found the bug (#NDC).

were fixed, we added flags to allow them to be individually re-introduced, for purposes of evaluation.

Table 2 presents the results from running the P# systematic testing engine on each case study with a re-introduced bug. The CS column shows which case study corresponds to each bug: "1" is for the Azure Storage vNext; and "2" is for MigratingTable. We do not include the Fabric case study, as we are awaiting confirmation of the found bug (see §5). We performed all experiments on a 2.50GHz Intel Core i5-4300U CPU with 8GB RAM running Windows 10 Pro 64-bit. We configured the P# systematic testing engine to perform 100,000 executions. All reported times are in seconds.

We implemented two different schedulers that are responsible for choosing the next P# machine to execute in each scheduling point: a *random* scheduler, which randomly chooses a machine from a list of enabled[4] machines; and a *randomized priority-based* [4] scheduler, which always schedules the highest priority enabled machine (these priorities change at random points during execution, based on a random distribution). We decided to use these two schedulers, because random scheduling has proven to be efficient for finding concurrency bugs [43, 9]. The random seed for the schedulers was generated using the current system time. The priority-based scheduler was configured with a budget of 2 random machine priority change switches per execution.

For the vNext case study, both schedulers were able to reproduce the ExtentNodeLivenessViolation bug within 11 seconds. The reason that the number of nondeterministic choices made in the buggy execution is much higher than the rest of the bugs is that ExtentNodeLivenessViolation is a liveness bug: as discussed in §2.5 we leave the program to run for a long time before checking if the liveness property holds.

For the MigratingTable case study, we evaluated the P# test harness of §4 on eleven bugs, including eight *organic* bugs that actually occurred in development and three *notional* bugs (denoted by ∗), which are other code changes that we deemed interesting ways of making the system incorrect. The harness found seven of the organic bugs, which are listed first in Table 2. The remaining four bugs (marked ◇) were not caught with our default test harness in the 100,000 executions. We believe this is because the inputs and schedules that trigger them are too rare in the used distribution. To confirm this, we wrote a custom test case for each bug with a specific input that triggers it and were able to quickly reproduce the bugs; the table shows the results of these runs. Note that the random scheduler only managed to trigger seven of the MigratingTable bugs; we had to use the priority-based scheduler to trigger the remaining four bugs.

The QueryStreamedBackUpNewStream bug in MigratingTable, which was found using P#, stands out because it reflects a type of oversight that can easily occur as systems evolve. This bug is in the implementation of a *streaming read* from the virtual table, which should return a stream of all rows in the table sorted by key. The essential implementation idea is to perform streaming reads from both backend tables and merge the results. According to the `IChainTable` specification, each row read from a stream may reflect the state of the table at any time between when the stream was started and the row was read. The developers sketched a proof that the

[4] A P# machine is considered enabled when it has at least one event in its queue waiting to be dequeued and handled.

merging process would preserve this property as long as the migrator was only copying rows from the old table to the new table. But when support was added for the migrator to delete the rows from the old table after copying, it became possible for the backend streams to see the deletion of a row from the old table but not its insertion into the new table, even though the insertion happens first, and the row would be missed.

The P# test discovered the above bug in a matter of seconds. The MigratingTable developers spent just 10 minutes analyzing the trace to diagnose what was happening, although admittedly, this was after they added MigratingTable-specific trace information and had several days of experience analyzing traces. Out of the box, P# traces include only machine- and event-level information, but it is easy to add application-specific information, and we did so in all of our case studies.

7 Related Work

Most related to our work are model checking [15] and systematic concurrency testing [38, 11, 43], two powerful techniques that have been widely used in the past for finding Heisenbugs in the actual implementation of distributed systems [25, 48, 47, 18, 21, 45, 19, 31].

State-of-the-art model checkers, such as MODIST [48] and dBug [45], typically focus on testing entire, often *unmodified*, distributed systems, an approach that easily leads to state-space explosion. DEMETER [21], built on top of MODIST, aims to reduce the state-space when exploring unmodified distributed systems. DEMETER explores individual components of a large system in isolation, and then dynamically extracts interface behavior between components to perform a global exploration. In contrast, we try to offer a more pragmatic approach for handling state-space explosion. We first *partially* model a distributed system using P#. Then, we systematically test the actual implementation of each system component against its P# test harness. Our approach aims to enhance unit and integration testing, techniques widely used in production, where only individual or a small number of components are tested at each time.

SAMC [31] offers a way of incorporating application-specific information during systematic testing to reduce the set of interleavings that the tool has to explore. Such techniques based on partial-order reduction [14, 13] are complementary to our approach: P# could use them to reduce the exploration state-space. Likewise, other tools can use language technology like P# to write models and reduce the complexity of the system-under-test.

MACEMC [25] is a model checker for distributed systems written in the MACE [26] language. The focus of MACEMC is to find liveness property violations using an algorithm based on bounded random walk, combined with heuristics. Because MACEMC can only test systems written in MACE, it cannot be easily used in an industrial setting. In contrast, P# can be applied on legacy code written in C#, a mainstream language.

Formal methods have been successfully used in industry to verify the correctness of distributed protocols. A notable example is the use of TLA+ [29] by the Amazon Web Services team [39]. TLA+ is an expressive formal specification language that can be used to design and verify concurrent programs via model checking. A limitation of TLA+, as well as other similar specification languages, is that they are applied on a model of the system and not the actual system. Even if the model is verified, the gap between the real-world implementation and the verified model is still significant, so implementation bugs are still a realistic concern.

More recent formal approaches include the Verdi [46] and IronFleet [22] frameworks. In Verdi, developers can write and verify distributed systems in Coq [3]. After the system has been successfully verified, Verdi translates the Coq code to OCaml, which can be then compiled for execution. Verdi does not currently support detecting liveness property violations, an important class of bugs in distributed systems. In IronFleet, developers can build a distributed system using the Dafny [34] language and program verifier. Dafny verifies system correctness using the Z3 [8] SMT solver, and finally compiles the verified system to a .NET executable. In contrast, P# performs bounded testing on a system already written in .NET, which in our experience lowers the bar for adoption by engineering teams.

8 Conclusion

We presented a new methodology for testing distributed systems. Our approach involves using P#, an extension of the C# language that provides advanced modeling, specification and systematic testing capabilities. We reported experience on applying P# on three distributed storage systems inside Microsoft. Using P#, the developers of these systems found, reproduced, confirmed and fixed numerous bugs.

9 Acknowledgments

We thank our shepherd Haryadi Gunawi for his valuable guidance that significantly improved the paper, and the anonymous reviewers for their constructive comments. We also thank Ankush Desai from UC Berkeley, and Rich Draves, David Goebel, David Nichols and SW Worth from Microsoft, for their valuable feedback and discussions at various stages of this work. We acknowledge that this research was partially supported by a gift from Intel Corporation.

References

[1] AMAZON. Summary of the AWS service event in the US East Region. `http://aws.amazon.com/message/67457/`, 2012.

[2] BALL, T., LEVIN, V., AND RAJAMANI, S. K. A decade of software model checking with SLAM. *Communications of the ACM 54*, 7 (2011), 68–76.

[3] BARRAS, B., BOUTIN, S., CORNES, C., COURANT, J., FILLIATRE, J.-C., GIMENEZ, E., HERBELIN, H., HUET, G., MUNOZ, C., MURTHY, C., ET AL. The Coq proof assistant reference manual: Version 6.1. `https://hal.inria.fr/inria-00069968/`, 1997.

[4] BURCKHARDT, S., KOTHARI, P., MUSUVATHI, M., AND NAGARAKATTE, S. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (2010), ACM, pp. 167–178.

[5] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., ET AL. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 143–157.

[6] CAVAGE, M. There's just no getting around it: you're building a distributed system. *ACM Queue 11*, 4 (2013), 30–41.

[7] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp. 398–407.

[8] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer-Verlag, pp. 337–340.

[9] DELIGIANNIS, P., DONALDSON, A. F., KETEMA, J., LAL, A., AND THOMSON, P. Asynchronous programming, analysis and testing with state machines. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), ACM, pp. 154–164.

[10] DESAI, A., JACKSON, E., PHANISHAYEE, A., QADEER, S., AND SESHIA, S. A. Building reliable distributed systems with P. Tech. Rep. UCB/EECS-2015-198, EECS Department, University of California, Berkeley, Sep 2015.

[11] EMMI, M., QADEER, S., AND RAKAMARIĆ, Z. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2011), ACM, pp. 411–422.

[12] FALEIRO, J., RAJAMANI, S., RAJAN, K., RAMALINGAM, G., AND VASWANI, K. CScale: A programming model for scalable and reliable distributed applications. In *Proceedings of the 17th Monterey Conference on Large-Scale Complex IT Systems: Development, Operation and Management* (2012), Springer-Verlag, pp. 148–156.

[13] FLANAGAN, C., AND GODEFROID, P. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2005), ACM, pp. 110–121.

[14] GODEFROID, P. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, vol. 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[15] GODEFROID, P. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1997), ACM, pp. 174–186.

[16] GRAY, J. Why do computers stop and what can be done about it? In *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems* (1986), IEEE, pp. 3–12.

[17] GREENBERG, A. SDN for the Cloud. Keynote in the 2015 ACM Conference on Special Interest Group on Data Communication, 2015.

[18] GUERRAOUI, R., AND YABANDEH, M. Model checking a networked system without the network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011), USENIX, pp. 225–238.

[19] GUNAWI, H. S., DO, T., JOSHI, P., ALVARO, P., HELLERSTEIN, J. M., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., SEN, K., AND BORTHAKUR, D. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011), USENIX, pp. 238–252.

[20] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYATAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing* (2014), ACM.

[21] GUO, H., WU, M., ZHOU, L., HU, G., YANG, J., AND ZHANG, L. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 265–278.

[22] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM.

[23] HENRY, A. Cloud storage FUD: Failure and uncertainty and durability. Keynote in the 7th USENIX Conference on File and Storage Technologies, 2009.

[24] HOGG, J. Azure storage table design guide: Designing scalable and performant tables. `https://azure.microsoft.com/en-us/documentation/articles/storage-table-design-guide/`, 2015.

[25] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation* (2007), USENIX, pp. 18–18.

[26] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007), ACM, pp. 179–188.

[27] LAGUNA, I., AHN, D. H., DE SUPINSKI, B. R., GAMBLIN, T., LEE, G. L., SCHULZ, M., BAGCHI, S., KULKARNI, M., ZHOU, B., CHEN, Z., AND QIN, F. Debugging high-performance computing applications at massive scales. *Communications of the ACM 58*, 9 (2015), 72–81.

[28] LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering 3*, 2 (1977), 125–143.

[29] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (1994), 872–923.

[30] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133–169.

[31] LEESATAPORNWONGSA, T., HAO, M., JOSHI, P., LUKMAN, J. F., AND GUNAWI, H. S. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), USENIX, pp. 399–414.

[32] LEESATAPORNWONGSA, T., LUKMAN, J. F., LU, S., AND GUNAWI, H. S. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ACM.

[33] LEIJEN, D., SCHULTE, W., AND BURCKHARDT, S. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (2009), ACM, pp. 227–242.

[34] LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (2010), Springer-Verlag, pp. 348–370.

[35] MADDOX, P. Testing a distributed system. *ACM Queue 13*, 7 (2015), 10–15.

[36] MCCUTCHEN, M. MigratingTable source repository. `https://github.com/mattmccutchen/MigratingTable`, 2015.

[37] MUSUVATHI, M., AND QADEER, S. Fair stateless model checking. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2008), ACM, pp. 362–371.

[38] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), USENIX, pp. 267–280.

[39] NEWCOMBE, C., RATH, T., ZHANG, F., MUNTEANU, B., BROOKER, M., AND DEARDEUFF, M. How amazon web services uses formal methods. *Communications of the ACM 58*, 4 (2015), 66–73.

[40] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference* (2014), USENIX, pp. 305–319.

[41] PNUELI, A. The temporal logic of programs. In *Proceedings of the Foundations of Computer Science* (1977), pp. 46–57.

[42] TASSEY, G. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, Planning Report 02-3* (2002).

[43] THOMSON, P., DONALDSON, A. F., AND BETTS, A. Concurrency testing using schedule bounding: An empirical study. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2014), ACM, pp. 15–28.

[44] TREYNOR, B. GoogleBlog – Today's outage for several Google services. `http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html`, 2014.

[45] ŠIMŠA, J., BRYANT, R., AND GIBSON, G. dBug: Systematic testing of unmodified distributed and multi-threaded systems. In *Proceedings of the 18th International SPIN Conference on Model Checking Software* (2011), Springer-Verlag, pp. 188–193.

[46] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), ACM, pp. 357–368.

[47] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (2009), USENIX, pp. 229–244.

[48] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (2009), USENIX, pp. 213–228.

The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments

Mingzhe Hao[†], Gokul Soundararajan[*], Deepak Kenchammana-Hosekote[*],
Andrew A. Chien[†], and Haryadi S. Gunawi[†]

[†]University of Chicago [*]NetApp, Inc.

Abstract

We study storage performance in over 450,000 disks and 4,000 SSDs over 87 days for an overall total of 857 million (disk) and 7 million (SSD) drive hours. We find that storage performance instability is not uncommon: 0.2% of the time, a disk is more than 2x slower than its peer drives in the same RAID group (and 0.6% for SSD). As a consequence, disk and SSD-based RAIDs experience at least one slow drive (i.e., storage tail) 1.5% and 2.2% of the time. To understand the root causes, we correlate slowdowns with other metrics (workload I/O rate and size, drive event, age, and model). Overall, we find that the primary cause of slowdowns are the internal characteristics and idiosyncrasies of modern disk and SSD drives. We observe that storage tails can adversely impact RAID performance, motivating the design of tail-tolerant RAID. To the best of our knowledge, this work is the most extensive documentation of storage performance instability in the field.

1 Introduction

Storage, the home of Big Data, has grown enormously over the past decade [21]. This year Seagate projects to ship more than 240 exabytes of disk drives [20], SSD market has doubled in recent years [32], and data stored in the cloud has also multiplied almost exponentially every year [10]. In a world of continuous collection and analysis of Big Data, storage performance is critical for many applications. Modern applications particularly demand low and predictable response times, giving rise to stringent performance SLOs such as "99.9% of all requests must be answered within 300ms" [15, 48]. Performance instability that produces milliseconds of delay lead to violations of such SLOs, degrading user experience and impacting revenues negatively [11, 35, 44].

A growing body of literature studies the general problem of performance instability in large-scale systems, specifically calling out the impact of stragglers on tail latencies [7, 13, 14, 34, 45, 50, 52, 54, 56]. Stragglers often arise from contention for shared local resources (*e.g.*, CPU, memory) and global resources (*e.g.*, network switches, back-end storage), background daemons, scheduling, power limits and energy management, and many others. These studies are mostly performed at server, network, or remote (cloud) storage levels.

To date, we find no systematic, *large-scale studies* of performance instability in *storage devices* such as disks and SSDs. Yet, mounting anecdotal evidence of disk and SSD performance instability in the field continue to appear in various forums (§2). Such ad-hoc information is unable to answer quantitatively key questions about drive performance instability, questions such as: How much slowdown do drives exhibit? How often does slowdown occur? How widespread is it? Does slowdown have temporal behavior? How long can slowdown persist? What are the potential root causes? What is the impact of tail latencies from slow drives to the RAID layer? Answers to these questions could inform a wealth of storage systems research and design.

To answer these questions, we have performed the largest empirical analysis of storage performance instability. Collecting *hourly* performance logs from customer deployments of 458,482 disks and 4,069 SSDs spanning on average 87 day periods, we have amassed a dataset that covers 857 million hours of disk and 7 million hours of SSD field performance data.

Uniquely, our data includes drive-RAID relationships, which allows us to compare the performance of each drive (D_i) to that of peer drives in the same RAID group ($i = 1..N$). The RAID and file system architecture in our study (§3.1) expects that the performance of every drive (specifically, hourly average latency L_i) is similar to peer drives in the same RAID group.

Our primary metric, *drive slowdown ratio* (S_i), the fraction of a drive's latency (L_i) over the median latency of the RAID group ($median(L_{1..N})$), captures deviation from the assumption of homogeneous drive performance. Assuming that most workloads are balanced across all the data drives, a normal drive should not be much slower than the other drives. Therefore, we define "**<u>slow</u>**" (unstable) drive hour when $S_i \geq 2$ (and "stable" the otherwise). Throughout the paper, we use 2x and occasionally 1.5x slowdown threshold to classify drives as slow.

In the following segment, we briefly summarize the findings from our large-scale analysis.

(i) Slowdown occurrences (§4.1.1): Disks and SSDs are slow ($S_i \geq 2$) for 0.22% and 0.58% of *drive hours* in our study. With a tighter $S_i \geq 1.5$ threshold, disks and SSDs are slow for 0.69% and 1.27% of disk hours respectively. Consequently, stable latencies at 99.9th percentile are hard to achieve in today's storage drives. Slowdowns can also be extreme (*i.e.*, long tails); we observe several slowdown incidents as large as 2-4 orders of magnitude.

(ii) Tail hours and RAID degradation (§4.1.2): A slow drive can often make an entire RAID perform poorly. The observed instability causes RAIDs to suffer 1.5% and 2.2% of *RAID hours* with at least one slow drive (*i.e.*, *"tail hours"*). Using 1.5x slowdown threshold, the numbers are 4.6% and 4.8%. As a consequent, stable latencies at 99th percentile (or 96th with 1.5x threshold) are impossible to guarantee in current RAID deployments. Workload performance (especially full-stripe balanced workload) will suffer as a consequence of RAID tails. In our dataset, we observe that RAID throughput can degrade during stable to tail hours (§4.4.1).

(iii) Slowdown temporal behavior and extent (§4.1.3, §4.1.4): We find that slowdown often persists; 40% and 35% of slow disks and SSDs respectively remain unstable for more than one hour. Slowdown periods exhibit temporal locality; 90% of disk and 85% of SSD slowdowns occur on the same day of the previous occurrence. Finally, slowdown is widespread in the drive population; our study shows 26% of disks and 29% of SSDs have experienced at least one slowdown occurrence.

(iv) Workload analysis (§4.2): Drive slowdowns are often blamed on unbalanced workloads (*e.g.*, a drive is busier than others). Our findings refute this, showing that more than 95% of slowdown periods *cannot* be attributed to I/O size or rate imbalance.

(v) "The fault is (likely) in our drives": We find that older disks exhibit more slowdowns (§4.3.2) and MLC flash drives exhibit more slowdowns than SLC drives (§4.3.3). Overall, evidence suggests that most slowdowns are caused by internal characteristics of modern disk and SSD drives.

In summary, drive performance instability means the homogeneous performance assumption of traditional RAID is no longer accurate. Drive slowdowns can appear at different times, persist, disappear, and recur again. Their occurrence is "silent"—not accompanied by observable drive events (§4.3.1). Most importantly, workload imbalance is not a major root cause (§4.2). Replacing slow drives is not a popular solution (§4.4.2-§4.4.3), mainly because slowdowns are often transient and drive replacement is expensive in terms of hardware and RAID rebuild costs.

(vi) The need for tail-tolerant RAID: All of the reasons above point out that file and RAID systems are now faced with more responsibilities. Not only must they handle well-known faults such as latent sector errors and corruptions, now they must mask storage tail latencies as well. Therefore, there is an opportunity to create "tail tolerant" RAID that can mask storage tail latencies online in deployment.

In the following sections, we present further motivation (§2), our methodology (§3), the main results (§4), an opportunity assessment of tail-tolerant RAID (§5), discussion (§6), related work (§7) and conclusion (§8).

2 Motivational Anecdotes

Our work is highly motivated by the mounting anecdotes of performance instability at the drive level. In the past several years, we have collected facts and anecdotal evidence of storage "limpware" [16, 26] from literature, online forums supported by various storage companies, and conversations with large-scale datacenter operators as well as product teams. We found many reports of storage performance problems due to various faults, complexities and idiosyncrasies of modern storage devices, as we briefly summarize below.

Disk: Magnetic disk drives can experience performance faults from various root causes such as mechanical wearout (*e.g.*, weak head [1]), sector re-reads due to media failures such as corruptions and sector errors [2], overheat from broken cooling fans [3], gunk spilling from actuator assembly and accumulating on disk head [4], firmware bugs [41], RAID controller defects [16, 47], and vibration from bad disk drive packaging, missing screws, earthquakes, and constant "noise" in data centers [17, 29]. All these problems can reduce disk bandwidth by 10-80% and increase latency by seconds. While the problems above can be considered as performance "faults", current generation of disks begin to induce performance instability "by default" (*e.g.*, with adaptive zoning and Shingled-Magnetic Recording technologies [5, 18, 33]).

SSD: The pressure to increase flash density translates to more internal SSD complexities that can induce performance instability. For example, SSD garbage collection, a well-known culprit, can increase latency by a factor of 100 [13]. Programming MLC cells to different states (*e.g.*, 0 vs. 3) may require different numbers of iterations due to different voltage thresholds [51]. The notion of "fast" and "slow" pages exists within an SSD; programming a slow page can be 5-8x slower compared to

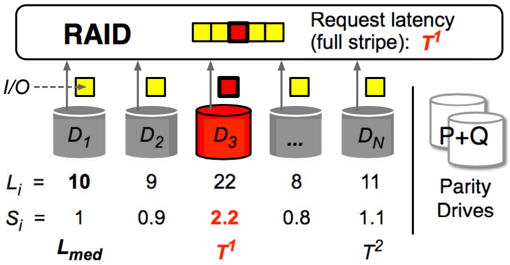


Figure 1: **Stable and slow drives in a RAID group.**

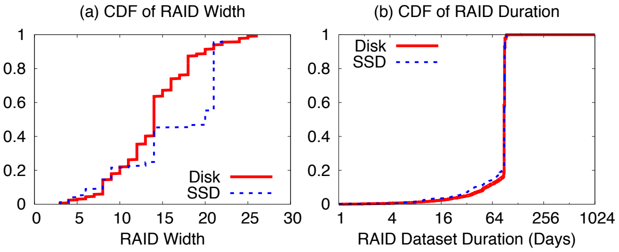


Figure 2: **RAID width and dataset duration.**

	Disk	SSD
RAID groups	38,029	572
Data drives per group	3-26	3-22
Data drives	458,482	4,069
Duration (days)	1-1470	1-94
Drive hours	857,183,442	7,481,055
Slow drive hours (§4.1.1)	1,885,804	43,016
Slow drive hours (%)	0.22	0.58
RAID hours	72,046,373	1,072,690
Tail hours (§4.1.2)	1,109,514	23,964
Tail hours (%)	1.54	2.23

Table 1: **Dataset summary.**

programming fast page [23]. As device wears out, breakdown of gate oxide will allow charge moves across gate easily, resulting in faster programming (10-50%), but also higher chance of corruption [22]. ECC correction, read disturb, and read retry are also factors of instability [19]. Finally, SSD firmware bugs can cause significant performance faults (*e.g.*, 300% bandwidth degradation in a Samsung firmware problem [49]).

Although the facts and anecdotes above are crucial, they do not provide empirical evidence that can guide the design of future storage systems. The key questions we raised in the introduction (slowdown magnitude, frequency, scope, temporal behavior, root causes, impacts, etc.) are still left unanswered. For this reason, we initiated this large-scale study.

3 Methodology

In this section, we describe the RAID systems in our study (§3.1), the dataset (§3.2), and the metrics we use to investigate performance instability (§3.3). The overall methodology is illustrated in Figure 1.

3.1 RAID Architecture

RAID group: Figure 1 provides a simple illustration of a RAID group. We study disk- and SSD-based RAID groups. In each group, disk or SSD devices are directly attached to a proprietary RAID controller. All the disk or SSD devices within a RAID group are homogeneous (same model, size, speed, etc.); deployment age can vary but most of them are the same.

RAID and file system design: The RAID layer splits each RAID request to per-drive I/Os. The size of a *per-drive I/O* (a square block in Figure 1) can vary from 4 to 256 KB; the storage stack breaks large I/Os to smaller I/Os with a maximum size of the processor cache size. Above the RAID layer runs a proprietary file system (not shown) that is highly tuned in a way that makes most of the RAID I/O requests cover the full stripe; most of the time the drives observe balanced workload.

RAID configuration: The RAID systems in our study use small chunk sizes (*e.g.*, 4 KB). More than 95% of the RAID groups use a custom version of RAID-6 where the parity blocks are not rotated; the parity blocks live in two separate drives (P and Q drives as shown in Figure 1). The other 4% use RAID-0 and 1% use RAID-4. We only select RAID groups that have at least three data drives ($D_1..D_N$ where $N \geq 3$ in Figure 1), mainly to allow us measure the relative slowdown compared to the median latency. Our dataset contains RAID groups with 3-26 data drives per group. Figure 2a shows the RAID width distribution (only data drives); wide RAID (*e.g.*, more than 8 data drives) is popular.

3.2 About the Dataset

Scale of dataset: A summary of our dataset is shown in Table 1. Our dataset contains 38,029 disk and 572 SSD groups within deployment duration of 87 days on average (Figure 2b). This gives us 72 and 1 million disk and SSD RAID hours to analyze respectively. When broken down to individual drives, our dataset contains 458,482 disks and 4069 SSDs. In total, we analyze 857 million and 7 million disk and SSD drive hours respectively.

Data collection: The performance and event logs we analyze come from production systems at customer sites. When the deployed RAID systems "call home", an auto-support system collects *hourly performance metrics* such as: average I/O latency, average latency per block, and number of I/Os and blocks received every hour. All these metrics are collected at the RAID layer. For each of

Label	Definition
	Measured metrics:
N	Number of *data* drives in a RAID group
D_i	Drive number within a RAID group; $i = 1..N$
L_i	Hourly average I/O latency observed at D_i
	Derived metrics:
L_{med}	Median latency; $L_{med} = Median\ of(L_{1..N})$
S_i	Latency slowdown of D_i compared to the median; $S_i = L_i/L_{med}$
T^k	The k-th largest slowdown ("k-th longest tail"); $T^1 = Max\ of\ (S_{1..N})$, $T^2 = 2nd\ Max\ of\ (S_{1..N})$, and so on
Stable	A stable drive hour is when $S_i < 2$
Slow	A slow drive hour is when $S_i \geq 2$
Tail	A tail hour implies a RAID hour with $T_i \geq 2$

Table 2: **Primary metrics.** *The table presents the metrics used in our analysis. The distribution of N is shown in Figure 2a. L_i, S_i and T^k are explained in Section 3.3.*

these metrics, the system separates read and write metrics. In addition to performance information, the system also records drive events such as response timeout, drive not spinning, unplug/replug events.

3.3 Metrics

Below, we first describe the metrics that are measured by the RAID systems and recorded in the auto-support system. Then, we present the metrics that we derived for measuring tail latencies (slowdowns). Some of the important metrics are summarized in Table 2.

3.3.1 Measured Metrics

Data drives (N): This symbol represents the number of data drives in a RAID group. Our study only includes data drives mainly because read operations only involve data drives in our RAID-6 with non-rotating parity. Parity drives can be studied as well, but we leave that for future work. In terms of write operations, the RAID small-write problem is negligible due to the file system optimizations (§3.1).

Per-drive hourly average I/O latency (L_i): Of all the metrics available from the auto-support system, we at the end only use the hourly average I/O latency (L_i) observed by every data drive (D_i) in every RAID group (i=1..N), as illustrated in Figure 1. We initially analyzed "throughput" metrics as well, but because the support system does not record per-IO throughput average, we cannot make an accurate throughput analysis based on hourly average I/O sizes and latencies.

Other metrics: We also use other metrics such as per-drive hourly average I/O rate (R_i) and size (Z_i), time of day, drive age, model, and events (replacements, unplug/replug, etc.), which we correlate with slowdown metrics to analyze root causes and impacts.

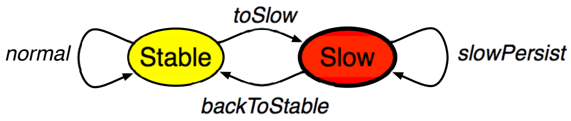


Figure 3: **Conceptual drive slowdown model.**

3.3.2 Derived Metrics

Slowdown (S_i): To measure tail latencies, RAID is a perfect target because it allows us to measure the relative slowdown of a drive compared to the other drives in the same group. Therefore, as illustrated in Figure 1, for every hour, we first measure the median group latency L_{med} from $L_{1..N}$ and then measure the hourly *slowdown* of a drive ($\underline{S_i}$) by comparing its latency with the median latency (L_i/L_{med}). The total number of S_i is essentially the "#drive hours" in Table 1. Our measurement of S_i is reasonably accurate because most of the workload is balanced across the data drives and the average latencies (L_i) are based on per-drive I/Os whose size variance is small (see §3.1).

Stable vs. slow drive hours: Assuming that most workload is balanced across all the data drives, a "stable" drive should not be much slower than other drives. Thus, we use a slowdown threshold of **2x** to differentiate slow drive hours ($S_i \geq 2$) and stable hours ($S_i < 2$). We believe 2x slowdown threshold is tolerant enough, but conversations with several practitioners suggest that a conservative 1.5x threshold will also be interesting. Thus, in some of our findings, we show additional results using 1.5x slowdown threshold.

Conceptually, drives appear to behave similar to a simple Markov model in Figure 3. In a given hour, a drive can be stable or slow. In the next hour, the drive can stay in the same or transition to the other condition.

Tails (T^k): For every hourly $S_{1..N}$, we derive the k-th largest slowdown represented as T^k. In this study, we only record the three largest slowdowns (T^1, T^2 and T^3). $\underline{T^1}$ represents the "longest tail" in a given RAID hour, as illustrated in Figure 1. The total number of T^1 is the "#RAID hours" in Table 1. The differences among T^k values will provide hints to the potential benefits of tail-tolerant RAID.

Tail hours: A "tail hour" implies a RAID hour that observes $T^1 \geq 2$ (*i.e.*, the RAID group observes at least one slow drive in that hour). This metric is important for full-stripe balanced workload where the performance will follow the longest tail (*i.e.*, the entire RAID slows down at the rate of T^1).

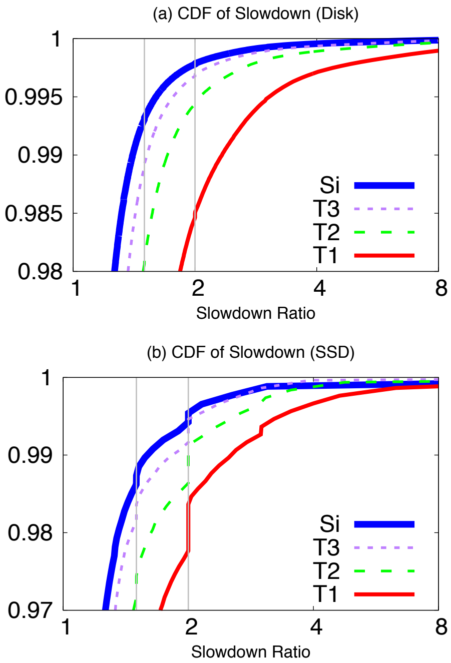


Figure 4: **Slowdown (S_i) and Tail (T^k) distributions (§4.1.1-§4.1.2).** *The figures show distributions of disk (top) and SSD (bottom) hourly slowdowns (S_i), including the three longest tails (T^{1-3}) as defined in Table 2. The y-axis range is different in each figure and the x-axis is in $\log_2$ scale. We plot two gray vertical lines representing 1.5x and 2x slowdown thresholds. Important slowdown-percentile intersections are listed in Table 3.*

From the above metrics, we can further measure other metrics such as slowdown intervals, extents, and repetitions. Overall, we have performed an in-depth analysis of all the measured and derived metrics. In many cases, due to space constraints, we aggregate some results whenever the sub-analysis does not show different behaviors. For example, we merge read and write slowdowns as I/O slowdown. In some graphs, we break down the slowdowns (*e.g.*, to 2-4x, 4-8x, 8-16x) if their characterizations are different.

4 Results

We now present the results of our study in four sets of analysis: slowdown and tail distributions and characteristics (§4.1), correlations between slowdowns and workload-related metrics (§4.2) and other available metrics (§4.3), and post-slowdown analysis (§4.4).

4.1 Slowdown Distributions and Characteristics

In this section, we present slowdown and tail distributions and their basic characteristics such as temporal behaviors and the extent of the problem.

(a)

Y:	**90^{th}**	**95**	**99**	**99.9**	**99.99**	**99.999**
Slowdown (S_i) at Y^{th} percentile						
Disk	1.1x	1.2	1.4	2.7	9	30
SSD	1.1x	1.2	1.7	3.1	10	39
Greatest slowdown (T^1) at Y^{th} percentile						
Disk	1.3x	1.5	2.4	9	29	229
SSD	1.3x	1.5	2.5	20	37	65

(b)

X:	**1.2x**	**1.5x**	**2x**	**4x**
Percentile at S_i=X				
Disk	97.0^{th}	99.3	99.78	99.96
SSD	95.9^{th}	98.7	99.42	99.92
Percentile at T^1=X				
Disk	83.3^{th}	95.4	98.50	99.72
SSD	87.0^{th}	95.2	97.77	99.67

Table 3: **Slowdown and percentile intersections.** *The table shows several detailed points in Figure 4. Table (a) details slowdown values at specific percentiles. Table (b) details percentile values at specific slowdown ratios.*

4.1.1 Slowdown (S_i) Distribution

We first take all S_i values and plot their distribution as shown by the thick (blue) line in Figure 4 (steeper lines imply more stability). Table 3 details some of the slowdown and percentile intersections.

Finding #1: *Storage performance instability is not uncommon.* Figure 4 and Table 3b show that there exists 0.22% and 0.58% of drive hours (99.8^{th} and 99.4^{th} percentiles) where some disks and SSDs exhibit at least 2x slowdown ($S_i \geq 2$). With a more conservative 1.5x slowdown threshold, the percentiles are 99.3^{th} and 98.7^{th} for disk and SSD respectively. These observations imply that user demands of stable latencies at 99.9^{th} percentile [15, 46, 54] (or 99^{th} with 1.5x threshold) are not met by current storage devices.

Disk and SSD slowdowns can be high in few cases. Table 3a shows that at four and five nines, slowdowns reach ≥9x and ≥30x respectively. In some of the worst cases, 3- and 4-digit disk slowdowns occurred in 2461 and 124 hours respectively, and 3-digit SSD slowdowns in 10 hours.

4.1.2 Tail (T^k) Distribution

We next plot the distributions of the three longest tails (T^{1-3}) in Figure 4. Table 3 details several T^1 values at specific percentiles.

Finding #2: *Storage tails appear at a significant rate.* The T^1 line in Figure 4 shows that there are 1.54% and 2.23% "tail hours" (*i.e.*, RAID hours with at least one slow drive). With a conservative 1.5x threshold, the percentiles are 95.4^{th} and 95.2^{th} for disk and SSD respectively. These numbers are alarming for full-stripe workload because the whole RAID will appear to be slow if one drive is slow. For such workload, stable latencies at

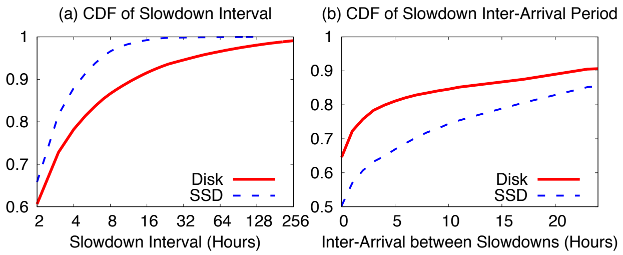


Figure 5: **Temporal behavior** (§4.1.3). *The figures show (a) the CDF of slowdown intervals (#hours until a slow drive becomes stable) and (b) the CDF of slowdown inter-arrival rates (#hours between two slowdown occurrences).*

99^{th} percentile (or 96^{th} with 1.5x threshold) cannot be guaranteed by current RAID deployments.

The differences between the three longest tails shed light on possible performance improvement from tail-tolerant RAID. If we reconstruct the late data from the slowest drive by reading from a parity drive, we can cut the longest tail. This is under an assumption that drive slowdowns are independent and thus reading from the parity drive can be faster. If two parity blocks are available (*e.g.*, in RAID-6), then tail-tolerant RAID can read two parity blocks to cut the last two tails.

Finding #3: *Tail-tolerant RAID has a significant potential to increase performance stability*. The T^1 and T^2 values at x=2 in Figure 4a suggests the opportunity to reduce disk tail hours from 1.5% to 0.6% if the longest tail can be cut, and furthermore to 0.3% (T^3) if the two longest tails can be cut. Similarly, Figure 4b shows that SSD tail hours can be reduced from 2.2% to 1.4%, and furthermore to 0.8% with tail-tolerant RAID.

The T^1 line in Figure 4b shows several vertical steps (*e.g.*, about 0.6% of T^1 values are exactly 2.0). To understand this, we analyze S_i values that are exactly 1.5x, 2.0x, and 3.0x. We find that they account for 0.4% of the entire SSD hours and their corresponding hourly and median latencies (L_i and L_{med}) are exactly multiples of 250 μs. We are currently investigating this further with the product groups to understand why some of the deployed SSDs behave that way.

4.1.3 Temporal Behavior

To study slowdown temporal behaviors, we first measure the *slowdown interval* (how many consecutive hours a slowdown persists). Figure 5a plots the distribution of slowdown intervals.

Finding #4: *Slowdown can persist over several hours.* Figure 5a shows that 40% of slow disks do not go back to stable within the next hour (and 35% for SSD). Furthermore, slowdown can also persist for a long time. For example, 13% and 3% of slow disks and SSDs stay slow for 8 hours or more respectively.

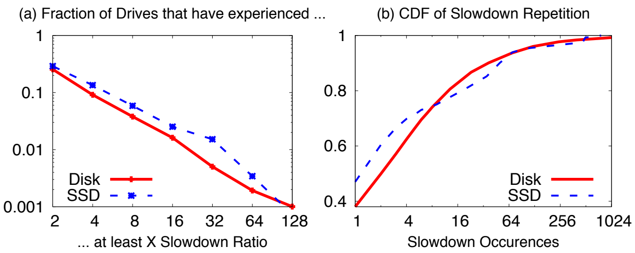


Figure 6: **Slowdown extent** (§4.1.4). *Figure (a) shows the fraction of all drives that have experienced at least one occurrence of X-time slowdown ratio as plotted on the x-axis; the y-axis is in $\log_{10}$ scale. Figure (b) shows the fraction of slow drives that has exhibited at least X slowdown occurrences.*

Next, we measure the *inter-arrival period* of slowdown occurrences from the perspective of each slow drive. Figure 5b shows the fraction of slowdown occurrences that arrive within X hours of the preceding slowdown; the arrival rates are binned by hour.

Finding #5: *Slowdown has a high temporal locality.* Figure 5b shows that 90% and 85% of disk and SSD slowdown occurrences from the same drive happen within the same day of the previous occurrence respectively. The two findings above suggest that history-based tail mitigation strategies can be a fitting solution; a slowdown occurrence should be leveraged as a good indicator for the possibility of near-future slowdowns.

4.1.4 Slowdown Extent

We now characterize the *slowdown extent* (*i.e.*, fraction of drives that have experienced slowdowns) in two ways. First, Figure 6a plots the fraction of all drives that have exhibited at least one occurrence of at least X-time slowdown ratio as plotted on the x-axis.

Finding #6: *A large extent of drive population has experienced slowdowns at different rates*. Figure 6a depicts that 26% and 29% of disk and SSD drives have exhibited $\geq$2x slowdowns at least one time in their lifetimes respectively. The fraction is also relatively significant for large slowdowns. For example, 1.6% and 2.5% of disk and SSD populations have experienced $\geq$16x slowdowns at least one time.

Next, we take only the population of slow drives (26% and 29% of the disk and SSD population) and plot the fraction of slow drives that has exhibited at least X slowdown occurrences, as shown in Figure 6b.

Finding #7: *Few slow drives experience a large number of slowdown repetitions.* Figure 6b shows that that around 6% and 5% of slow disks and SSDs exhibit at least 100 slowdown occurrences respectively. The majority of slow drives only incur few slowdown repetitions. For example, 62% and 70% of slow disks and SSDs exhibit only less than 5 slowdown occurrences re-

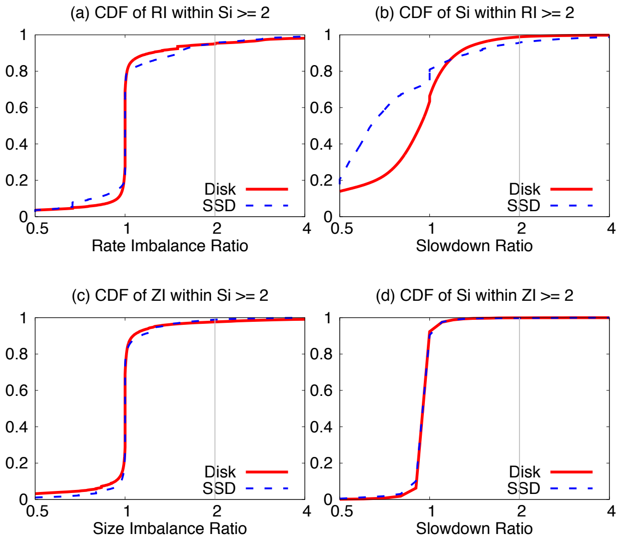


Figure 7: **CDF of size and rate imbalance (§4.2).** *Figure (a) plots the rate imbalance distribution (RI_i) within the population of slow drive hours ($S_i \geq 2$). A rate imbalance of X implies that the slow drive serves X times more I/Os, as plotted in the x-axis. Reversely, Figure (b) plots the slowdown distribution (S_i) within the population of rate-imbalanced drive hours ($RI_i \geq 2$). Figures (c) and (d) correlate slowdown and size imbalance in the same way as Figures (a) and (b).*

spectively. We emphasize that frequency of slowdown occurrences above are *only within the time duration of 87 days on average* (§3.2).

4.2 Workload Analysis

The previous section presents the basic characteristics of drive slowdowns. We now explore the possible root causes, starting with workload analysis. Drive slowdowns are often attributed to unbalanced workload (*e.g.*, a drive is busier than other drives). We had a hypothesis that such is not the case in our study due to the storage stack optimization (§3.2). To explore our hypothesis, we correlate slowdown with two workload-related metrics: size and rate imbalance.

4.2.1 Slowdown vs. Rate Imbalance

We first measure the hourly I/O count for every drive (R_i), the median (R_{med}), and the rate imbalance ($RI_i = R_i/R_{med}$); this method is similar to the way we measure S_i in Table 2. *If* workload is to blame for slowdowns, then we should observe a high correlation between slowdown (S_i) and rate imbalance (R_i). That is, slowdowns happen in conjunction with rate imbalance, for example, $S_i \geq 2$ happens during $R_i \geq 2$.

Figure 7a shows the rate imbalance distribution (RI_i) *only within* the population of slow drive hours. A rate imbalance of X (on the x-axis) implies that the slow drive serves X times more I/Os. The figure reveals that *only* 5% of slow drive hours happen when the drive receives 2x more I/Os than the peer drives. 95% of the slowdowns happen in the absence of rate imbalance (the rate-imbalance distribution is mostly aligned at x=1).

To strengthen our conjecture that rate imbalance is not a factor, we perform a reverse analysis. To recap, Figure 7a essentially shows how often slowdowns are caused by rate imbalance. We now ask the reverse: how often does rate imbalance cause slowdowns? The answer is shown in Figure 7b; it shows the slowdown distribution (S_i) *only within* the population of "overly" rate-imbalanced drive hours ($RI_i \geq 2$). Interestingly, rate imbalance has negligible effect on slowdowns; only 1% and 5% of rate-imbalanced disk and SSD hours experience slowdowns. From these two analyses, we conclude that rate imbalance is not a major root cause of slowdown.

4.2.2 Slowdown vs. Size Imbalance

Next, we correlate slowdown with size imbalance. Similar to the method above, we measure the hourly average I/O size for every drive (Z_i), the median (Z_{med}), and the size imbalance ($ZI_i = Z_i/Z_{med}$). Figure 7c plots the size imbalance distribution (ZI_i) only within the population of slow drive hours. A size imbalance of X implies that the slow drive serves X times larger I/O size. The size-imbalance distribution is very much aligned at x=1. Only 2.5% and 1.1% of slow disks and SSDs receive 2x larger I/O size than the peer drives in their group. Reversely, Figure 7d shows that only 0.1% and 0.2% of size-imbalanced disk and SSD hours experience more than 2x slowdowns.

Finding #8: *Slowdowns are independent of I/O rate and size imbalance.* As elaborated above, the large majority of slowdown occurrences (more than 95%) cannot be attributed to workload (I/O size or rate) imbalance.

4.3 Other Correlations

As workload imbalance is not a major root cause of slowdowns, we now attempt to find other possible root causes by correlating slowdowns with other metrics such as drive events, age, model and time of day.

4.3.1 Drive Event

Slowdown is often considered as a "silent" fault that needs to be monitored continuously. Thus, we ask: are there any *explicit* events surfacing near slowdown occurrences? To answer this, we collect *drive events* from our auto-support system.

Finding #9: *Slowdown is a "silent" performance fault.* A large majority of slowdowns are not accompanied with any explicit drive events. Out of the millions slow drive hours, we only observe hundreds of

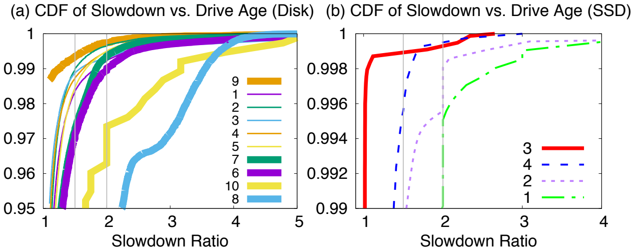


Figure 8: **Drive age (§4.3.2).** *The figures plot the slowdown distribution across different (a) disk and (b) SSD ages. Each line represents a specific age by year. Each figure legend is sorted from the left-most to right-most lines.*

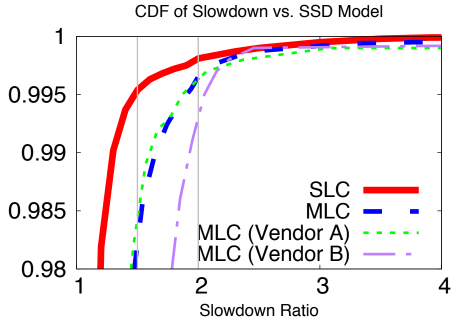


Figure 9: **SSD models (§4.3.3).** *The figure plots the slowdown distribution across different SSD models and vendors.*

drive events. However, when specific drive events happen (specifically, "disk is not spinning" and "disk is not responding"), 90% of the cases lead to slowdown occurrences. We rarely see storage timeouts (*e.g.*, SCSI timeout) because timeout values are typically set coarsely (*e.g.*, 60 seconds). Since typical latency ranges from tens of microseconds to few milliseconds, a slowdown must be five orders of magnitude to hit a timeout. Thus, to detect tail latencies, storage performance should be monitored continuously.

4.3.2 Drive Age

Next, we analyze if drive age matters to performance stability. We break the the slowdown distribution (S_i) by different ages (*i.e.*, how long the drive has been deployed) as shown in Figure 8.

For disks, the bold lines in Figure 8a clearly show that older disks experience more slowdowns. Interestingly, the population of older disks is small in our dataset and yet we can easily observe slowdown prevalence within this small population (the population of 6-10 year-old disks ranges from 0.02-3% while 1-5 year-old disks ranges from 8-33%). In the worst case, the 8th year, the 95^{th} percentile already reaches 2.3x slowdown. The 9th year (0.11% of the population) seems to be an outlier. Performance instability from disk aging due to mechanical wear-out is a possibility (§2).

For SSD, we do not observe a clear pattern. Although Figure 8b seemingly shows that young SSDs experience more slowdowns than older drives, it is hard to make such as a conclusion because of the small old-SSD population (3-4 year-old SSDs only make up 16% of the population while the 1-2 year-old is 83%).

Finding #10: Older disks tend to exhibit more slowdowns. For SSDs, no high degree of correlation can be made between slowdown and drive age.

4.3.3 Drive Model

We now correlate slowdown with drive model. Not all of our customers upload the model of the drives they use. Only 70% and 86% of customer disks and SSDs have model information. Thus, our analysis in this section is based on partial population.

We begin by correlating SSD model and slowdown. The SSD literature highlights the pressure to increase density, which leads to internal idiosyncrasies that can induce performance instability. Thus, it is interesting to know the impact of different flash cell levels to SSD performance stability.

Finding #11: *SLC slightly outperforms MLC drives in terms of performance stability*. As shown in Figure 9, at 1.5x slowdown threshold, MLC drives only reaches 98.2^{th} percentile while SLC reaches 99.5^{th} percentile. However, at 2x slowdown threshold, the distribution is only separated by 0.1%. As MLC exhibits less performance stability than SLC, future comparisons with TLC drives will be interesting.

Our dataset contains about 60:40 ratio of SLC vs. MLC drives. All the SLC drives come from one vendor, but the MLC drives come from two vendors with 90:10 population ratio. This allows us to compare vendors.

Finding #12: *SSD vendors seem to matter.* As shown by the two thin lines in Figure 9, one of the vendors (the 10% MLC population) has much less stability compared to the other one. This is interesting because the instability is clearly observable even within a small population. At 1.5x threshold, this vendor's MLC drives already reach 94.3^{th}percentile (out of the scope of Figure 9).

For disks, we use different model parameters such as storage capacity, RPM, and SAN interfaces (SATA, SAS, or Fibre Channel). However, we do not see any strong correlation.

4.3.4 Time of Day

We also perform an analysis based on time of day to identify if night-time background jobs such as disk scrubbing cause slowdowns. We find that slowdowns are uniformly distributed throughout the day and night.

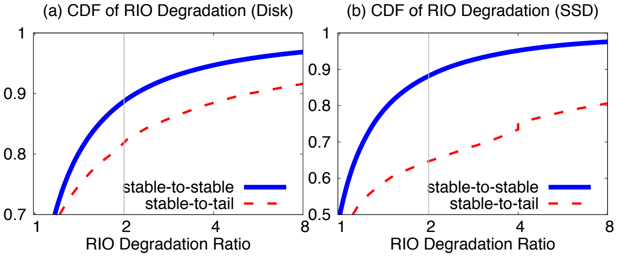


Figure 10: **RAID I/O degradation (§4.4.1).** *The figures contrast the distributions of RIO degradation between stable-to-stable and stable-to-tail transitions.*

4.4 Post-Slowdown Analysis

We now perform a post-mortem analysis: what happens *after* slowdown occurrences? We analyze this from two angles: RAID performance degradation and unplug/replug events.

4.4.1 RAID Performance Degradation

A slow drive has the potential to degrade the performance of the entire RAID, especially for full-stripe workload common in the studied RAID systems (§3.1), it is reasonable to make the following hypothesis: during the hour when a drive slows down, the RAID aggregate throughput will drop as the workload's intensity will be throttled by the slow drive. Currently, we do not have access to throughput metrics at the file system or application levels, and even if we do, connecting metrics from different levels will not be trivial. We leave cross-level analysis as future work, but meanwhile, given this constraint, we perform the following analysis to explore our hypothesis.

We derive a new metric, ***RIO*** (hourly RAID I/O count), which is the aggregate number of I/Os per hour from all the data drives in every RAID hour. Then, we derive ***RIO degradation*** (RAID throughput degradation) as the ratio RIO$_{lastHour}$ to RIO$_{currentHour}$. If the degradation is larger than one, it means the RAID group serves less I/Os than the previous hour.

Next, we distinguish *stable-to-stable* and *stable-to-tail* transitions. Stable RAID hour means all the drives are stable ($Si < 2$). Tail RAID hour implies at least one of the drives is slow. In stable-to-stable transitions, RIO degradation can naturally happen as workload "cools down". Thus, we first plot the distribution of stable-to-stable RIO degradation, shown by the solid blue line in Figure 10. We then select only the stable-to-tail transitions and plot the RIO degradation distribution, shown by the dashed red line in Figure 10.

Finding #13: *A slow drive can significantly degrade the performance of the entire RAID.* Figure 10 depicts a big gap of RAID I/O degradation between stable-to-stable and stable-to-tail transitions. In SSD-based RAID, the degradation impact is quite severe. Figure 10b for example shows that only 12% of stable-to-stable transitions observe ≥2x RIO degradation (likely from workload cooling down). However, in stable-to-slow transitions, there is *23% more chance* (the vertical gap at x=2) that RIO degrades by more than 2x. In disk-based RAID, RIO degradation is also felt with 7% more chance. This finding shows the real possibilities of workload throughput being degraded and stable drives being under-utilized during tail hours, which again motivates the need for tail-tolerant RAID.

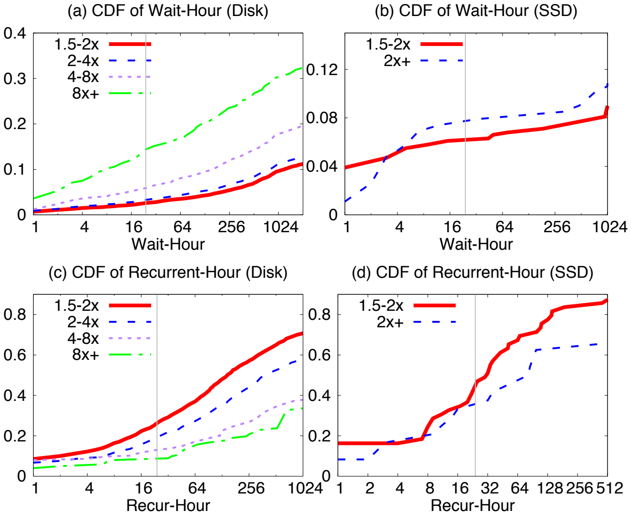


Figure 11: **Unplug/replug events (§4.4.2-§4.4.3).** *The figures show the relationships between slowdown occurrences and unplug/replug events. The top and bottom figures show the distribution of "wait-hour" and "recur-hour" respectively.*

We note that RAID degradation is felt more if user requests are casually dependent; RIO degradation only affects I/Os that are waiting for the completion of previous I/Os. Furthermore, since our dataset is based on hourly average latencies, there is no sufficient information that shows every I/O is delayed at the same slowdown rate. We believe these are the reasons why we do not see a complete collapse of RIO degradation.

4.4.2 Unplug Events

When a drive slows down, the administrator might unplug the drive (*e.g.*, for offline diagnosis) and later replug the drive. Unplug/replug is a manual administrator's process, but such events are logged in our auto-support system. To analyze unplug patterns, we define *wait-hour* as the number of hours between a slowdown occurrence and a subsequent unplug event; if a slowdown persists in consecutive hours, we only take the first slow hour. Figures 11a-b show the wait-hour distribution within the population of slow disks and SSDs respectively.

Finding #14: *Unplug events are common.* Figures 11a-b show that within a day, around 4% and 8% of slow

(≥2x) disks and SSDs are unplugged respectively. For "mild" slowdowns (1.5-2x), the numbers are 3% and 6%. Figure 11a also shows a pattern where disks with more severe slowdowns are unplugged at higher rates; this pattern does not show up in SSD.

4.4.3 Replug Events

We first would like to note that unplug is not the same as drive replacement; a replacement implies an unplug without replug. With this, we raise two questions: What is the replug rate? Do replugged drives exhibit further slowdowns? To analyze the latter, we define *recur-hour* as the number of hours between a replug event and the next slowdown occurrence. Figures 11c-d show the recur-hour distribution within the population of slow disks and SSDs respectively.

Finding #15: *Replug rate is high and slowdowns still recur after replug events.* In our dataset, customers replug 89% and 100% of disks and SSDs that they unplugged respectively (not shown in figures). Figures 11c-d answer the second question, showing that 18% and 35% of replugged disks and SSDs exhibit another slowdown within a day. This finding points out that administrators are reluctant to completely replace slow drives, likely because slowdowns are transient (not all slowdowns appear in consecutive hours) and thus cannot be reproduced in offline diagnosis and furthermore the cost of drive replacement can be unnecessarily expensive. Yet, as slowdown can recur, there is a need for online tail mitigation approaches.

In terms of unplug-replug duration, 54% of unplugged disks are replugged within 2 hours and 90% within 10 hours. For SSD, 61% are replugged within 2 hours and 97% within 10 hours.

4.5 Summary

It is now evident that storage performance instability at the drive level is not uncommon. One of our major findings is the little correlation between performance instability and workload imbalance. One major analysis challenge is the "silent" nature of slowdowns; they are not accompanied by explicit drive events, and therefore, pinpointing the root cause of each slowdown occurrence is still an open problem. However, in terms of the overall findings, our conversations with product teams and vendors [4] confirm that many instances of drive performance faults are caused by drive anomalies; there are strong connections between our findings and some of the anecdotal evidence we gathered (§2). As RAID deployments can suffer from storage tails, we next discuss the concept of tail-tolerant RAID.

5 Tail-Tolerant RAID

With drive performance instability, RAID performance is in jeopardy. When a request is striped across many drives, the request cannot finish until all the individual I/Os complete (Figure 1); the request latency will follow the tail latency. As request throughput degrades, stable drives become under-utilized. Tail-tolerant RAID is one solution to the problem and it brings two advantages.

First, slow drives are masked. This is a simple goal but crucial for several reasons: stringent SLOs require stability at high percentiles (*e.g.*, 99% or even 99.9% [15, 45, 48, 52]); slow drives, if not masked, can create cascades of performance failures to applications [16]; and drive slowdowns can falsely signal applications to back off, especially in systems that treat slowdowns as hints of overload [24].

Second, tail-tolerant RAID is a cheaper solution than drive replacements, especially in the context of transient slowdowns (§4.1.3) and high replug rates by administrators (§4.4.3). Unnecessary replacements might be undesirable due to the hardware cost and the expensive RAID re-building process as as drive capacity increases.

Given these advantages, we performed an opportunity assessment of tail-tolerant strategies at the RAID level. We emphasize that the main focus of this paper is the large-scale analysis of storage tails; the initial exploration of tail-tolerant RAID in this section is only to assess the benefits of such an approach.

5.1 Tail-Tolerant Strategies

We explore three tail-tolerant strategies: *reactive*, *proactive*, and *adaptive*. They are analogous to popular approaches in parallel distributed computing such as speculative execution [14] and hedging/cloning [6, 13]. To mimic our RAID systems (§3.2), we currently focus on tail tolerance for RAID-6 with *non-rotating* parity (Figure 1 and §3.1). We name our prototype ToleRAID, for simplicity of reference.

Currently, we only focus on full-stripe read workload where ToleRAID can cut "read tails" in the following ways. In normal reads, the two parity drives are unused (if no errors), and thus can be leveraged to mask up to two slow data drives. For example, if one data drive is slow, ToleRAID can issue an extra read to one parity drive and rebuild the "late" data.

Reactive: A simple strategy is reactive. If a drive (or two) has not returned the data for STx (slowdown threshold) longer than the median latency, reactive will perform an extra read (or two) to the parity drive(s). Reactive strategy should be enabled by default in order to cut extremely long tails. It is also good for mostly stable environment where slowdowns are rare. A small ST will create more extra reads and a large ST will respond late to tails. We set $ST = 2$ in our evaluation, which means we still need to wait for roughly an additional 1x median latency to complete the I/O (a total slowdown of 3x in our case). While reactive strategies work well in cluster computing (*e.g.*, speculative execution for medium-long jobs), they can react too late for small I/O latencies (*e.g.*, hundreds of microseconds). Therefore, we explore proactive and adaptive approaches.

Proactive: This approach performs extra reads to the parity drives concurrently with the original I/Os. The number of extra reads can be one (P drive) or two (P and Q); we name them PROACTIVE$_1$ and PROACTIVE$_2$ respectively. Proactive works well to cut short tails (near the slowdown threshold); as discussed above, reactive depends on ST and can be a little bit too late. The downside of proactive strategy is the extra read traffic.

Adaptive: This approach is a middle point between the two strategies above. Adaptive by default runs the reactive approach. When the reactive policy is triggered repeatedly for SR times (slowdown repeats) on the same drive, then ToleRAID becomes proactive until the slowdown of the offending drive is less than ST. If two drives are persistently slow, then ToleRAID runs PROACTIVE$_2$. Adaptive is appropriate for instability that comes from persistent and periodic interferences such as background SSD GC, SMR log cleaning, or I/O contention from multi-tenancy.

5.2 Evaluation

Our user-level ToleRAID prototype stripes each RAID request into 4-KB chunks (§3.2), merge consecutive per-drive chunks, and submit them as direct I/Os. We insert a delay-injection layer that emulates I/O slowdowns. Our prototype takes two inputs: block-level trace and slowdown distribution. Below, we show ToleRAID results from running a block trace from Hadoop Wordcount benchmark, which contains mostly big reads. We perform the experiments on 8-drive RAID running IBM 500GB SATA-600 7.2K disk drives.

We use two slowdown distributions: (1) *Rare* distribution, which is uniformly sampled from our disk dataset (Figure 4a). Here, tails (T^1) are rare (1.5%) but long tails exist (Table 3). (2) *Periodic* distribution, based on our study of Amazon EC2 ephemeral SSDs (not shown due to space constraints). In this study, we rent SSD nodes and found a case where one of the locally-attached SSDs periodically exhibited 5x read slowdowns that lasted for 3-6 minutes and repeated every 2-3 hours (2.3% instability period on average).

Figure 12 shows the pros and cons of the four policies using the two different distributions. In all cases, PROACTIVE$_1$ and PROACTIVE$_2$ always incur roughly 16.7% and 33.3% extra reads. In Figure 12a, REACTIVE can cut long tails and ensure RAID only slows down by at most 3x, while only introducing 0.5% I/O overhead. PROACTIVE$_2$, compared to PROACTIVE$_1$, gives slight benefits (*e.g.*, 1.8x vs. 2.3x at 99th percentile). Note also that PROACTIVE$_1$ does not use REACTIVE, and thus PROACTIVE$_1$ loses to REACTIVE within the 0.1% chance where two disks are slow at the same time. ADAPTIVE does not show more benefits in non-persistent scenarios. Figure 12b shows that in periodic distribution with persistent slowdowns, ADAPTIVE works best; it cuts long tails but only incurs 2.3% I/O overhead.

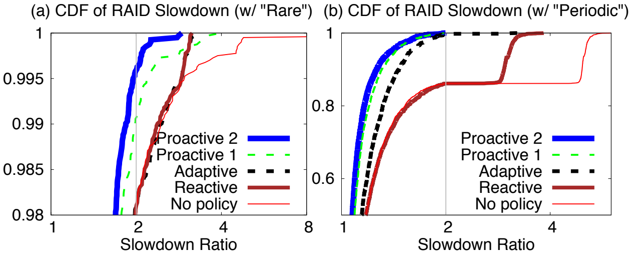


Figure 12: **ToleRAID evaluation.** *The figures show the pros and cons of various ToleRAID strategies based on two slowdown distributions: (a) Rare and (b) Periodic. The figures plot the T^1 distribution (i.e., the RAID slowdown). T^1 is essentially based on the longest tail latency among the necessary blocks that each policy needs to wait for.*

Overall, ToleRAID shows potential benefits. In separate experiments, we have also measured Linux Software RAID degradation in the presence of storage tails (with `dmsetup` delay utilities). We are integrating ToleRAID to Linux Software RAID and extending it to cover more policies and scenarios (partial reads, writes, etc.).

6 Discussions

We hope our work will spur a set of interesting future research directions for the larger storage community to address. We discuss this in the context of performance-log analysis and tail mitigations.

Enhanced data collection: The first limitation of our dataset is the hourly aggregation, preventing us from performing micro analysis. Monitoring and capturing fine-grained data points will incur high computation and storage overhead. However, during problematic periods, future monitoring systems should capture detailed data. Our ToleRAID evaluation hints that realistic slowdown distributions are a crucial element in benchmarking tail-tolerant policies. More distribution benchmarks are needed to shape the tail-tolerant RAID research area. The second limitation is the absence of other metrics that can be linked to slowdowns (*e.g.*, heat and vibration levels). Similarly, future monitoring systems can include such metrics.

Our current SSD dataset is two orders of magnitude smaller than the disk dataset. As SSD becomes the front-end storage in datacenters, larger and longer studies of SSD performance instability is needed. Similarly, denser SMR disk drives will replace old generation disks [5, 18]. Performance studies of SSD-based and SMR-based RAID will be valuable, especially for understanding the ramifications of internal SSD garbage-collection and SMR cleaning to the overall RAID performance.

Further analyses: Correlating slowdowns to latent sector errors, corruptions, drive failures (*e.g.*, from SMART logs), and application performance would be interesting future work. One challenge we had was that not all vendors consistently use SMART and report drive errors. In this paper, we use median values to measure tail latencies and slowdowns similar to other work [13, 34, 52]. We do so because using median values will not hide the severity of long tails. Using median is exaggerating if (N-1)/2 of the drives have significantly higher latencies than the rest; however, we did not observe such cases. Finally, we mainly use 2x slowdown threshold, and occasionally show results from a more conservative 1.5x threshold. Further analyses based on average latency values and different threshold levels are possible.

Tail mitigations: We believe the design space of tail-tolerant RAID is vast considering different forms of RAID (RAID-5/6/10, etc.), different types of erasure coding [38], various slowdown distributions in the field, and diverse user SLA expectations. In our initial assessment, ToleRAID uses a black-box approach, but there are other opportunities to cut tails "at the source" with transparent interactions between devices and the RAID layer. In special cases such as materials trapped between disk head and platter (which will be more prevalent in "slim" drives with low heights), the file system or RAID layer can inject random I/Os to "sweep" the dust off. In summary, each root cause can be mitigated with specific strategies. The process of identifying all possible root causes of performance instability should be continued for future mitigation designs.

7 Related Work

Large-scale storage studies at the same scale as ours were conducted for analysis of whole-disk failures [37, 40], latent sector errors [8, 36], and sector corruptions [9]. Many of these studies were started based on anecdotal evidence of storage faults. Today, as these studies had provided real empirical evidence, it is a common expectation that storage devices exhibit such faults. Likewise, our study will provide the same significance of contribution, but in the context of performance faults.

Krevat *et al.* [33] demonstrate that disks are like "snowflakes" (same model can have 5-14% throughput variance); they only analyze throughput metrics on 70 drives with simple microbenchmarks. To the best of our knowledge, our work is the first to conduct a large-scale performance instability analysis at the drive level.

Storage performance variability is typically addressed in the context of storage QoS (*e.g.*, mClock [25], PARDA [24], Pisces [42]) and more recently in cloud storage services (*e.g.*, C3 [45], CosTLO [52]). Other recent work reduces performance variability at the file system (*e.g.*, Chopper [30]), I/O scheduler (*e.g.*, split-level scheduling [55]), and SSD layers (*e.g.*, Purity [12], Flash on Rails [43]). Different than ours, these sets of work do not specifically target drive-level tail latencies.

Finally, as mentioned before, reactive, proactive and adaptive tail-tolerant strategies are lessons learned from the distributed cluster computing (*e.g.*, MapReduce [14], dolly [6], Mantri [7], KMN [50]) and distributed storage systems (*e.g.*, Windows Azure Storage [31], RobuSTore [53]). The applications of these high-level strategies in the context of RAID will significantly differ.

8 Conclusion

We have "transformed" anecdotes of storage performance instability into large-scale empirical evidence. Our analysis so far is solely based on last generation drives (few years in deployment). With trends in disk and SSD technology (*e.g.*, SMR disks, TLC flash devices), the worst might be yet to come; performance instability can be more prevalent in the future, and our findings are perhaps just the beginning. File and RAID systems are now faced with more responsibilities. Not only must they handle known storage faults such as latent sector errors and corruptions [9, 27, 28, 39], but also now they must mask drive tail latencies as well. Lessons can be learned from the distributed computing community where a large body of work has been born since the issue of tail latencies became a spotlight a decade ago [14]. Similarly, we hope "the tail at store" will spur exciting new research directions within the storage community.

9 Acknowledgments

We thank Bianca Schroeder, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. We also would like to thank Lakshmi N. Bairavasundaram and Shiqin Yan for their initial help and Siva Jayasenan and Art Harkin for their managerial support. This material is based upon work supported by NetApp, Inc. and the NSF (grant Nos. CCF-1336580, CNS-1350499 and CNS-1526304).

References

[1] Weak Head. http://forum.hddguru.com/search.php?keywords=weak-head.

[2] Personal Communication from Andrew Baptist of Cleversafe, 2013.

[3] Personal Communication from Garth Gibson of CMU, 2013.

[4] Personal Communication from NetApp Hardware and Product Teams, 2015.

[5] Abutalib Aghayev and Peter Desnoyers. Skylight-A Window on Shingled Disk Operation. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[7] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[8] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007.

[9] 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.

[10] Jeff Barr. Amazon S3 - 905 Billion Objects and 650,000 Requests/Second. http://aws.amazon.com/cn/blogs/aws/amazon-s3-905-billion-objects-and-650000-requestssecond, 2012.

[11] Jake Brutlag. Speed Matters for Google Web Search. http://services.google.com/fh/files/blogs/google_delayexp.pdf, 2009.

[12] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[13] Jeffrey Dean and Luiz Andr Barroso. The Tail at Scale. *Communications of the ACM*, 56(2), February 2013.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[16] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[17] Michael Feldman. Startup Takes Aim at Performance-Killing Vibration in Datacenter. http://www.hpcwire.com/2010/01/19/startup_takes_aim_at_performance-killing_vibration_in_datacenter/, 2010.

[18] Tim Feldman and Garth Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *USENIX ;login: Magazine*, 38(3), Jun 2013.

[19] Robert Frickey. Data Integrity on 20nm SSDs. In *Flash Memory Summit*, 2012.

[20] Natalie Gagliordi. Seagate Q2 solid, 61.3 exabytes shipped. http://www.zdnet.com/article/seagate-q2-solid-61-3-exabytes-shipped/, 2015.

[21] John Gantz and David Reinsel. The digital universe in 2020. http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf, 2012.

[22] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*, 2009.

[23] Laura M. Grupp, John D. Davis, and Steven Swanson. The Bleak Future of NAND Flash Memory. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[24] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST)*, 2009.

[25] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[26] 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.

[27] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[28] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[29] Robin Harris. Bad, bad, bad vibrations. http://www.zdnet.com/article/bad-bad-bad-vibrations/, 2010.

[30] Jun He, Duy Nguyen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Reducing File System Tail

Latencies with Chopper. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[31] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, 2012.

[32] Jennifer Johnson. SSD Market On Track For More Than Double Growth This Year. http://hothardware.com/news/SSD-Market-On-Track-For-More-Than-Double-Growth-This-Year, 2013.

[33] Elie Krevat, Joseph Tucek, and Gregory R. Ganger. Disks Are Like Snowflakes: No Two Are Alike. In *The 13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, 2011.

[34] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[35] Jim Liddle. Amazon found every 100ms of latency cost them 1% in sales. http://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/, 2008.

[36] Ao Ma, Fred Douglis, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. RAIDShield: Characterizing, Monitoring, and Proactively Protecting Against Disk Failures. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[37] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.

[38] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST)*, 2009.

[39] 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.

[40] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.

[41] Anand Lal Shimpi. Intel Discovers Bug in 6-Series Chipset: Our Analysis. http://www.anandtech.com/show/4142/intel-discovers-bug-in-6series-chipset-begins-recall, 2011.

[42] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[43] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on rails: Consistent flash performance through redundancy. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[44] Steve Souders. Velocity and the Bottom Line. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html, 2009.

[45] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[46] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[47] Utah Emulab Testbed. RAID controller timeout problems on boss node, on 6/21/2013. http://www.emulab.net/news.php3, 2013.

[48] Beth Trushkowsky, Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST)*, 2011.

[49] Kristian Vatto. Samsung Acknowledges the SSD 840 EVO Read Performance Bug - Fix Is on the Way. http://www.anandtech.com/show/8550/samsung-acknowledges-the-ssd-840-evo-read-performance-bug-fix-is-on-the-way, 2014.

[50] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. The Power of Choice in Data-Aware Cluster Scheduling. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[51] Guanying Wu and Xubin He. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[52] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[53] Huaxia Xia and Andrew A. Chien. RobuSTore: Robust Performance for Distributed Storage Systems. In *Proceedings of the 2007 Conference on High Performance Networking and Computing (SC)*, 2007.

[54] Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. Wrangler: Predictable and Faster Jobs using Fewer Resources. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[55] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-Level I/O Scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[56] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

Estimating Unseen Deduplication – from Theory to Practice

Danny Harnik, Ety Khaitzin and Dmitry Sotnikov
IBM Research–Haifa
{dannyh, etyk, dmitrys}@il.ibm.com

Abstract

Estimating the deduplication ratio of a very large dataset is both extremely useful, but genuinely very hard to perform. In this work we present a new method for accurately estimating deduplication benefits that runs $3X$ to $15X$ faster than the state of the art to date. The level of improvement depends on the data itself and on the storage media that it resides on. The technique is based on breakthrough theoretical work by Valiant and Valiant from 2011, that give a provably accurate method for estimating various measures while seeing only a fraction of the data. However, for the use case of deduplication estimation, putting this theory into practice runs into significant obstacles. In this work, we find solutions and novel techniques to enable the use of this new and exciting approach. Our contributions include a novel approach for gauging the estimation accuracy, techniques to run it with low memory consumption, a method to evaluate the combined compression and deduplication ratio, and ways to perform the actual sampling in real storage systems in order to actually reap benefits from these algorithms. We evaluated our work on a number of real world datasets.

1 Introduction

1.1 Deduplication and Estimation

After years of flourishing in the world of backups, deduplication has taken center stage and is now positioned as a key technology for primary storage. With the rise of all-flash storage systems that have both higher cost and much better random read performance than rotating disks, deduplication and data reduction in general, makes more sense than ever. Combined with the popularity of modern virtual environments and their high repetitiveness, consolidating duplicate data reaps very large benefits for such high-end storage systems. This trend is bound to continue with new storage class memories looming, that are expected to have even better random access and higher cost per GB than flash.

This paper is about an important yet extremely hard question – How to estimate the deduplication benefits of a given dataset? Potential customers need this information in order to make informed decisions on whether high-end storage with deduplication is worthwhile for them. Even more so, the question of sizing and capacity planning is deeply tied to the deduplication effectiveness expected on the specific data. Indeed, all vendors of deduplication solutions have faced this question and unfortunately there are no easy solutions.

1.2 The hardness of Deduplication Estimation and State of the Art

The difficulty stems from the fact that deduplication is a *global* property and as such requires searching across large amounts of data. In fact, there are theoretical proofs that this problem is hard [12] and more precisely, that in order to get an accurate estimation one is required to read a large fraction of the data from disk. In contrast, *compression* is a local procedure and therefore the compression estimation problem can be solved very efficiently [11].

As a result, the existing solutions in the market take one of two approaches: The first is simply to give an educated guess based on prior knowledge and based on information about the workload at hand. For example: Virtual Desktop Infrastructure (VDI) environments were reported (e.g. [1]) to give an average 0.16 deduplication ratio (a 1:6 reduction). However in reality, depending on the specific environment, the results can vary all the way between a 0.5 to a 0.02 deduplication ratio (between 1:2 and 1:50). As such, using such vague estimation for sizing is highly inaccurate.

The other approach is a full scan of the data at hand. In practice, a typical user runs a full scan on as much data as possible and gets an accurate estimation, but *only for the data that was scanned*. This method is not without challenges, since evaluating the deduplication ratio of a scanned dataset requires a large amount of memory and disk operations, typically much higher than would be allowed for an estimation scan. As a result, research on the topic [12, 19] has focused on getting accurate estimations with low memory requirement, while still reading all data from disk (and computing hashes on all of the data).

In this work we study the ability to estimate deduplication while *not* reading the entire dataset.

1.3 Distinct Elements Counting

The problem of estimating deduplication has surfaced in the past few years with the popularity of the technology. However, this problem is directly linked to a long standing problem in computer science, that of *estimating the number of distinct elements* in a large set or population. With motivations ranging from Biology (estimating the number of species) to Data Bases (distinct values in a table/column), the problem received much attention. There is a long list of heuristic statistical estimators (e.g. [4, 9, 10]), but these do not have tight guarantees on their accuracy and mostly target sets that have a relatively low number of distinct elements. Their empirical tests perform very poorly on distributions with a long tail (distributions in which a large fraction of the data has low duplication counts) which is the common case with deduplication. Figure 1 shows examples of how inaccurate heuristic estimations can be on a real dataset. It also shows how far the deduplication ratio of the sample can be from that of the entire dataset.

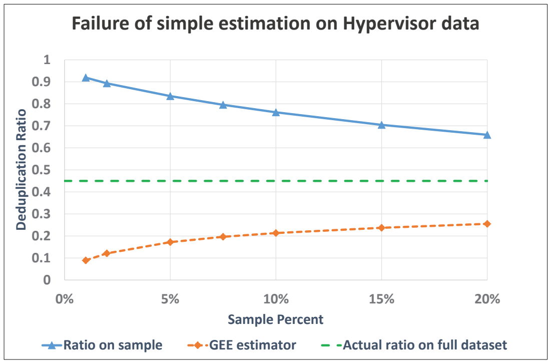


Figure 1: An example of the failure of current sampling approaches: Each graph depicts a deduplication ratio estimate as a function of the sampling percent. The points are an average over 100 random samples of the same percent. We see that simply looking at the ratio on the sample gives a very pessimistic estimation while the estimator from [4] is always too optimistic in this example.

This empirical difficulty is also supported by theoretical lower bounds [16] proving that an accurate estimation would require scanning a large fraction of the data. As a result, a large bulk of the work on distinct elements focused on low-memory estimation on data streams (including a long list of studies starting from [7] and culminating in [14]). These estimation methods require a full scan of the data and form the foundation for the low-memory scans for deduplication estimation mentioned in the previous section.

In 2011, in a breakthrough paper, Valiant and Valiant [17] showed that at least $\Omega(\frac{n}{\log n})$ of the data must be inspected and more over, that there is a matching upper bound. Namely, they showed a theoretical algorithm that achieves provable accuracy if at least $O(\frac{n}{\log n})$ of the elements are examined. Subsequently, a variation on this algorithm was also implemented by the same authors [18]. Note that the Valiants work, titled “Estimating the unseen” is more general than just distinct elements estimation and can be used to estimate other measures such as the Entropy of the data (which was the focus in the second paper [18]).

This new ”Unseen” algorithm is the starting point of our work in which we attempt to deploy it for deduplication estimation.

1.4 Our Work

More often than not, moving between theory and practice is not straightforward and this was definitely the case for estimating deduplication. In this work we tackle many challenges that arise when trying to successfully employ this new technique in a practical setting. For starters, it is not clear that performing random sampling at a small granularity has much benefit over a full sequential scan in a HDD based system. But there are a number of deeper issues that need to be tackled in order to actually benefit from the new approach. Following is an overview of the main topics and our solutions:

Understanding the estimation accuracy. The proofs of accuracy of the Unseen algorithm are theoretic and asymptotic in nature and simply do not translate to concrete real world numbers. Moreover, they provide a worst case analysis and do not give any guarantee for datasets that are easier to analyze. So there is no real way to know how much data to sample and what fraction is actually sufficient. In this work we present a novel method to gauge the accuracy of the algorithm. Rather than return an estimation, our technique outputs a range in which the actual result is expected to lie. This is practical in many ways, and specifically allows for a gradual execution: first take a small sample and evaluate its results and if the range is too large, then continue by increasing the sample size. While our tests indicate that a 15% sample is sufficient for a good estimation on all workloads, some real life workloads reach a good enough estimation with a sample as small as 3% or even less. Using our method, one can stop early when reaching a sufficiently tight estimation.

The memory consumption of the algorithm. In real systems, being able to perform the estimation with a small memory footprint and without additional disk IOs is highly desirable, and in some cases a must. The problem is that simply running the Unseen algorithm as prescribed requires mapping and counting all of the distinct chunks in the sample. In the use case of deduplication this number can be extremely large, on the same order of magnitude as the number of chunks in the entire dataset. Note that low memory usage also benefits in lower communication bandwidth when the estimation is performed in a distributed system.

Existing solutions for low-memory estimation of distinct elements cannot be combined in a straightforward manner with the Unseen algorithm. Rather, they require some modifications and a careful combination. We present two such approaches: one with tighter accuracy, and a second that loses more estimation tightness but is better for distributed and adaptive settings. In both cases the overall algorithm can run with as little as 10MBs of memory.

Combining deduplication and compression. Many of the systems deploying deduplication also use compression to supplement the data reduction. It was shown in multiple works that this combination is very useful in order to achieve improved data reduction for all workloads [5, 13, 15]. A natural approach to estimating the combined benefits is to estimate each one separately and multiply the ratios for both. However, this will only produce correct results when the deduplication effectiveness and compression effectiveness are independent, which in some cases is not true. We present a method to integrate compression into the Unseen algorithm and show that it yields accurate results.

How to perform the sampling? As stated above, performing straightforward sampling at a small granularity (e.g. 4KB) is extremely costly in HDD based systems (in some scenarios, sampling as little as 2% may already take more than a full scan). Instead we resort to sampling at large "super-chunks" (of 1MB) and performing reads in a sorted fashion. Such sampling runs significantly faster than a full scan and this is the main source of our time gains.

Equally as important, we show that our methods can be tuned to give correct and reliable estimations under this restriction (at the cost of a slightly looser estimation range). We also suggest an overall sampling strategy that requires low memory, produces sorted non-repeating reads and can be run in gradual fashion (e.g. if we want to first read a 5% sample and then enlarge the sample to a 10% one).

Summary of our results. In summary, we design and evaluate a new method for estimating deduplication ratios in large datasets. Our strategy utilizes less than 10MBs of RAM space, can be distributed, and can accurately estimate the joint benefits of compression and deduplication (as well as their separate benefits). The resulting estimation is presented as a range in which the actual ratio is expected to reside (rather than a single number). This allows for a gradual mode of estimation, where one can sample a small fraction, evaluate it and continue sampling if the resulting range is too loose. Note that the execution time of the actual estimation algorithms is negligible vs. the time that it takes to scan the data, so being able to stop with a small sampling fraction is paramount to achieving an overall time improvement.

We evaluate the method on a number of real life workloads and validate its high accuracy. Overall our method achieves at least a 3X time improvement over the state of the art scans. The time improvement varies according to the data and the medium on which data is stored, and can reach time improvements of 15X and more.

2 Background and the Core algorithm

2.1 Preliminaries and Background

Deduplication is performed on data chunks of size that depends on the system at hand. In this paper we consider fixed size chunks of 4KB, a popular choice since it matches the underlying page size in many environments. However the results can be easily generalized to different chunking sizes and methods. Note that variable-sized chunking can be handled in our framework but adds complexity especially with respect to the actual sampling of chunks.

As is customary in deduplication, we represent the data chunks by a hash value of the data (we use the SHA1 hash function). Deduplication occurs when two chunks have the same hash value.

Denote the dataset at hand by $\mathcal{S}$ and view it as consisting of N data chunks (namely N hash values). The dataset is made up from D distinct chunks, where the i^{th} element appears n_i times in the dataset. This means that $\sum_{i=1}^{D} n_i = N$.

Our ultimate target is to come up with an estimation of the value D, or equivalently of the ratio $r = \frac{D}{N}$. Note that throughout the paper we use the convention where data reduction (deduplication or compression) is a ratio between in $[0, 1]$ where lower is better. Namely, ratio 1.0 means no reduction at all and 0.03 means that the data is reduced to 3% of its original size (97% saving).

When discussing the sampling, we will consider a sample of size K out of the entire dataset of N chunks. The corresponding sampling rate is denoted by $p = \frac{K}{N}$ (for brevity, we usually present p in percentage rather than a fraction). We denote by $\mathcal{S}_p$ the random sample of fraction p from $\mathcal{S}$.

A key concept for this work that is what we term a Duplication Frequency Histogram (DFH) that is defined next (note that in [17] this was termed the "fingerprint" of the dataset).

Definition 1. *A* **Duplication Frequency Histogram (DFH)** *of a dataset $\mathcal{S}$ is a histogram $x = \{x_1, x_2, ...\}$ in which the value x_i is the number of distinct chunks that appeared exactly i times in $\mathcal{S}$.*

For example, the DFH of a dataset consisting of N distinct elements will have $x_1 = N$ and zero elsewhere. An equal sized dataset where all elements appear exactly twice will have a DFH with $x_2 = \frac{N}{2}$ and zero elsewhere. Note that for a legal DFH it must hold that $\sum_i x_i \cdot i = N$ and moreover that $\sum_i x_i = D$. The length of a DFH is set by the highest non-zero x_i. In other words it is the frequency of the most popular chunk in the dataset. The same definition of DFH holds also when discussing a sample rather than entire dataset.

The approach of the Unseen algorithm is to estimate the DFH of a dataset and from it devise an estimation of the deduplication.

2.2 The Unseen Algorithm

In this section we give a high level presentation of the core Unseen algorithm. The input of the algorithm is a DFH y of the observed sample $\mathcal{S}_p$ and from it the algorithm finds an estimation $\widehat{x}$ of the DFH of the entire dataset $\mathcal{S}$. At a high level, the algorithm finds a DFH $\widehat{x}$ on the full set that serves as the "best explanation" to the observed DFH y on the sample.

As a preliminary step, for a DFH x' on the dataset $\mathcal{S}$ define the **expected DFH** y' on a random p sample of $\mathcal{S}$. In the expected DFH each entry is exactly the statistical expectancy of this value in a random p sample. Namely y'_i is the expected number of chunks that appear exactly i times in a random p fraction sample. For fixed p this expected DFH can be computed given x' via a linear transformation and can be presented by a matrix A_p such that $y' = A_p \cdot x'$.

The main idea in the Unseen algorithm is to find an x' that minimizes a distance measure between the expected DFH y' and the observed DFH y. The distance measure used is a normalized L1 Norm (normalized by the values of the observed y). We use the following notation for the exact measure being used:

$$\Delta_p(x', y) = \sum_i \frac{1}{\sqrt{y_i + 1}} \left| y_i - (A_p \cdot x')_i \right| .$$

The algorithm uses Linear Programming for the minimization and is outlined in Algorithm 1.

```
Algorithm 1: Unseen Core
  Input: Sample DFH y, fraction p, total size N
  Output: Estimated deduplication ratio $\widehat{r}$
  /* Prepare expected DFH transformation*/
      $A_p \leftarrow$ prepareExpectedA(p);

  Linear program:
      Find $x'$ that minimizes: $\Delta_p(x', y)$
      Under constraints: /* $x'$ is a legal DFH */
          $\sum_i x'_i \cdot i = N$ and $\forall i \; x'_i \geq 0$

      return $\widehat{r} = \frac{\sum_i x'_i}{N}$
```

The actual algorithm is a bit more complicated due to two main issues: 1) this methodology is suited for estimating the duplication frequencies of unpopular chunks. The very popular chunks can be estimated in a straightforward manner (an element with a high count c is expected to appear approximately $p \cdot c$ times in the sample). So the DFH is first broken into the easy part for straightforward estimation and the hard part for estimation via Algorithm 1. 2) Solving a Linear program with too many variables is impractical, so instead of solving for a full DFH x, a sparser mesh of values is used (meaning that not all duplication values are allowed in x). This relaxation is acceptable since this level of inaccuracy has very little influence for high frequency counts. It is also crucial to make the running time of the LP low and basically negligible with respect to the scan time.

The matrix A_p is computed by a combination of binomial probabilities. The calculation changes significantly if the sampling is done with repetition (as was used in [18]) vs. without repetition. We refer the reader to [18] for more details on the core algorithm.

3 From Theory to Practice

In this section we present our work to actually deploying the Unseen estimation method for real world deduplication estimation. Throughout the section we demonstrate the validity of our results using tests on a single dataset. This is done for clarity of the exposition and only serves as a representative of the results that where tested across all our workloads. The dataset is the Linux Hypervisor data (see Table 1 in Section 4) that was also used in Figure 1. The entire scope of results on all workloads appears in the evaluation section (Section 4).

3.1 Gauging the Accuracy

We tested the core Unseen algorithm on real life workloads and it has impressive results, and in general it thoroughly outperforms some of the estimators in the literature. The overall impression is that a 15% sample is sufficient for accurate estimation. On the other hand the accuracy level varies greatly from one workload to the next, and often the estimation obtained from 5% or even less is sufficient for all practical purposes. See example in Figure 2.

So the question remains: how to interpret the estimation result and when have we sampled enough? To address these questions we devise a new approach that returns a range of plausible deduplica-

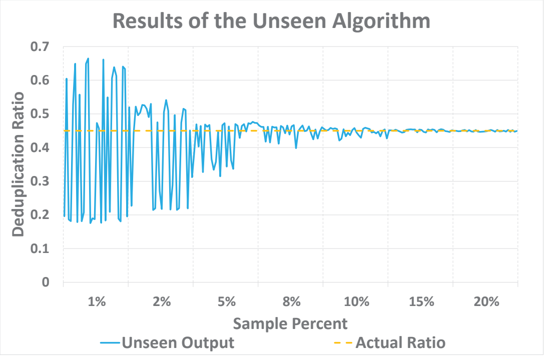


Figure 2: The figure depicts sampling trials at each of the following sample percentages 1%, 2%, 5%, 7.5%, 10%. 15% and 20%. For each percentage we show the results of Unseen on 30 independent samples. The results are very noisy at first, for example, on the 1% samples they range from ~ 0.17 all the way to ~ 0.67. But we see a nice convergence starting at 7.5%. The problem is that seeing just a single sample and not 30, gives little indication about the accuracy and convergence.

tion ratios rather than a single estimation number. In a nut shell, the idea is that rather than give the DFH that is the ”best explanation” to the observed y, we test all the DFHs that are a ”reasonable explanation” of y and identify the range of possible duplication in these plausible DFHs.

Technically, define criteria for all plausible solutions x' that can explain an observed sample DFH y. Of all these plausible solutions we find the ones which give minimal deduplication ratio and maximal deduplication ratio. In practice, we add two additional linear programs to the first initial optimization. The first linear program helps in identifying the neighborhood of plausible solutions. The second and third linear programs find the two limits to the plausible range. In these linear programs we replace the optimization on the distance measure with an optimization on the number of distinct chunks. The method is outlined in Algorithm 2.

Note that in [18] there is also a use of a second linear program for entropy estimation, but this is done for a different purpose (implementing a sort of Occam's razor).

Why does it work? We next describe the intuition behind our solution: Consider the distribution of $\Delta_p(x, y)$ for a fixed x and under random y (random y means a DFH of a randomly chose p-sample). Suppose that we knew the expectancy $E(\Delta_p)$ and standard deviation σ of Δ_p. Then given an observed y, we expect, with very high probability, that the only plausible source DFHs x' are such that $\Delta_p(x', y)$ is close to $E(\Delta_p)$ (within $\alpha \cdot \sigma$ for some slackness variable α). This set of plausible x' can be fairly large, but all we really care to learn about it is its boundaries in terms of deduplication ratio. The second and third linear programs find out of this set of plausible DFHs the ones with the best and worst deduplication ratios .

Algorithm 2: Unseen Range

```
Input: Sample DFH y, fraction p, total size N,
       slackness $\alpha$ (default $\alpha = 0.5$)
Output: Deduplication ratio range $[\underline{\mathbf{r}}, \overline{\mathbf{r}}]$

/* Prepare expected DFH transformation*/
    $A_p \leftarrow$ prepareExpectedA(p);

$1^{st}$ Linear program:
    Find $x'$ that minimizes: $\Delta_p(x', y)$
    Under constraints: /* $x'$ is a legal DFH */
        $\sum_i x'_i \cdot i = N$ and $\forall i\ x'_i \geq 0$
For the resulting $x'$ compute: $Opt = \Delta_p(x', y)$
$2^{nd}$ Linear program:
    Find $\underline{x}$ that minimizes: $\sum_i \underline{x}_i$
    Under constraints:
        $\sum_i \underline{x}_i \cdot i = N$ and $\forall i\ \underline{x}_i \geq 0$ and
          $\Delta_p(\underline{x}, y) < Opt + \alpha\sqrt{Opt}$
$3^{rd}$ Linear program:
    Find $\overline{x}$ that maximizes: $\sum_i \overline{x}_i$
    Under constraints:
        $\sum_i \overline{x}_i \cdot i = N$ and $\forall i\ \overline{x}_i \geq 0$ and
          $\Delta_p(\overline{x}, y) < Opt + \alpha\sqrt{Opt}$

return $[\underline{r} = \frac{\sum_i \underline{x}_i}{N}, \overline{r} = \frac{\sum_i \overline{x}_i}{N}]$
```

The problem is that we do not know how to cleanly compute the expectancy and standard deviation of Δ_p, so we use the first linear program to give us a single value within the plausible range. We use this result to estimate the expectancy and standard deviation and give bounds on the range of plausible DFHs.

Setting the slackness parameter.. The main tool that we have in order to fine tune the plausible solutions set is the slackness parameter α. A small α will result in a tighter estimation range, yet risks having the actual ratio fall outside of the range. Our choice of slackness parameter is heuristic and tailored to the desired level of confidence. The choices of this parameter throughout the paper are made by thorough testing across all of our datasets and the various sample sizes. Our evaluations show that a slackness of $\alpha = 0.5$ is sufficient and one can choose a slightly larger number for playing it safe. A possible approach is to use two different levels of slackness and present a “likely range” along with a “safe range”.

In Figure 3 we see an evaluation of the range method. We see that our upper and lower bounds give an excellent estimation of the range of possible results that the plain Unseen algorithm would have produced on random samples of the given fraction.

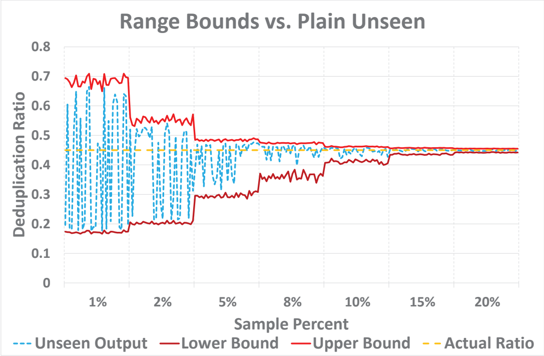


Figure 3: This figure depicts the same test as in Figure 2, but adds the range algorithm results. This gives a much clearer view – For example, one can deduce that the deduplication ratio is better than 50% already at a 5% sample and that the range has converged significantly at this point, and is very tight already at 10%.

An interesting note is that unlike many statistical estimation in which the actual result has a high likelihood to be at the center of the range, in our case all values in the range can be equally likely.

Evaluating methods via average range tightness. The range approach is also handy in comparing the success of various techniques. We can evaluate the average range size of two different methods and choose the one that gives tighter average range size using same sample percentage. For example we compare running the algorithm when the sampling is with repetitions (this was the approach taken in [18]) versus taking the sample without repetitions. As mentioned in Section 2.2, this entails a different computation of the matrix A_p. Not surprisingly, taking samples without repetitions is more successful, as seen in Figure 4. This is intuitive since repetitions reduce the amount of information collected in a sample. Note that sampling without repetition is conceptually simpler since it can be done in a totally stateless manner (see Section 3.4). Since the results in Figure 4 were consistent with other workloads, we focus our attention from here on solely on the no repetition paradigm.

3.2 Running with Low Memory

Running the estimation with a 15% sample requires the creation of a DFH for the entire sample, which in turn requires keeping tab on the duplication frequencies of all distinct elements in the sample. Much like the case of full scans, this quickly becomes an obstacle in actually deploying the algorithm. For example, in our largest test data set, that would mean keeping tab on approximately 200 Million distinct chunks, which under very strict assumptions would require on the order of 10GBs of RAM, unless one is willing to settle for slow disk IOs instead of RAM operations. Moreover, in a distributed setting it would require moving GBs of data between nodes. Such high resource consumption may be feasible in a dedicated system, but not for actually determining deduplication ratios in the field, possibly at a customer’s site and on the customers own servers.

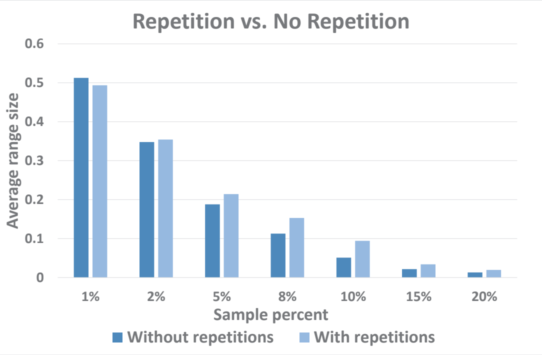


Figure 4: An average range size comparison of sampling with repetitions vs. without repetitions. For each sampling percentage the averages are computed over 100 samples. Other than the 1% sample which is bad to begin with, the no repetition sampling is consistently better.

We present two approaches in order to handle this issue, both allowing the algorithm to run with as little as 10MBs of RAM. The first achieves relatively tight estimations (comparable to the high memory algorithms). The second produces somewhat looser estimations but is more flexible to usage in distributed or dynamic settings.

The base sample approach. This approach follows the low-memory technique of [12] and uses it to estimate the DFH using low memory. In this method we add an additional base step so the process is as follows:

1. **Base sample:** Sample C chunks from the data set (C is a “constant” – a relatively small number, independent of the database size). Note that we allow the same hash value to appear more than once in the base sample.
2. **Sample and maintain low-memory chunk histogram:** Sample a p fraction of the chunks and iterate over all the chunks in the sample. Record a histogram (duplication counts) for all the chunks in the base sample (and ignore the rest). Denote by c_j the duplication count of the j^{th} chunk in the base sample ($j \in \{1, ..., C\}$).

3. **Extrapolate DFH:** Generate an estimated DFH for the sample as follows:

   $$\forall i,\ y_i = \frac{|\{j|c_j = i\}|}{i}\frac{pN}{C}.$$

   In words, use the number of chunks in the base sample that had count i, extrapolated to the entire sample.

The crux is that the low-memory chunk histogram can produce a good approximation to the DFH. This is because the base sample was representative of the distribution of chunks in the entire dataset. In our tests we used a base sample of size $C = 50,000$ which amounts to less than 10MBs of memory consumption. The method does, however, add another estimation step to the process and this adds noise to the overall result. To cope with it we need to increase the slackness parameter in the Range Unseen algorithm (from $\alpha = 0.5$ to $\alpha = 2.5$). As a result, the estimation range suffers a slight increase, but the overall performance is still very good as seen in Figure 5.

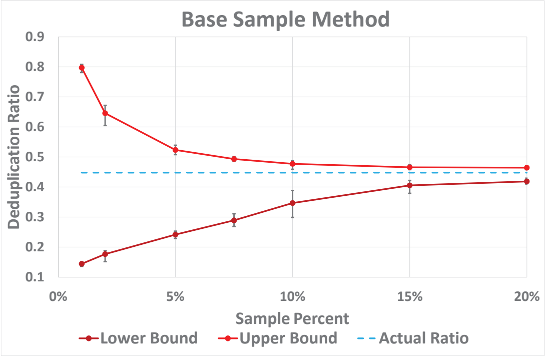


Figure 5: The graph depicts the average upper and lower bounds that are given by the algorithm when using the base sample method. Each point is the average over 100 different samples, and the error bars depict the maximum and minimum values over the 100 tests.

The only shortcoming of this approach is that the dataset to be studied needs to be set in advance, otherwise the base sample will not cover all of it. In terms of distribution and parallel execution, the base sample stage needs to be finished and finalized before running the actual sampling phase which is the predominant part of the work (this main phase can then be easily parallelized). To overcome this we present a second approach, that is more dynamic and amenable to parallelism yet less tight.

A streaming approach. This method uses techniques from streaming algorithms geared towards distinct elements evaluation with low memory. In order to mesh with the Unseen method the basic technique needs to be slightly modified and collect frequency counts that were otherwise redundant. The core principles, however, remain the same: A small (constant size) sample of distinct chunks is taken *uniformly over the distinct chunks* in the sample. Note that such a sampling disregards the popularity of a specific hash value, and so the most popular chunks will likely not be part of the sample. As a result, this method cannot estimate the sample DFH correctly but rather takes a different approach. Distinct chunk sampling can be done using several techniques (e.g. [2, 8]). We use here the technique of [2] where only the C chunks that have the highest hash values (when ordered lexicographically) are considered in the sample. The algorithm is then as follows:

1. **Sample and maintain low-memory chunk histogram:** Sample a p fraction of the chunks and iterate over all the chunks in the sample. Maintain a histogram only of chunks that have one of the C highest hash values:
   - If the hash is in the top C, increase its counter.
   - If it is smaller than all the C currently in the histogram then ignore it.
   - Otherwise, add it to the histogram and discard the lowest of the current C hashes.

   Denote by δ the fraction of the hash domain that was covered by the C samples. Namely, if the hashes are calibrated to be numbers in the range $[0, 1]$ then δ is the distance between 1 and the lowest hash in the top-C histogram.
2. **Run Unseen:** Generate a DFH solely of the C top hashes and run the Range Unseen algorithm. But rather than output ratios, output a range estimation on the number of distinct chunks. Denote this output range by $[\underline{d}, \overline{d}]$.
3. **Extrapolation to full sample:** Output estimation range $[\underline{r} = \frac{\underline{d}}{\delta \cdot N}, \overline{r} = \frac{\overline{d}}{\delta \cdot N}]$.

Unlike the base sample method, the streaming approach does not attempt to estimate the DFH of the p-sample. Instead, it uses an exact DFH of a small δ fraction of the hash domain. The Unseen algorithm then serves as a mean of estimating the actual number of distinct hashes in this δ sized portion of the hash domain. The result is then extrapolated from the number of distinct chunks in a small hash domain, to the number of hashes in the entire domain. This relies on the fact that hashes should be evenly distributed over the entire range, and a δ fraction of the domain should hold approximately a δ portion of the distinct hashes.

The problem here is that the Unseen algorithm runs on a substantially smaller fraction of the data than originally. Recall that it was shown in [18] that accuracy is achieved at a sample fraction of $O(\frac{1}{\log N})$ and therefore we expect the accuracy to be better when N is larger. Indeed, when limiting the input of Unseen to such a small domain (in some of our tests the domain is reduced by a factor of more than $20,000$) then the tightness of the estimation suffers. Figure 6 shows an example of the estimation achieved with this method.

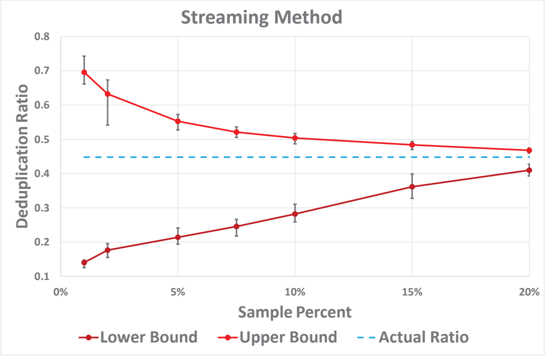


Figure 6: The graph depicts the average upper and lower bounds that are given by the algorithm with the streaming model as a function of sample percentage. Each point is the average over 100 different samples, and the error bars depict the maximum and minimum values over the 100 tests.

In Figure 7 we compare the tightness of the estimation achieved by the two low-memory approaches. Both methods give looser results than the full fledged method, but the base sampling technique is significantly tighter.

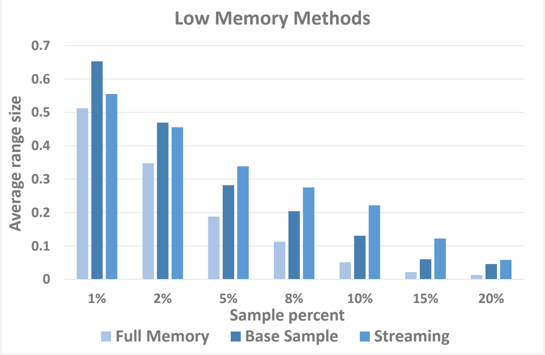


Figure 7: An average range size comparison of the two low-memory methods (and compared to the full memory algorithm). We see that the base method achieves tighter estimation ranges, especially for the higher and more meaningful percentages.

On the flip side, the streaming approach is much simpler to use in parallel environments where each node can run his sample independently and at the end all results are merged and a single Unseen execution is run. Another benefit is that one can run an estimation on a certain set of volumes and store the low-memory histogram. Then, at a later stage, new volumes can be scanned and merged with the existing results to get an updated estimation. Although the streaming approach requires a larger sample in order to reach the same level of accuracy, there are scenarios where the base sample method cannot be used and this method can serve as a good fallback option.

3.3 Estimating Combined Compression and Deduplication

Deduplication, more often than not, is used in conjunction with compression. The typical usage is to first apply deduplication and then store the actual chunks in compressed fashion. Thus the challenge of sizing a system must take into account compression as well as deduplication. The obvious solution to estimating the combination of the two techniques is by estimating each one separately and then looking at their multiplied effect. While this practice has its merits (e.g. see [6]), it is often imprecise. The reason is that in some workloads there is a correlation between the duplication level of a chunk and its average compression ratio (e.g. see Figure 8).

We next describe a method of integrating compression into the Unseen algorithm that results in accurate estimations of this combination.

The basic principle is to replace the DFH by a **compression weighted DFH**. Rather than having x_i hold the number of chunks that appeared i times, we define it as the size (in chunks) that it takes to store the chunks with reference count i. Or in other words, multiply each count in the regular DFH by the average compression ratio of chunks with the specific duplication count.

The problem is that this is no longer a legal DFH and in particular it no longer holds that

$$\sum_i x_i \cdot i = N.$$

Instead, it holds that

$$\sum_i x_i \cdot i = \mathrm{CR} \cdot N$$

where CR is the average compression ratio over the entire dataset (plain compression without deduplication). Luckily, the average compression ratio can be estimated extremely well with a small random sample of the data.

The high level algorithm is then as follows:

1. Compute a DFH $\{y_1, y_2, ...\}$ on the observed sample, but also compute $\{\mathrm{CR}_1, \mathrm{CR}_2, ...\}$ where CR_i is the average compression ratio for all chunks that had reference count i. Denote by $z = \{y_1 \cdot \mathrm{CR}_1, y_2 \cdot \mathrm{CR}_2, ...\}$ the compression weighted DFH of the sample.

2. Compute CR, the average compression ratio on the dataset. This can be done using a very small random sample (which can be part of the already sampled data).
3. Run the Unseen method where the optimization is for $\Delta_p(x,z)$ (rather than $\Delta(x,y)_p$) and under the constraint that $\sum_i x_i \cdot i = \mathrm{CR} \cdot N$.

Figure 8 shows the success of this method on the same dataset and contrasts it to the naive approach of looking at deduplication and compression independently.

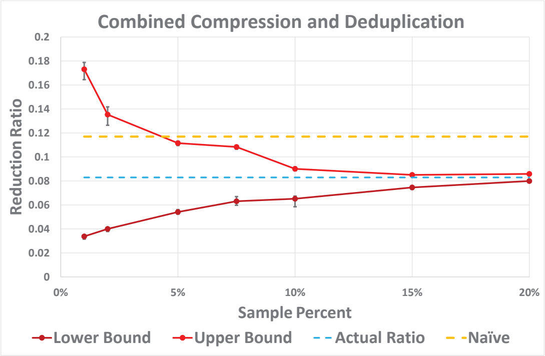


Figure 8: The graph depicts the average upper and lower bounds that are given by the algorithm with compression. Each point is based on 100 different samples. The naive ratio is derived by multiplying independent compression and deduplication ratios.

Note that estimating CR can also be done efficiently and effectively under our two models of low-memory execution: In the base sample method, taking the average compression ratio on the base chunks only is sufficient. So compression needs to be computed only in the initial small base sample phase. In the streaming case, things are a bit more complicated, but in a nutshell, the average CR for the chunks in the fraction of hashes at hand is estimated as the weighted average of the compression ratios of the chunks in the top C hashes, were the weight is their reference counts (a chunk that appeared twice is given double the weight).

3.4 How to Sample in the Real World

Thus far we have avoided the question of how to actually sample chunks from the dataset, yet our sampling has a long list of requirements:

- Sample uniformly at random over the entire (possibly distributed) dataset.
- Sample without repetitions.
- Use low memory for the actual sampling.
- We want the option to do a gradual sample, e.g., first sample a small percent, evaluate it, and then add more samples if needed (without repeating old samples).
- Above all, we need this to be substantially faster than running a full scan (otherwise there is no gain). Recall that the scan time dominates the running time (the time to solve the linear programs is negligible).

The last requirement is the trickiest of them all, especially if the storage medium is based on rotating disks (HDDs). If the data lies on a storage system that supports fast short random reads (flash or solid state drive based systems), then sampling is much faster than a full sequential scan. The problem is that in HDDs there is a massive drop-off from the performance of sequential reads to that of small random reads.

There are some ways to mitigate this drop off: sorting the reads in ascending order is helpful, but mainly reading at larger chunks than 4KB, where 1MB seems the tipping point. In Figure 9 we see measurements on the time it takes to sample a fraction of the data vs. the time a full scan would take. While it is very hard to gain anything by sampling at 4KB chunks, there are significant time savings in sampling at 1MBs, and for instance, sampling a 15% fraction of the data is 3X faster than a full scan (this is assuming sorted 1MB reads).

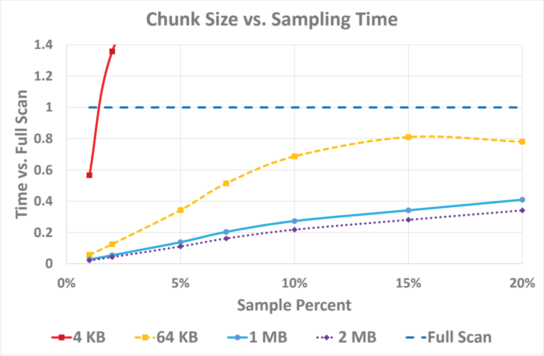


Figure 9: The graph depicts the time it takes to sample different percentages at different chunk sizes with respect to a full scan. We see that sampling at 4K is extremely slow, and while 64K is better, one has to climb up to 1MB to get decent savings. Going higher than 1MB shows little improvement. The tests where run on an Intel Xeon E5440 @ 2.83GHz CPU with a 300GB Hitachi GST Ultrastar 15K rpm HDD.

Accuracy with 1MB reads? The main question is then: does our methodology work with 1MB reads? Clearly, estimating deduplication at a 1MB granularity is not a viable solution since deduplication ratios can change drastically with the chunk size. Instead we read super-chunks of 1MB and break them into 4KB chunks and use these correlated chunks for our sample. The main concern here is that the fact that samples are not independent will form

Name	*Description*	*Size*	*Deduplication Ratio*	*Deduplication + Compression*
VM Repository	A large repository of VMs used by a development unit. This is a VMWare environment with a mixture of Linux and Windows VMs.	8TB	0.3788	0.2134
Linux Hypervisor	Backend store for Linux VMs of a KVM Hypervisor. The VMs belong to a research unit.	370 GB	0.4499	0.08292
Windows Hypervisor	Backend store for Windows VMs of a KVM Hypervisor. The VMs belong to a research unit.	750 GB	0.7761	0.4167
VDI	A VDI benchmark environment containing 50 Windows VMs generated by VMWare's ViewPlanner tool	770 GB	0.029	0.0087
DB	An Oracle Data Base containing data from a TPCC benchmark	1.2TB	0.37884	0.21341
Cloud Store	A research unit's private cloud storage. Includes both user data and VM images.	3.3TB	0.26171	–

Table 1: Data generation approaches for several widely adopted benchmarks in various storage domains.

high correlations between the reference counts of the various chunks in the sample. For example, in many deduplication friendly environments, the repetitions are quite long, and a repetition of a single chunk often entails a repetition of the entire super-chunk (and vice-versa, a non repetition of a single chunk could mean high probability of no repetitions in the super-chunk).

The good news is that due to linearity of expectations, the expected DFH should not change by sampling at a large super-chunk. On the other hand the variance can grow significantly. As before, we control this by increasing the slackness parameter α to allow a larger scope of plausible solutions to be examined. In our tests we raise the slackness from $\alpha = 0.5$ to $\alpha = 2.0$ and if combined with the base sample it is raised to $\alpha = 3.5$. Our tests show that this is sufficient to handle the increased variance, even in workloads where we know that there are extremely high correlations in the repetitions. Figure 10 shows the algorithm result when reading with 1MB super-chunks and with the increased slackness.

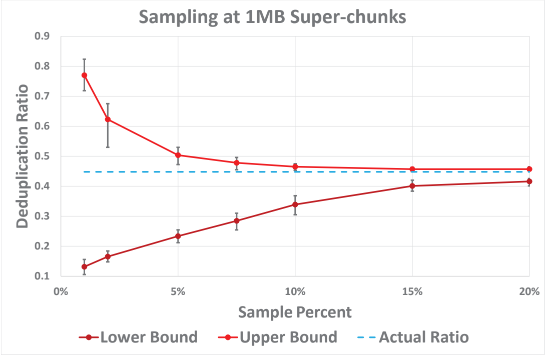


Figure 10: The graph depicts the average upper and lower bounds that are given by the algorithm when reading at 1MB super-chunks. Each point is based on 100 different samples.

How to sample? We next present our sampling strategy that fulfills all of the requirements listed above with the additional requirement to generate *sorted* reads of a configurable chunk size. The process iterates over all chunk IDs in the system and computes a fast hash function on the chunk ID (the ID has to be a unique identifier, e.g. volume name and chunk offset). The hash can be a very efficient function like CRC (we use a simple linear function modulo a large number). This hash value is then used to determine if the chunk is in the current sample or not. The simple algorithm is outlined in Algorithm 3.

```
Algorithm 3: Simple Sample
Input: Fraction bounds $p_0, p_1$, Total chunks M
Output: Sample $\mathcal{S}_{(p_1 - p_0)}$

for $j \in [M]$ do
    $q \longleftarrow FastHash(j)$
    /* FastHash outputs a number in $[0, 1)$ */
    if $p_0 \leq q < p_1$ then
        Add $j^{th}$ chunk to sample
```

This simple technique fulfills all of the requirements that we listed above. It can be easily parallelized and in fact there is no limitation on the enumeration order. However, within each disk, if one iterates in ascending order then the reads will come out sorted, as required. It is nearly stateless, one only need to remember the current index j and the input parameters. In order to run a gradual sample, for example, a first sample of 1% and then add another 5% – run it first with $p_0 = 0, p_1 = 0.01$ and then again with $p_0 = 0.01, p_1 = 0.06$. The fast hash is only required to be sufficiently random (any pairwise independent hash would suffice [3]) and there is no need for a heavy full-fledged cryptographic hash like SHA1. As a result, the main loop can be extremely fast, and our tests show that the overhead of the iteration and hash is negligible (less than 0.5% of the time that it takes to sample a 1% fraction of the dataset). Note that the result is a sample of fraction *approximately* $(p_1 - p_0)$ which is sufficient for all practical means.

4 Evaluation

4.1 Implementation and Test Data

We implemented the core techniques in Matlab and evaluated the implementation on data from a variety of real life datasets that are customary to enterprise storage systems. The different datasets are listed in Table 1 along with their data reduction ratios. It was paramount to tests datasets from a variety of deduplication and compression ratios in order to validate that our techniques are accurate for all ranges. Note that in our datasets we remove all zero chunks since identifying the fraction of zero chunks is an easy task and the main challenge is estimating the deduplication ratio on the rest of the data.

For each of the datasets we generated between 30-100 independent random samples for each sample percentage and for each of the relevant sampling strategy being deployed (e.g., with or without repetitions/ at 1MB super-chunks). These samples serve as the base for verifying our methods and fine tuning the slackness parameter.

4.2 Results

Range sizes as function of dataset. The most glaring phenomena that we observe while testing the technique over multiple datasets is the big discrepancy in the size of the estimation ranges for different datasets. The most defining factor was the data reduction ratio at hand. It turns out that deduplication ratios that are close to $\frac{1}{2}$ are in general harder to approximate accurately and require a larger sample fraction in order to get a tight estimation. Highly dedupable data and data with no duplication, on the other hand tend to converge very quickly and using our method, one can get a very good read within the first 1-2% of sampled data. Figure 12 shows this phenomena clearly.

Note that the addition of compression ratios in the mix has a different effect and it basically reduces the range by a roughly a constant factor that is tied to the compression benefits. For example, for the Windows Hypervisor data the combine deduplication and compression ratio is 0.41, an area where the estimation is hardest. But the convergence seen by the algorithm is significantly better – it is similar to what is seen for deduplication (0.77 ratio) with a roughly constant reduction factor. See example in Figure 13. According to the other dataset we observe that this reduction factor is more significant when the compression ratio is better (as seen in other datasets).

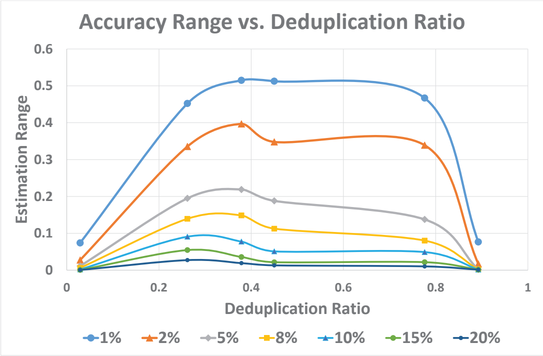


Figure 12: The graph depicts the effect that the deduplication ratio has on the tightness of our estimation method (based on the 6 different datasets and their various deduplication ratios). Each line stands for a specific sample percent and charts the average range size as a function of the deduplication ratio. We see that while the bottom lines of 10% and more are good across all deduplication ratios, the top lines of 1-5% are more like a "Boa digesting an Elephant" – behave very well at the edges but balloon in the middle.

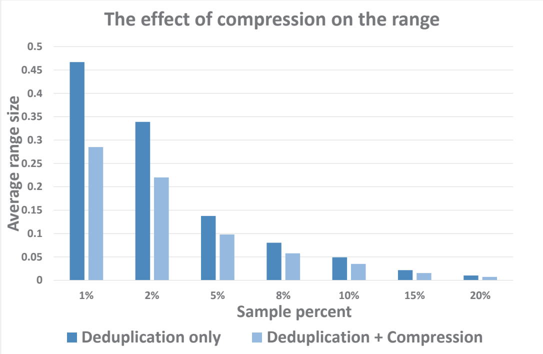


Figure 13: A comparison of the estimation range sizes achieved on the Windows Hypervisor data. We see a constant and significantly tighter estimation when compression is involved. This phenomena holds for all datasets.

The accumulated effect on estimation tightness. In this work we present two main techniques that are critical enablers for the technology but reduce the tightness of the initial estimation. The accumulated effect on the tightness of estimation by using the combination of these techniques is shown in Figure 14. It is interesting to note that combining the two techniques has a smaller negative effect on the tightness than the sum of their separate effects.

Putting it all together. Throughout the paper we evaluated the effect of each of our innovations separately and sometimes understanding joint effects. In this section we aim to put all of our techniques together and show a functional result. The final construction runs the *Range Unseen* algorithm, with the *Base Sample* low-memory technique while sampling at *1MB super-chunks*. We test both estimating deduplication only and estimation with compression. Each test consists of a single gradual execu-

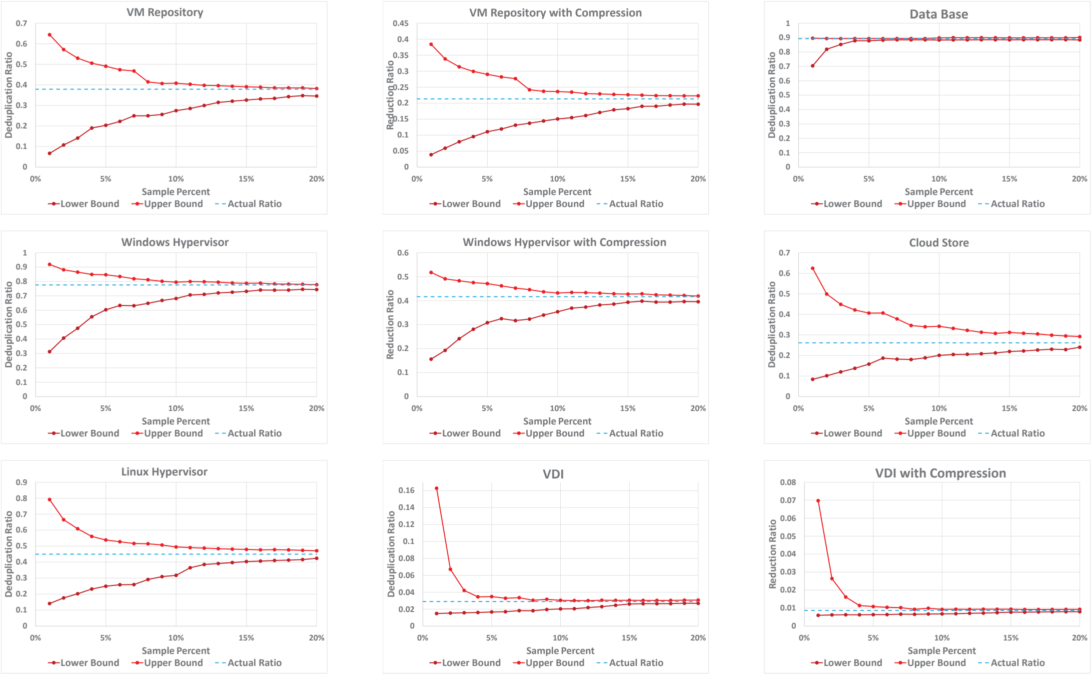


Figure 11: Execution of the full real life setting algorithm on the various datasets. This is a gradual run of the low-memory, 1MB read algorithm, with and without compression. Note that some of the tests reach very tight results with a 3% sample and can thus achieve a much improved running time.

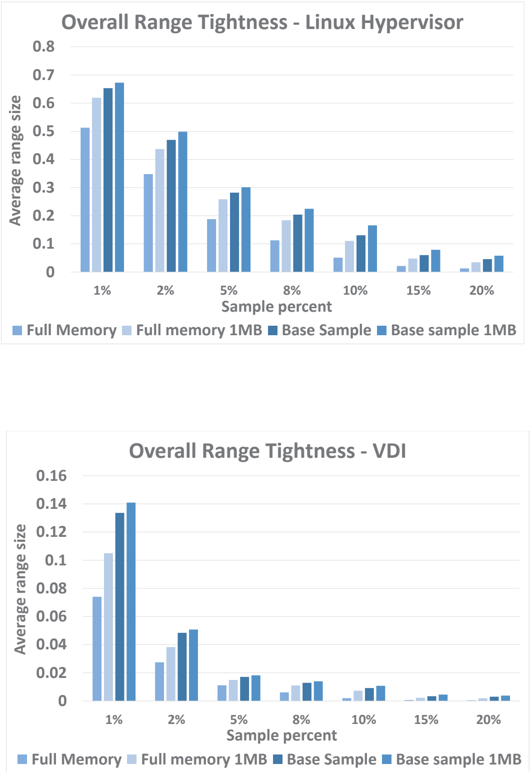


Figure 14: A comparison of the estimation range sizes achieved with various work methods: The basic method, the method with sampling at 1MB, using the base sample low-memory technique and the combination of both the base sample and 1MB sampling. There is a steady decline in tightness when moving from one to the next. This is shown on two different datasets with very different deduplication ratios.

tion starting from 1% all the way through 20% at small intervals of 1%. The results are depicted in Figure 11.

There are two main conclusions from the results in Figure 11. One is that the method actually works and produces accurate and useful results. The second is the great variance between the different runs and the fact that some runs can end well short of a 5% sample. As mentioned in the previous paragraph, this is mostly related to the deduplication and compression ratios involved. But the bottom line is that we can calibrate this into the expected time it takes to run the estimation. In the worst case, one would have to run read at least 15% of the data, which leads to a time improvement of approximately 3X in HDD systems (see Section 3.4). On the other hand, we have tests that can end with a sample of 2-3% and yield a time saving of 15-20X over a full scan. The time improvement can be even more significant in cases where the data resides on SSDs and if the hash computation is a bottle neck in the system.

5 Concluding remarks

Our work introduced new advanced algorithms into the world of deduplication estimation. The main challenges were to make these techniques actually applicable and worthwhile in a real world scenario. We believe we have succeeded in proving the value of this approach, which can be used to replace full scans used today.

Acknowledgements. We thank Evgeny Lipovetsky for his help and thank Oded Margalit and David Chambliss for their insights. Finally, we are grateful to our many colleagues at IBM that made the collections of the various datasets possible. This work was partially funded by the European Community's Seventh Framework Programme (FP7/2007-2013) project ForgetIT under grant No. 600826.

References

[1] XtremeIO case studies. `http://xtremio.com/case-studies`.

[2] 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.

[3] CARTER, L., AND WEGMAN, M. N. Universal classes of hash functions. *J. Comput. Syst. Sci. 18*, 2 (1979), 143–154.

[4] CHARIKAR, M., CHAUDHURI, S., MOTWANI, R., AND NARASAYYA, V. R. Towards estimation error guarantees for distinct values. In *Symposium on Principles of Database Systems, PODS 2010* (2000), pp. 268–279.

[5] CONSTANTINESCU, C., GLIDER, J. S., AND CHAMBLISS, D. D. Mixing deduplication and compression on active data sets. In *Data Compression Conference, DCC* (2011), pp. 393–402.

[6] CONSTANTINESCU, C., AND LU, M. Quick Estimation of Data Compression and Deduplication for Large Storage Systems. In *Proceedings of the 2011 First International Conference on Data Compression, Communications and Processing* (2011), IEEE, pp. 98–102.

[7] FLAJOLET, P., AND MARTIN, G. N. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci. 31*, 2 (1985), 182–209.

[8] GIBBONS, P. B. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Very Large Data Bases VLDB* (2001), pp. 541–550.

[9] HAAS, P. J., NAUGHTON, J. F., SESHADRI, S., AND STOKES, L. Sampling-based estimation of the number of distinct values of an attribute. In *Very Large Data Bases VLDB'95* (1995), pp. 311–322.

[10] HAAS, P. J., AND STOKES, L. Estimating the number of classes in a finite population. *IBM Research Report RJ 10025, IBM Almaden Research 93* (1998), 1475–1487.

[11] HARNIK, D., KAT, R., MARGALIT, O., SOTNIKOV, D., AND TRAEGER, A. To Zip or Not to Zip: Effective Resource Usage for Real-Time Compression. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST 2013)* (2013), USENIX Association, pp. 229–241.

[12] 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.

[13] JIN, K., AND MILLER, E. L. The effectiveness of deduplication on virtual machine disk images. In *The Israeli Experimental Systems Conference, SYSTOR 2009* (2009).

[14] KANE, D. M., NELSON, J., AND WOODRUFF, D. P. An optimal algorithm for the distinct elements problem. In *Symposium on Principles of Database Systems, PODS 2010* (2010), pp. 41–52.

[15] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *FAST-Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)* (2011), pp. 1–13.

[16] RASKHODNIKOVA, S., RON, D., SHPILKA, A., AND SMITH, A. Strong lower bounds for approximating distribution support size and the distinct elements problem. *SIAM J. Comput. 39*, 3 (2009), 813–842.

[17] VALIANT, G., AND VALIANT, P. Estimating the unseen: an n/log(n)-sample estimator for entropy and support size, shown optimal via new clts. In *43rd ACM Symposium on Theory of Computing, STOC* (2011), pp. 685–694.

[18] VALIANT, P., AND VALIANT, G. Estimating the unseen: Improved estimators for entropy and other properties. In *Advances in Neural Information Processing Systems 26*. 2013, pp. 2157–2165.

[19] 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.

OrderMergeDedup: Efficient, Failure-Consistent Deduplication on Flash

Zhuan Chen and Kai Shen
Department of Computer Science, University of Rochester

Abstract

Flash storage is commonplace on mobile devices, sensors, and cloud servers. I/O deduplication is beneficial for saving the storage space and reducing expensive Flash writes. This paper presents a new approach, called *OrderMergeDedup*, that deduplicates storage writes while realizing failure-consistency, efficiency, and persistence at the same time. We devise a soft updates-style metadata write ordering that maintains storage data consistency without consistency-induced additional I/O. We further explore opportunities of I/O delay and merging to reduce the metadata I/O writes. We evaluate our Linux device mapper-based implementation using several mobile and server workloads—package installation and update, BBench web browsing, vehicle counting, Hadoop, and Yahoo Cloud Serving Benchmark. Results show that OrderMergeDedup can realize 18–63% write reduction on workloads that exhibit 23–73% write content duplication. It has significantly less metadata write overhead than alternative I/O shadowing-based deduplication. Our approach has a slight impact on the application latency and may even improve the performance due to reduced I/O load.

1 Introduction

I/O deduplication [4, 7, 17, 20, 21, 23] has been widely employed to save storage space and I/O load. I/O deduplication is beneficial for storage servers in data centers, as well as for personal devices and field-deployed sensing systems. Flash writes on smartphones and tablets may occur during package installation or through the frequent use of SQLite transactions [11, 19]. In cyber-physical systems, high volumes of data may be captured by field-deployed cameras and stored / processed for applications like intelligent transportation [22].

A deduplication system maintains metadata such as logical to physical block mapping, physical block reference counters, block fingerprints, etc. Such metadata and data structures must remain consistent on storage after system failures. It is further essential to persist writes in a prompt manner to satisfy the storage durability semantics. On Flash storage, I/O deduplication must also minimize the expensive Flash writes resulted from metadata management. This paper presents a new I/O deduplication mechanism that meets these goals.

Specifically, we order the deduplication metadata and data writes carefully so that any fail-stop failure produces no other data inconsistency (on durable storage) than uncollected garbage. This approach is, in concept, similar to soft updates-based file system design [6]. It is efficient by not requiring additional I/O traffic for maintaining consistency (in contrast to logging / journaling or shadowing-based I/O atomicity). While the original file system soft updates suffer from dependency cycles and rollbacks [18], the relatively simple structure of deduplicated storage allows us to recognize and remove all possible dependency cycles with no impact on performance.

The metadata I/O overhead can be further reduced by merging multiple logical writes that share common deduplication metadata blocks. In particular, we sometimes delay I/O operations in anticipation for metadata I/O merging opportunities in the near future. Anticipatory I/O delay and merging may prolong responses to the user if the delayed I/O is waited on due to the data persistence semantics. We show that the performance impact is slight when the delay is limited to a short duration. It may even improve the application latency due to reduced I/O load. With failure-consistent I/O ordering and anticipatory merging, we name our deduplication approach *OrderMergeDedup*.

We have implemented our OrderMergeDedup approach in Linux 3.14.29 kernel as a custom device mapper target. Our prototype system runs on an Intel Atom-based tablet computer and an Intel Xeon server machine. We have experimentally evaluated our system using a range of mobile and server workloads.

Data consistency over failures was not ignored in prior deduplication systems. iDedup [20] relied on a non-volatile RAM to stage writes (in a log-structured fashion) that can survive system failures. Other systems [4, 7] utilized supercapacitors or batteries to allow continued data maintenance after power failures. Our deduplication approach does not assume the availability of such hardware aids. Venti [17] provided consistency checking and repair tools that can recover from failures at a significant cost in time. dedupv1 [13] maintained a redo log to recover from failures but redo logging incurs the cost of additional writes (even at the absence of failures). Most recently, Dmdedup [21] supported data consistency after failures through I/O shadowing, which incurs the cost of additional index block writes. It achieved efficiency by delaying and batch-flushing a large number of metadata updates but such delayed batching is hindered by synchronous writes in some applications and databases.

2 Design of OrderMergeDedup

I/O deduplication eliminates duplicate writes in the I/O stream. We capture all write I/O blocks at the device layer for deduplication. With a fixed-sized chunking approach, each 4 KB incoming data block is intercepted and a hashed fingerprint is computed from its content. This fingerprint is looked up against the fingerprints of existing storage blocks to identify duplicates.

A deduplication system maintains additional metadata information. Specifically, a logical-to-physical block mapping directs a logical block access (with its logical address) to its physical content on storage. For each physical block, the associated reference counter records the number of logical blocks mapped to it, and the fingerprint is computed to facilitate the block content matching. A write request received by a deduplication system can result in a series of physical writes to both the block data and metadata. For deduplication metadata management, it is challenging to realize (1) *failure consistency*—data / metadata writes must be carefully performed to enable fast, consistent recovery after failures; (2) *efficiency*—the additional I/O cost incurred by metadata writes should not significantly diminish deduplication I/O saving; (3) *persistence*—the deduplication layer should not prematurely return an I/O write in violation of persistence semantics.

2.1 I/O Ordering for Failure-Consistency

File and storage systems [6, 9] have recognized the importance of atomic I/O to support consistent failure-recovery. Existing techniques include journaling, shadowing [1, 9], and soft updates [6]. In journaling, an atomic I/O operation is recorded in a redo log before writing to the file system. A failure after a partial write can be recovered at system restart by running the redo log. In shadowing, writes to existing files are handled in a copy-on-write fashion to temporary shadow blocks. The final commit is realized through one atomic I/O write to a file index block that points to updated shadow data/index blocks. Index blocks (potentially at multiple hierarchy levels) must be re-written to create a complete shadow. Both journaling and shadowing require additional write I/O to achieve failure consistency of durable data.

The soft updates approach [6] carefully orders writes in file system operations such that any mid-operation failure always leaves the file system structure in a consistent state (except for possible space leaking on temporarily written blocks). While it requires no I/O overhead during normal operations, rollbacks may be necessary to resolve cyclic dependencies in the block commit order. Seltzer et al. [18] showed that such rollbacks sometimes led to poor soft updates performance on the UNIX Fast File System. Due to relatively simple semantics of a deduplicated storage (compared to a file system), we show that a careful design of all deduplication I/O paths can efficiently resolve possible dependency cycles. We next present our soft updates-style deduplication design.

A unique aspect of our design is that our physical block reference counter counts logical block references as well as a reference from the physical block's fingerprint. Consequently the reclamation of block fingerprint does not have to occur together with the removal of the last logic block referencing the physical block. Separating them into two failure-consistent transactions makes each less complex and reduces the probability of cyclically dependent write ordering. It also allows fingerprint reclamation to be delayed—e.g., performed offline periodically. Lazy fingerprint reclamation may improve performance since the same data rewritten after a period of non-existence may still be deduplicated. Such scenario has been shown to happen in certain workloads [16].

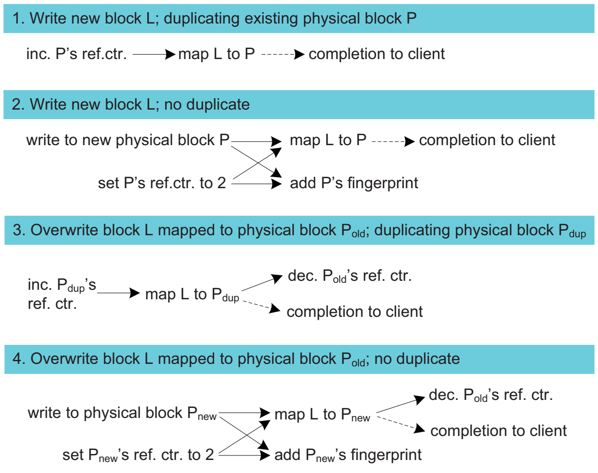
Specifically, we maintain the following ordering between I/O operations during deduplication.

1. The physical data block should always be persisted before being linked with the logical address or the computed fingerprint. A failure recovery may leave some data block inaccessible, but will never lead to any logical address or fingerprint that points to incorrect content.

2. For reference counters, we guarantee that when a sudden failure occurs, the only possibly resulted inconsistency is higher-than-actual reference counters for some physical blocks. A higher-than-actual reference counter may produce garbage (that can be reclaimed asynchronously) while a lower-than-actual reference counter could lead to the serious damage of premature block deletion. To achieve this goal, a new linkage that points to a physical block from some logical address or fingerprint must be preceded by the increment of the physical block's reference counter, and the corresponding unlinking operations must precede the decrement of the physical block's reference counter.

3. Meanwhile, the update of the logical-to-physical block mapping and fingerprints can be processed in parallel since there is no failure-consistent dependency between them.

Figure 1 illustrates our complete soft updates-style write ordering in different write conditions.



1. Write new block L; duplicating existing physical block P
inc. P's ref.ctr.
map L to P
completion to client
2. Write new block L; no duplicate
write to new physical block P
map L to P
completion to client
set P's ref.ctr. to 2
add P's fingerprint
3. Overwrite block L mapped to physical block P_{old}; duplicating physical block P_{dup}
inc. P_{dup}'s ref. ctr.
map L to P_{dup}
dec. P_{old}'s ref. ctr.
completion to client
4. Overwrite block L mapped to physical block P_{old}; no duplicate
write to physical block P_{new}
map L to P_{new}
dec. P_{old}'s ref. ctr.
completion to client
set P_{new}'s ref. ctr. to 2
add P_{new}'s fingerprint

Figure 1: Failure-consistent deduplication write ordering at different write conditions (new write / overwrite, duplicate identified or not, etc.). Solid arrows indicate ordering of writes. The dashed arrow in each case shows when a completion signal is sent to client. Note that a new physical block' reference counter begins at two—one reference from the first logical block and the other from the fingerprint.

A block metadata entry is much smaller than the block itself (e.g., we use the 64-bit address for physical block indexing and the 8-bit reference counter [1]). The I/O cost can be reduced when multiple metadata writes that fall into the same metadata block are merged into one I/O operation. Merging opportunities arise for metadata writes as long as they are not subject to any ordering constraint.

While the metadata write merging presents apparent benefit for I/O reduction, the merging may create additional write ordering constraints and lead to cyclic dependencies or deadlocks. For example, for cases 3 and 4 in Figure 1, if the reference counters for new block P_{dup} / P_{new} and old block P_{old} are on the same metadata block, then the merging of their reference counter updates would cause a deadlock. No metadata write merging is allowed for such situation.

Cyclic dependencies prevent metadata write merging for cost saving and also complicate the implementation. We resolve this issue by delaying the *non-critical* metadata updates involved in the cyclic dependencies. A non-critical update is the write operation that the client completion signal does not depend on, particularly for the same example above, the decrement of P_{old}'s reference counter and the follow-up work in cases 3 and 4 of Figure 1. A delay of those operations until their associated dependencies are cleared simply eliminates the cyclic dependencies. Since the delayed action is not on the critical path of I/O completion, client I/O response will not be affected. Fortunately, under our soft updates-style deduplication metadata management, all the potential deadlocks can be resolved in this way.

2.2 Metadata I/O Merging for Efficiency

As mentioned, substantial I/O cost reduction may result from the merging of multiple metadata writes that fall into the same metadata block. We may enhance the opportunities of metadata I/O merging by delaying metadata update I/O operations so their chances of merging with future metadata writes increase. We explore several opportunities of such I/O delay.

Weak persistence Our deduplication system supports two persistence models with varying performance tradeoffs. Specifically, a *strong persistence* model faithfully preserves the persistence support of the underlying device—an I/O operation is returned by the deduplication layer only after all corresponding physical I/O operations are returned from the device. On the other hand, a *weak persistence* model performs writes asynchronously under which a write is returned early while the corresponding physical I/O operations can be delayed to the next flush or Force Unit Access (FUA)-flagged request.

Under weak persistence, I/O operations can be delayed aggressively to present metadata I/O merging opportunities. Such delays, however, may be hindered by synchronous writes in some applications and databases.

Non-critical I/O delay and merging The example in Section 2.1 shows the removal of cyclic dependencies through the delay of some non-critical metadata updates. In fact, those updates can be free if the delay lasts long enough to merge with future metadata writes that reside on the same metadata block. Moreover, this applies to all the non-critical metadata writes. Specifically, besides the example mentioned in Section 2.1, we also aggressively delay the operations of fingerprint insertion for P / P_{new} in cases 2 and 4 of Figure 1. A sudden system failure may leave some of the reference counters to be higher than actual values, resulting in unreclaimed garbage, or lose some fingerprints for the physical block deduplication chances, but no other serious inconsistency occurs.

Anticipatory I/O delay and merging Two metadata writes to the same physical block will generally result in separate device commits if the interval between their executions is longer than the typical cycle upon which a deduplication system writes to the physical device. If such interval is small, it may be beneficial to impose a short idle time to the physical device (by

[1] A reference counter overflow will lead to the allocation of another new physical block to hold the same block content. Later writes with such content will be mapped to the new block.

stop issuing writes to it) to generate more opportunities of metadata write merging. This is reminiscent of the I/O anticipation scheduling which was proposed as a performance-enhancing seek-reduction technique for mechanical disks [10]. In our case, we temporarily idle the physical device in anticipation of soon-arriving desirable requests for metadata update merging with the existing one.

Anticipatory I/O delay and merging may gain high benefits under a high density of write requests because a short anticipatory device idle period would produce high degree of merging. On the other hand, a light load affords little merging opportunity so that anticipatory device idling only prolongs the I/O response latency. To maximize the benefit of anticipatory I/O delay and merging, we apply a simple heuristic hint as the guidance—the frequency of incoming write requests received by the deduplication system. Intuitively, if the idling period can cover more than one incoming write request, a metadata write merging is likely to happen. We only enable the anticipatory I/O delay and merging under this situation.

3 Implementation

We have implemented our deduplication mechanism at the generic operating system block device layer to enable transparent full-device deduplication for software applications. Specifically, our mechanism is implemented in Linux 3.14.29 kernel as a custom device mapper target. Our implementation follows the basic block read / write interfaces for deduplication checks in the open-source Dmdedup [21] framework.

Duplicates of 4 KB I/O blocks are identified through their hashed fingerprints. We use the widely adopted SHA-1 cryptographic hash algorithm to produce 160-bit (20-byte) block fingerprints. The SHA-1 hashes are collision resistant [14] and we deem two blocks as duplicates if they have matching fingerprints without performing a full block comparison. This is a widely-accepted practice in data deduplication [4, 15, 17, 23] since the chance of hash collision between two different blocks is negligible—less than the error rate of memory and network transfer. A hash table is maintained to organize fingerprints in memory. We partition the fingerprint value space into N segments according to the total number of physical data blocks, and for each fingerprint f, map it to the corresponding segment ($f \bmod N$).

For simplicity, we organize metadata blocks on storage as linear tables. A possible future enhancement is to use a radix-tree structure. The radix tree hierarchical writes could be incorporated into our failure-consistent write ordering without introducing cyclic dependencies.

File systems maintain redundant durable copies of critical information such as the superblock for reliability. For Ext4 file systems, multiple copies of the superblock and block group descriptors are kept across the file system while the main copy resides at the first a few blocks. Deduplicating these blocks could harm such reliability-oriented redundancy measure. We adopt a simple approach to prevent the deduplication of the main copy of the critical file system information (with recognizable block addresses). Specifically, we do not keep their fingerprints in the cache for deduplication; we do not attempt to deduplicate a write to such a block either. A possible future enhancement is to assist such decisions based on hints directly passed from the file system [12].

In our implementation, we delay the non-critical metadata writes for 30 seconds after their failure-consistent dependencies are cleared (during this period they may be merged with other incoming metadata updates residing on the same metadata block). We choose the 1-millisecond idling period for the anticipatory I/O delay and merging which is at the same magnitude as the Flash write latency of our experimental platforms. Our deduplication system is configured with the weak persistence model by default. For better balance between performance and persistence, we periodically commit the delayed metadata writes (besides the synchronous flush or FUA-flagged requests and non-critical metadata writes) to the physical device every 1 second. This is the same setup supported by other device mapper targets in Linux (e.g., `dm-cache`). When data durability is critical, our deduplication system also supports the strong persistence model described in Section 2.2. Our evaluation will cover both models.

4 Evaluation

We evaluate the effectiveness of our proposed deduplication systems on mobile and server workloads. We will compare I/O saving and impact on application performance under several deduplication approaches. We will also assess the deduplication-resulted storage space saving and impact on mobile energy usage.

4.1 Evaluation Setup

Mobile system evaluation setup Our mobile experiments run on an Asus Transformer Book T100 tablet. It contains a 1.33 GHz quad-core Atom (x86) Z3740 processor and 2 GB memory. We deploy the Ubuntu 12.04 Linux distribution with 3.14.29 kernel. The tablet has an internal 64 GB Flash storage with the following random read / write latency (in mSecs)—

	4KB	8KB	16KB	32KB	64KB	128KB
Read	0.27	0.32	0.40	0.63	0.89	1.45
Write	2.91	2.86	4.33	4.96	7.64	11.60

We use the following mobile application workloads—

Advanced Packaging Tool (APT) is a software package installation and maintenance tool on Linux. We study two common package management scenarios via the `apt-get` command: 1) the global package index update (`sudo apt-get update`) and 2) the installation of Firefox (`sudo apt-get install firefox`). We evaluate these workloads under a Ubuntu 12.04 `chroot` environment to facilitate the capture of I/O throughout the root directory. To minimize the noises such as network latencies, we set up the complete Ubuntu 12.04 software repository and pre-download the necessary packages outside the `chroot` jail. The package index update and installation workloads exhibit 23% and 30% write content duplication respectively.

BBench [8] is a smartphone benchmarking tool to assess a web-browser's performance. We run BBench under Firefox 40.0.3 with its provided workloads which include some of the most popular and complex sites on the web. The same setup of Ubuntu 12.04 `chroot` environment (as above) is used along with the BBench web sites workloads located outside the jail. The workload exhibits 73% write duplication.

A field sensor-based *vehicle counting* application that monitors the number and frequency of passing vehicles can help detect the traffic volume, congestion level, and abnormal traffic patterns. Our application leverages the Canny edge detector algorithm [3] from the OpenCV computer vision library. It observes the moving vehicles and records the frames at the time when those vehicles enter the monitored zones. Nearby images in a data stream are often substantially similar, and exhibit block-level redundancy under JPEG/JFIF-style image formats that split an image into multiple sub-regions and encode each separately. We use the pre-collected California highway video streams (at 10 frames per second) from the publicly accessible Caltrans live traffic data [2]. The workload exhibits 27% write duplication.

Server system evaluation setup Our server experiments run on a dual-socket machine where each socket contains an Intel Xeon E5-2620 v3 "Haswell" processor. We deploy the Fedora 20 Linux distribution with 3.14.29 kernel. We perform I/O on a Samsung 850 Pro SSD (256GB) with the following I/O latency (in mSecs)—

	4KB	8KB	16KB	32KB	64KB	128KB
Read	0.12	0.13	0.15	0.18	0.28	0.45
Write	4.70	4.96	5.45	6.13	7.18	7.35

We use the following server / cloud workloads—

Hadoop software library is a framework to use simple programming models for large data sets processing across computers, each offering local computation and storage. We apply the regular expression match of "`dedup[a-z]*`" via Hadoop to all files in the `Documentation` directory of Linux 3.14.29 kernel release. The workload exhibits 55% write duplication on Hadoop's temporary files.

Yahoo Cloud Serving Benchmark (YCSB) [5] is a benchmarking framework for could evaluation. It particularly focuses on the online read / write access-based web serving systems. We perform the YCSB-0.5.0 client under MongoDB-3.2 database. Server load is generated based on the provided `workloada` of YCSB. We tune the parameters of `recordcount` and `operationcount` to 100 and 20,000 respectively, and set `fieldlength` to 8 KB. The workload exhibits 24% write duplication.

4.2 Evaluation on Deduplication Performance

We compare the total volume of Flash I/O writes (including the original application write data and our deduplication metadata, in the 4 KB unit) and application performance (execution latency) between the following system setups—1) the original system that does not support I/O deduplication; 2) I/O shadowing-based Dmdedup [21]; 3) our deduplication system with failure-consistent I/O ordering; our further optimizations of 4) non-critical I/O delay and merging and 5) anticipatory I/O delay and merging. Traces are acquired at the storage device layer to compare the write volumes sent to the storage device under different system setups.

This section evaluates the deduplication performance under a weak persistence model—device writes are performed asynchronously in batches for high efficiency; a write batch is issued at the arrival of a flush or FUA-flagged request, or issued every second at the absence of any flush or FUA-flagged request. We also adapt Dmdedup to follow this pattern. The performance of supporting strong persistence is reported in the next section.

Figure 2 (A) illustrates the results of normalized Flash I/O write volumes under different system conditions. The blue line in the figure indicates the ideal-case deduplication ratio that can only be realized without any deduplication metadata writes. We use the original execution without deduplication as the basis. Dmdedup achieves 7–33% I/O savings for package update / installation, Vehicle counting, Hadoop, and YCSB, but adds 7% I/O writes for BBench. In comparison, our deduplication system with failure-consistent I/O ordering reduces the Flash writes by 17–59% for all the workload cases. The optimization of non-critical I/O delay and merging brings slight benefits (up to 2%) except for the BBench case where 8% additional saving on Flash I/O writes is reached. The optimization of anticipatory I/O delay and merging further increases the I/O saving up to another 6% for all the workloads. Overall, we save 18–63% Flash I/O writes compared to the original non-deduplicated case. These results are very close to the ideal-case deduplication ratios for these workloads.

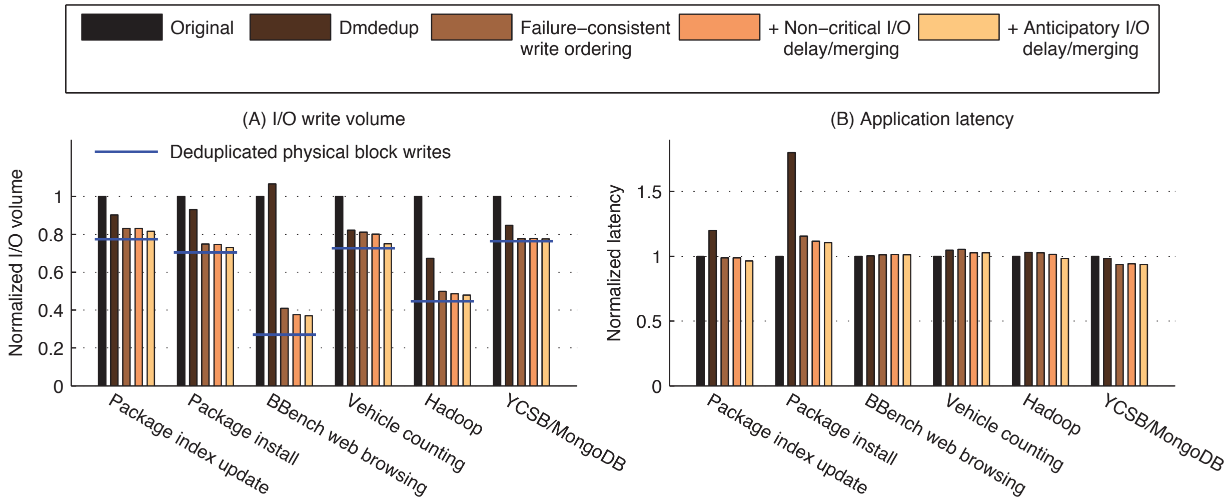


Figure 2: Flash I/O write volume and application performance of different I/O deduplication approaches. The performance in each case is normalized to that under the original (non-deduplicated) execution.

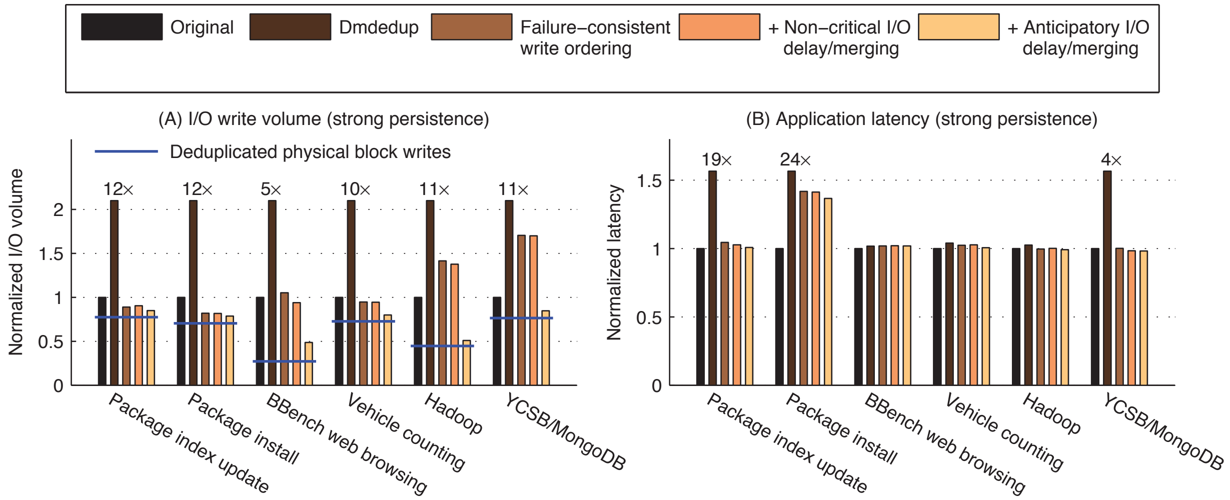


Figure 3: Flash I/O write volume and application performance when supporting strong I/O persistence. The performance in each case is normalized to that under the original (non-deduplicated) execution.

We also evaluate the application performance impact due to deduplication. Figure 2 (B) shows application execution latencies under the same system setups as above. We use the original execution without deduplication as the basis. While Dmdedup has small impact (less than 5% of performance overhead) to the workloads of BBench, Vehicle counting, Hadoop, and YCSB, it increases the costs to 1.2× and 1.8× for package update and installation respectively. In comparison, our deduplication system, either with or without optimizations, only imposes moderate overhead to the package installation case (around 11–15%). The impacts to other workloads' performance are small (less than 5%) and in some cases we actually achieve slight performance improvement (around 1–6%).

4.3 Evaluation under Strong Persistence

We evaluate the performance of our deduplication system between the similar system setups as Section 4.2, but with the support of strong persistence model—an I/O operation is returned by the deduplication layer only after all corresponding physical I/O operations are returned from the device. Dmdedup is configured accordingly with the single-write transaction size—the corresponding data / metadata updates are committed after every write operation [21].

Figure 3 (A) illustrates the results of normalized Flash I/O write volumes under different system conditions. We use the original execution without deduplication as the basis. Dmdedup adds large overhead for all the workloads from 5× to 12×. In comparison, our deduplication system with failure-consistent I/O ordering reduces the Flash writes by 5–18% for package update / installation and vehicle counting. Meanwhile, it adds 5% I/O volumes for BBench while such overhead becomes large for Hadoop and YCSB (around 41–71%). The optimization of non-critical I/O delay and merging brings slight benefits (up to 3%) except for the BBench case where 11% additional saving on Flash I/O writes is reached. The optimization of anticipatory I/O delay and merging exhibits significant benefit for all the workloads under the strong persistence model. Specifically, enabling it brings up to 63% additional saving on Flash I/O writes. Overall, we save 15–51% Flash I/O writes compared to the original non-deduplicated case.

Figure 3 (B) shows application execution latencies under the same system setups as above. We use the original execution without deduplication as the basis. Dmdedup adds large overhead for package update / installation and YCSB from 4× to 24× while the performance impact is small for other workloads (less than 4%). In comparison, our deduplication system, either with or without optimizations, only imposes large overhead to the package installation case (around 37–42%). The impacts to other workloads' performance are small (less than 4%).

4.4 Evaluation on Storage Space Saving

We compare the space usage between the non-deduplicated system and our deduplication system. Under the non-deduplicated execution, we directly calculate the occupied blocks during the workload running. For our deduplication system, the space usage is obtained by putting together the following items— 1) the space for physical blocks written along with the corresponding physical block metadata (reference counters and fingerprints); 2) the space for logical block metadata (logical-to-physical block mapping) for the occupied logical blocks. The table below shows that the workload executions exhibit strong deduplication space saving—

Workload	Space usage		Saving
	Original	Dedup	
Update	168.8 MB	131.5 MB	22%
Install	137.9 MB	98.9 MB	28%
BBench	11.0 MB	4.8 MB	56%
Vehicle	12.8 MB	9.4 MB	27%
Hadoop	80.0 MB	39.8 MB	50%
YCSB	800.6 MB	618.5 MB	23%

4.5 Evaluation on Mobile Energy Usage

We assess the energy impact of our deduplication system on mobile platforms. The energy usage of a workload is the product of its power consumption and runtime. The runtime is normally the application latency, except in the case of our vehicle counting workload where the application operates at a fixed frame-per-second rate and therefore its runtime is not affected by the frame processing latency. We compare the power usage, runtime difference, and energy usage between the original (non-deduplicated) system and our deduplication system—

Workload	Power (Watts)		Runtime	Energy
	Orig.	Dedup	impact	impact
Update	6.35	6.41	-3.6%	-3%
Install	6.03	6.06	+10.5%	+11%
BBench	6.10	6.15	+1.1%	+2%
Vehicle	6.70	6.70	0.0%	0%

Results show that our deduplication mechanism adds 11% energy usage for package installation, mostly due to the increase of runtime. The energy impact is no more than 2% in the other three workloads. The energy usage even decreases by 3% for package index update primarily due to a reduction in runtime.

5 Conclusion

This paper presents a new I/O mechanism, called OrderMergeDedup, that deduplicates writes to the primary Flash storage with failure-consistency and high efficiency. We devise a soft updates-style metadata write ordering that maintains data / metadata consistency over failures (without consistency-induced additional I/O) on the storage. We further use anticipatory I/O delay and merging to reduce the metadata I/O writes. We have made a prototype implementation at the Linux device mapper layer and experimented with a range of mobile and server workloads.

Results show that OrderMergeDedup is highly effective—realizing 18–63% write reduction on workloads that exhibit 23–73% write content duplication. We also save up to 56% in space usage. The anticipatory I/O delay optimization is particularly effective to increase metadata merging opportunities when supporting the strong I/O persistence model. OrderMergeDedup has a slight impact on the application latency and mobile energy. It may even improve the application performance due to reduced I/O load.

Acknowledgments This work was supported in part by the National Science Foundation grants CNS-1217372, CNS-1239423, and CCF-1255729, and by a Google Research Award. We also thank the anonymous FAST reviewers, our shepherd Hakim Weatherspoon, and Vasily Tarasov for comments that helped improve this paper.

References

[1] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, et al. System R: Relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[2] Live traffic cameras, california department of transportation. `video.dot.ca.gov`.

[3] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (6):679–698, 1986.

[4] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of Flash memory based solid state drives. In *the 9th USENIX Conf. on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2011.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *the First ACM Symp. on Cloud Computing (SOCC)*, pages 143–154, Indianapolis, IN, June 2010.

[6] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. on Computer Systems*, 18(2):127–153, May 2000.

[7] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *the 9th USENIX Conf. on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2011.

[8] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-system analysis and characterization of interactive smartphone applications. In *the IEEE Intl. Symp. on Workload Characterization (IISWC)*, pages 81–90, Austin, TX, Nov. 2011.

[9] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Technical Conf.*, San Francisco, CA, Jan. 1994.

[10] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *the 8th ACM Symp. on Operating Systems Principles (SOSP)*, pages 117–130, Banff, Alberta, Canada, Oct. 2001.

[11] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *the 10th USENIX Conf. on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2012.

[12] S. Mandal, G. Kuenning, D. Ok, V. Shastry, P. Shilane, S. Zhen, V. Tarasov, and E. Zadok. Using hints to improve inline block-layer deduplication. In *the 14th USENIX Conf. on File and Storage Technologies (FAST)*, Santa Clara, CA, Feb. 2016.

[13] D. Meister and A. Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *IEEE 26th Symp. on Mass Storage Systems and Technologies (MSST)*, pages 1–6, May 2010.

[14] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[15] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *the 8th ACM Symp. on Operating Systems Principles (SOSP)*, pages 174–187, Banff, Alberta, Canada, Oct. 2001.

[16] P. Nath, M. A. Kozuch, D. R. O'Hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *USENIX Annual Technical Conf.*, pages 71–84, Boston, MA, June 2006.

[17] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *the First USENIX Conf. on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

[18] M. I. Seltzer, G. R. Granger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus soft updates: Asynchronous metadata protection in file systems. In *USENIX Annual Technical Conf.*, San Deigo, CA, June 2000.

[19] K. Shen, S. Park, and M. Zhu. Journaling of journal is (almost) free. In *the 12th USENIX Conf. on File and Storage Technologies (FAST)*, pages 287–293, Santa Clara, CA, Feb. 2014.

[20] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *the 10th USENIX Conf. on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2012.

[21] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok.

Dmdedup: Device mapper target for data deduplication. In *Ottawa Linux Symp.*, Ottawa, Canada, July 2014.

[22] P. F. Williams. Street smarts: How intelligent transportation systems save money, lives and the environment. Technical report, ACS Transportation Solutions Group, Xerox, Feb. 2009.

[23] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *the 6th USENIX Conf. on File and Storage Technologies (FAST)*, pages 269–282, San Jose, CA, Feb. 2008.

CacheDedup: In-line Deduplication for Flash Caching

Wenji Li
Arizona State University

Gregory Jean-Baptise
Florida International University

Juan Riveros
Florida International University

Giri Narasimhan
Florida International University

Tong Zhang
Rensselaer Polytechnic Institute

Ming Zhao
Arizona State University

Abstract

Flash caching has emerged as a promising solution to the scalability problems of storage systems by using fast flash memory devices as the cache for slower primary storage. But its adoption faces serious obstacles due to the limited capacity and endurance of flash devices. This paper presents CacheDedup, a solution that addresses these limitations using in-line deduplication. First, it proposes a novel architecture that integrates the caching of data and deduplication metadata (source addresses and fingerprints of the data) and efficiently manages these two components. Second, it proposes duplication-aware cache replacement algorithms (D-LRU, D-ARC) to optimize both cache performance and endurance. The paper presents a rigorous analysis of the algorithms to prove that they do not waste valuable cache space and that they are efficient in time and space usage. The paper also includes an experimental evaluation using real-world traces, which confirms that CacheDedup substantially improves I/O performance (up to 20% reduction in miss ratio and 51% in latency) and flash endurance (up to 89% reduction in writes sent to the cache device) compared to traditional cache management. It also shows that the proposed architecture and algorithms can be extended to support the combination of compression and deduplication for flash caching and improve its performance and endurance.

1 Introduction

Flash caching employs flash-memory-based storage as a caching layer between the DRAM-based main memory and HDD-based primary storage in a typical I/O stack of a storage system to exploit the locality inherent in the I/Os at this layer and improve the performance of applications. It has received much attention in recent years [7, 5, 2, 23], which can be attributed to two important reasons. First, as the level of consolidation—in terms of both the number of workloads consolidated to a single host and the number of hosts consolidated to a single storage system—continues to grow in typical computing systems such as data centers and clouds, the scalability of the storage system becomes a serious issue. Second, the high performance of flash-memory-based storage devices has made flash caching a promising option to address this scalability issue: it can reduce the load on the primary storage and improve workload performance by servicing I/Os using cached data.

There are however several key limitations to effective caching with flash memories. First, with the increasing data intensity of modern workloads and the number of consolidated workloads in the system, the demands on cache capacity have skyrocketed compared to the limited capacity of commodity flash devices. Second, since flash memories wear out with writes, the use of flash for caching aggravates the endurance issue, because both the writes inherent in the workload and the reads that miss the cache induce wear-out.

This paper presents CacheDedup, an in-line flash cache deduplication solution to address the aforementioned obstacles. First, deduplication reduces the cache footprint of workloads, thereby allowing the cache to better store their working sets and reduce capacity misses. Second, deduplication reduces the number of necessary cache insertions caused by compulsory misses and capacity misses, thereby reducing flash memory wear-out and enhancing cache durability. Although deduplication has been studied for a variety of storage systems including flash-based primary storage, this paper addresses the unique challenges in integrating deduplication with caching in an efficient and holistic manner.

Efficient cache deduplication requires seamless integration of caching and deduplication management. To address this need, CacheDedup embodies a novel architecture that integrates the caching of data and deduplication metadata—the source addresses and fingerprints of the data, using a separate Data Cache and Metadata Cache. This design solves two key issues. First, it allows CacheDedup to bound the space usage of metadata, making it flexible enough to be deployed either on the client side or the server side of a storage system, and implemented either in software or in hardware. Second, it enables the optimization of caching historical source addresses and fingerprints in the Metadata Cache after their data is evicted from the Data Cache. These historical deduplication metadata allow CacheDedup to quickly recognize duplication using the cached fingerprints and produce cache hits when these source addresses are referenced again.

Based on this architecture, we further study duplication-aware cache replacement algorithms that can exploit deduplication to improve flash cache performance and endurance. First, we present D-LRU, a duplication-aware version of LRU which can be efficiently implemented by enforcing an LRU policy on both the Data and Metadata Caches. Second, we present D-ARC, a duplication-aware version of ARC that exploits the scan-resistant ability of ARC to further improve cache performance and endurance. For both algorithms, we also prove theoretically that they do not lead to wastage in the Data and Metadata caches and can efficiently use their space.

CacheDedup is implemented atop block device virtualization [5], and can be transparently deployed on existing storage systems. We evaluate it using real-world traces, including the FIU traces [18] and our own traces collected from a set of Hadoop VMs. The results show that CacheDedup substantially outperforms traditional cache replacement algorithms (LRU and ARC) by reducing the cache miss ratio by up to 20%, I/O latency by 51%, and the writes sent to flash memories by 89%. It can effectively deduplicate data both within a workload and across multiple workloads that share the cache. We also measure the overhead of CacheDedup using the fio benchmark [1], which shows that the throughput overhead is negligible and the latency overhead from fingerprinting can be overlapped with concurrent I/O operations and dominated by the hit ratio gain from deduplication. In terms of space overhead, CacheDedup needs $< 4\%$ of the flash cache to store the deduplication metadata in order for our algorithms to achieve peak performance.

CacheDedup is among the first to study duplication-aware cache management for cache deduplication. Compared to the related work, which also considered data reduction techniques for server-side flash caching [19], we show that our approach can be naturally extended to support both duplication- and compression-aware cache management and that it can improve the read hit ratio by 12.56%. Our approach is not specific to flash-based caches—it leverages only flash devices' faster speed compared to HDDs and larger capacity compared to DRAMs. Therefore, it is also applicable to other non-volatile memory technologies used as a caching layer between DRAMs and the slower secondary storage. While the new technologies may have better endurance, they are likely to have quite limited capacity compared to NAND flash, and will still benefit greatly from CacheDedup, which can substantially reduce both cache footprint and writes sent to cache device.

The rest of the paper is organized as follow: Section 2 explains the background; Section 3 presents the architectural design of CacheDedup; Section 4 describes the duplication-aware cache management algorithms; Section 5 presents the evaluation results; Section 6 examines the related work; and Section 7 concludes the paper.

2 Background and Motivations

Need of Integrated Flash Cache Deduplication. The emergence of flash-memory-based storage has greatly catalyzed the adoption of flash caching at both the client side and server side of a network storage system [7, 5, 2]. However, flash caches still face serious capacity and endurance limitations. Given the increasingly data-intensive workloads and increasing level of storage consolidation, the size of commodity flash devices is quite limited. The use of flash for caching also aggravates the wear-out problem of flash devices, because not only the writes from the workloads cause wear-out, but also all the reads that are inserted into the cache due to cache misses.

Deduplication is a technique for eliminating duplicate copies of data and has been used to reduce the data footprint for primary storage [12, 15] and backup and archival storage [27, 22]. It often uses a collision-resistant cryptographic hash function [4, 28] to identify the content of a data block and discover duplicate ones [27, 18, 8]. Deduplication has the potential to solve the above challenges faced by flash caching. By reducing the data footprint, it allows the flash cache to more effectively capture the locality of I/O workloads and improve the performance. By eliminating the caching of duplicate data, it also reduces the number of writes to the flash device and the corresponding wear-out.

Although one can take existing flash caching and deduplication solutions and stack them together to realize cache deduplication, the lack of integration will lead to inefficiencies in both layers. On one hand, it is infeasible to stack a caching layer upon a deduplication layer because the former would not be able to exploit the space reduction achieved by the latter. On the other hand, there are also serious limitations to simply stacking a deduplication layer upon a caching layer. First, the deduplication layer has to manage the fingerprints for the entire primary storage, and may make fingerprint-management decisions that are detrimental to the cache, e.g., evicting fingerprints belonging to data with good locality and causing duplicate copies in the cache. Second, the caching layer cannot exploit knowledge of data duplication to improve cache management. In contrast, CacheDedup employs an integrated design to optimize the use of deduplication for flash caching for both performance and endurance.

The recent work Nitro [19] studied the use of both deduplication and compression for server-side flash caches. CacheDedup is complementary to Nitro in its new architecture and algorithms for duplication-aware cache management. Moreover, our approach can be applied to make the cache management aware of both compression and deduplication and improve such a solution that uses both techniques. A quantitative comparison to Nitro is presented in Section 5.6.

Deduplication can be performed in-line—in the I/O path—or offline. CacheDedup does in-line deduplication to prevent any duplicate block from entering the cache, thereby achieving the greatest reduction of data footprint and wear-out. Our results show that the overhead introduced by CacheDedup is small. Deduplication can be done at the granularity of fixed-size chunks or content-defined variable-size chunks, where the latter can achieve greater space reduction but at a higher cost. CacheDedup chooses deduplication at the granularity of cache blocks, which fits the structure of flash caches and facilitates the design of duplication-aware cache replacement. Our results also confirm that a good level of data reduction

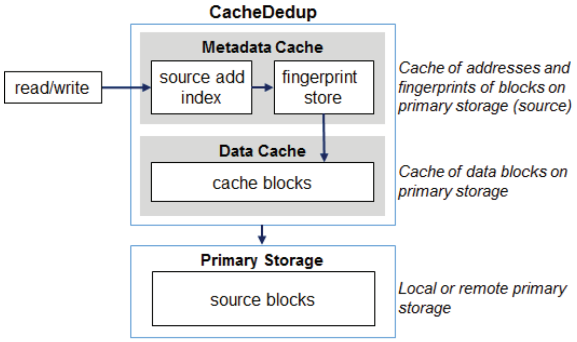

CacheDedup
Metadata Cache
read/write
source add index
fingerprint store
Cache of addresses and fingerprints of blocks on primary storage (source)
Data Cache
cache blocks
Cache of data blocks on primary storage
Primary Storage
source blocks
Local or remote primary storage

Figure 1: Architecture of CacheDedup

can be achieved with fixed-size cache deduplication.

Need for Deduplication-aware Cache Management. There exists a number of cache replacement schemes. In particular, the widely used LRU algorithm is designed to exploit temporal locality by always evicting the least-recently used entry in the cache. Theoretically, it has been shown to have the best guarantees in terms of its worst-case performance [30]. But it is not “scan resistant”, i.e., items accessed only once could occupy the cache and reduce the space available for items that are accessed repeatedly. ARC [21] is an adaptive algorithm that considers both recency and frequency in cache replacement. It is “scan resistant” and shown to offer better performance for many real-world workloads.

However, cache replacement algorithms typically focus on maximizing the hit ratio, and disregard any issues related to the lifespan and wear-and-tear of the hardware device, which are unfortunately crucial to flash-based caches. Attempts to reduce writes by bypassing the cache invariably affect the hit ratio adversely. The challenge is to find the “sweet spot” between keeping hit ratios close to the optimum and lowering the number of write operations to the device. CacheDedup addresses this challenge with its duplication-aware cache replacement algorithms, designed by optimizing LRU and ARC and enabled by an integrated deduplication and cache management architecture.

3 Architecture

3.1 Integrated Caching and Deduplication

CacheDedup seamlessly integrates the management of cache and deduplication and redesigns the key data structures required for these two functionalities. A traditional cache needs to manage the mappings from the *source addresses* of blocks on the primary storage to the *cache addresses* of blocks on the cache device. A deduplication layer needs to track the fingerprints of the data blocks in order to identify duplicate blocks. There are several unique problems caused by the integration of caching and deduplication to the design of these data structures.

First, unlike a traditional cache, the number of source-to-cache address mappings in CacheDedup is not bounded by the size of the cache, because with deduplication, there is now a many-to-one relationship between these two address spaces. Second, even though the number of fingerprints that a cache has to track is bounded by the cache size, there is an important reason for CacheDedup to track fingerprints for data beyond what is currently stored in the cache. Specifically, it is beneficial to keep historical fingerprints for the blocks that have already been evicted from the cache, so that when these blocks are requested again, CacheDedup does not have to fetch them from the primary storage in order to determine whether they are duplicates of the currently cached blocks. Such an optimization is especially important when CacheDedup is employed as a client-side cache because it can reduce costly network accesses. However, fingerprint storage still needs to abide by the limit on CacheDedup’s space usage.

To address these issues, we propose a new *Metadata Cache* data structure to cache source addresses and their fingerprints. This design allows us to solve the management of these metadata as a cache replacement problem and consider it separately from the *Data Cache* that stores data blocks (Figure 1). The Metadata Cache contains two key data structures. The *source address index* maps a source addess of the primary storage to a fingerprint in the Metadata Cache. Every cached source address is associated with a cached fingerprint, and because of deduplication, multiple source addresses may be mapped to the same fingerprint. The *fingerprint store* maps fingerprints to block addresses in the Data Cache. It also contains historical fingerprints whose corresponding blocks are not currently stored in the Data Cache. When the data block pointed to by a historical fingerprint is brought back to the Data Cache, all the source addresses mapped to this fingerprint can generate cache hits when they are referenced again. Each fingerprint has a reference count to indicate the number of source blocks that contain the same data. When it drops to zero, the fingerprint is removed from the Metadata Cache.

The decoupled Metadata Cache and Data Cache provide separate control “knobs” for our duplication-aware algorithms to optimize the cache management (Section 4). The algorithms limit the metadata space usage by applying their replacement policies to the Metadata Cache and exploiting the cached historical fingerprints to opportunistically improve read performance. Both caches can be persistently stored on a flash device to tolerate failures as discussed in Section 3.3. Moreover, our architecture enables the flash space to be flexibly partitioned between the two caches. For a given Data Cache size, the minimum Metadata Cache size is the required space for storing the metadata of all the cached data. Our evaluation using real-world traces in Section 5 shows that the “minimum” size is small enough to not be an issue in practice. More importantly, the Metadata Cache can be expanded by taking away some space from the Data Cache to store more historical fingerprints and potentially improve the performance. This tradeoff is studied in Section 5.4.

3.2 Operations

Based on the architecture discussed above, the general operations of CacheDedup are as follows. The necessary cache replacement is governed by the duplication-aware algorithms presented in Section 4.

Read. A read that finds its source block address in the Metadata Cache with a fingerprint pointing to a Data Cache block is a hit in the Data Cache. Otherwise, it is a miss. Note that the read may match a historical fingerprint in the Metadata Cache which does not map to any data block in the Data Cache, and it is still a miss. Upon a miss, the requested data block is fetched from the primary storage and, if it is not a duplicate of any existing cached data block, it is inserted into the Data Cache. The corresponding source address and fingerprint are inserted into the Metadata Cache if necessary. If the fingerprint already exists in the Metadata Cache, it is then "revived" by pointing to the new Data Cache block, and all the historical source addresses that point to this fingerprint are also "revived" because they can generate hits when accessed again.

Write. The steps differ by the various write policies that CacheDedup supports:

1. *Write invalidate*—the requested source address and the cached data for this address are invalidated in the Metadata and Data Caches if they exist. The write goes directly to the primary storage.
2. *Write through*—the write is stored in cache and at the same time submitted to the primary storage. The source address and fingerprint are inserted into the Metadata Cache if they are not already there. If the data contained in the write matches an existing cached block, no change needs to be made to the Data Cache, while the reference count of its fingerprint is incremented by one. If the previous data of this block is already in the Data Cache, the reference count of the previous fingerprint is decremented by one.
3. *Write back*—the write is stored only in the cache and submitted to the primary storage later, when the block is evicted or when the total amount of dirty data exceeds a predefined threshold. The steps are identical to the write-through policy except that the write is not immediately submitted to the primary storage.

3.3 Fault Tolerance

The nonvolatile nature of flash storage allows the cached data to persist when the host of the cache crashes, but to recover the cached data, their metadata also needs to be stored persistently. If the goal is to avoid data loss, only the source-to-cache address mappings for the locally modified data from using the write-back policy must be persistent. CacheDedup synchronously commits both the metadata and data to the cache device to ensure consistency. If the goal is to avoid warmup after the host restarts, the entire Metadata Cache, including the source block index and fingerprints, is made persistent. The time overhead of making the Metadata Cache persistent is not as significant as its space overhead on the flash device, because the metadata for clean blocks can be written in batches and asynchronously. For both fault tolerance goals, the Metadata Cache is also kept in main memory to speed up cache operations, and its memory usage is bounded. Finally, if the goal is to tolerate flash device failures, additional mechanisms [26] need to be employed. We can also leverage related work [17] to provide better consistency for flash caching.

3.4 Deployment

CacheDedup can be deployed at both the client side and server side of a storage system: client-side CacheDedup can more directly improve application performance by hiding the high network I/O latency, whereas server-side CacheDedup can use the I/Os from multiple clients to achieve a higher level of data reduction. When CacheDedup is used by multiple clients that share data, a *cache coherence* protocol is required to ensure that each client has a consistent view of the shared data. Although it is not the focus of this paper, CacheDedup can straightforwardly extend well-studied cache coherence protocols [13, 24] to synchronize both the data in the Data Cache and the fingerprints in the Metadata Cache, thereby ensuring consistency across the clients.

While the discussions in this paper focus on a software-based implementation of CacheDedup, its design allows it to be incorporated into the flash translation layer of specialized flash devices [29, 25]. The space requirement is bounded by the Metadata Cache size, and the computational complexity is also limited (Section 4), making CacheDedup affordable for modern flash hardware.

The discussions in the paper also assume the deployment of CacheDedup at the block-I/O level, but its design is largely applicable to the caching of filesystem-level reads and writes, which requires only changing the Metadata Cache to track (*file handle*, *offset*) tuples instead of source block addresses.

4 Algorithms

In this section we present two duplication-aware cache replacement algorithms. Both are enabled by the integrated cache and deduplication management framework described above. We first define some symbols (Table 1).

- *Data Cache*, D, stores the contents of up to d deduplicated data blocks, indexed by their fingerprints.
- *Metadata Cache*, M, holds a set of up to m source addresses and corresponding fingerprints, a function f that maps a source address to the fingerprint of its data block, and a function h that maps a fingerprint to the location of the corresponding data block in D. We denote the composition of f and h by the function h' that maps a source address to the location of the corresponding data block.

Symbol	Definition
D	Data Cache
d	Total number of data blocks in the Data Cache
M	MetaData Cache
m	Total number of source addresses in M
p_i	A source address in M
g_i	A fingerprint in fingerprint store
a_i	The content of a data block in D
$f(p_i)$	Function that maps a source address to a fingerprint
$h(g_i)$	Function that maps a fingerprint to a data block
$h'(p_i)$	Function that maps a source address to a data block

Table 1: Variable definitions

Thus, for a source address x in M, $f(x)$ is its fingerprint, and $h'(x) = h(f(x))$ is the corresponding data block in D. If $f(x) = f(y)$ and $x \neq y$, then x and y contain *duplicate* data. If $f(x) = g$, we will refer to x as an *inverse map* or one of the addresses of fingerprint g. If we have a source address in M for which the corresponding data is absent from D, we call it an *orphaned address*; if we have a block in D for which the mapping information is not available in M, then it is an *orphaned data block*. Instead of strictly disallowing orphaned addresses and data, we will require our algorithms to comply with the **no-wastage policy**, which states that the cache replacement algorithms are required to not have orphaned addresses and orphaned data blocks simultaneously. The no-wastage policy is important because "wastage" implies suboptimal use of the cache, i.e., instead of bringing in other useful items into cache, we are storing items in cache with incomplete information that would surely result in a miss if requested.

In the rest of this section, we describe the algorithms and analyze their no-wastage property and complexity. Note that the size of data structures required by the algorithms is bounded by the space required to store up to m source addresses and fingerprints; therefore, we omit the space complexity analysis.

4.1 D-LRU

4.1.1 Algorithm

We present D-LRU (pronounced "dollar-you"), a duplication-aware variant of LRU. The pseudocode (Algorithm 1) consists of two separate LRU policies being enforced first on the Metadata Cache (M) and then on the Data Cache (D). INSERT-USING-LRU(x,A,n) inserts x in list A (with capacity n) only if it is not already in A, in which case the LRU item is evicted to make room for it.

4.1.2 Analysis

The algorithm of D-LRU is rather simple, but our analysis shows that it is also quite powerful as it allows efficient use of both the Metadata and Data Caches with no wastage. We start the analysis with several useful observations. The first is that no duplicate addresses are inserted into M and no duplicate data blocks are inserted into D. However, every new address does result in an entry in M, even if it corresponds to a duplicate data block.

Algorithm 1: D-LRU pseudocode

```
REMARKS: $D$ is indexed by $f(x)$ and $M$ is indexed by $x$
INPUT: The request stream $x_1, x_2, \dots, x_t, \dots$
INITIALIZATION: Set $D = \emptyset$, $M = \emptyset$
for every $t \geq 1$ and any $x_t$ do
    INSERT-USING-LRU$(x_t, M, m)$
    INSERT-USING-LRU$(f(x_t), D, d)$
```

To discuss more observations, we introduce the following notation. Let $\{p_1, \dots, p_m\}$ be the source addresses in the Metadata Cache M, ordered so that p_1 is the LRU entry and p_m the MRU entry. Let $\{g_1, \dots, g_n\}$ be the corresponding fingerprints stored in the fingerprint store. Let $\{a_1, a_2, \dots, a_d\}$ be the Data Cache contents, ordered so that a_1 is the LRU entry and a_d the MRU entry. For any data block a, let $max_M(a)$ be the position in the Metadata Cache of its most recently accessed source address. In other words, for any $a \in D$, $max_M(a) = \max\{i | h(f(p_i)) = a \ \wedge p_i \in M\}$, if a is not an orphan, and 0 otherwise. Next we observe that the order in D is the same as the order of their most recently accessed addresses. Finally, any orphans in D must occupy contiguous positions at the bottom of the DataCache LRU list. Any orphans in M need not be in contiguous positions but must occupy positions that are lower than $max_M(a_1)$, where a_1 is the LRU item in D.

To prove that D-LRU does indeed comply with the no-wastage policy, we propose the following invariants.

P1: If $\exists a \in D$ s.t. $max_M(a) = 0$, then $\forall q \in M, h'(q) \in D$.
P2: If $\exists q \in M$ s.t. $h'(q) \notin D$, then $\forall a \in D, max_M(a) > 0$.

Simply put, invariant P1 states that if there are orphaned data items in D, there are no orphaned addresses in M. Invariant P2 states the converse. When p is the only entry in M, then $p \in M$ and $h'(p) \in D$. The invariants hold. We then need to show that if these two invariants hold after serving a set of requests (inductive hypothesis), then it continues to hold after processing one more request. Let the next request be for source address x and fingerprint $f(x)$. We list the base case and then one of three cases must occur before the new request is processed.

CASE 1: $x \in M$ and $h(f(x)) \in D$. D-LRU performs no evictions and the contents of the Metadata and Data Caches remain unchanged, as do the invariants.

CASE 2: $x \in M$ and $h(f(x)) \notin D$. In this case D-LRU evicts an item from D to bring back the data $h(f(x))$ for the orphaned address x. Using the inductive hypothesis, and the fact that $x \in M$ is orphaned, we know that no orphaned items exist in

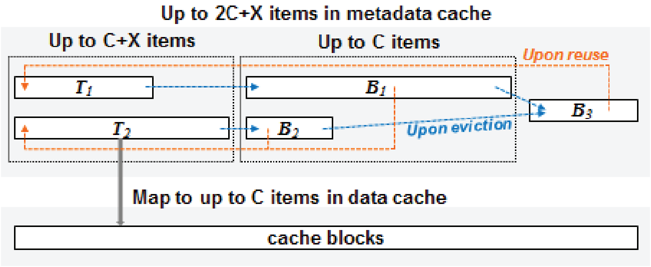


Figure 2: Architecture of D-ARC

D. Since D-LRU does not evict any entry from M, no new orphans will be created in D, leaving the invariants true.

CASE 3: $x \notin M$. In this case, D-LRU evicts p_1 and adds x as the new MRU in M. Also, if $h(f(x)) \notin D$, then D-LRU will evict a_1 and add $h(f(x))$ as MRU in D. Two possible cases apply for the analysis of this situation.

CASE 3.1: If there is at least one orphaned data item in D (and none in M) prior to processing the new request, then since all orphans in D are at the bottom, its LRU item a_1 is an orphan. Thus, the eviction of a_1 cannot create any orphans in M, and thus the invariants will hold.

CASE 3.2: If there are no orphans in D and $x \notin M$, we have two cases. First, if $h(f(x)) \notin D$, the algorithm must evict the LRU items from both M and D. If $max_M(a_1) > 1$ then the eviction of p_1 will leave no orphans. If $max_M(a_1) = 1$ then p_1 is the only address in M that maps to a_1 and since both will get evicted, no new orphans are created in the process. Second, if $h(f(x)) \in D$, the algorithm only evicts p_1 from M. If $max_M(a_1) > 1$ then the eviction of p_1 will not leave orphans in D. If $max_M(a_1) = 1$ then the eviction of p_1 makes a_1 an orphan. But because there can be no orphan addresses occupying positions lower than p_1 in M, the invariants still hold.

Thus, D-LRU complies with the *no-wastage policy*.

Complexity. D-LRU can service every request in $O(1)$ time because the LRU queues are implemented as doubly linked lists and the elements in the lists are also indexed to be able to access in constant time. Evicting an element or moving it to the MRU position can also be done in constant time.

4.2 D-ARC

4.2.1 Algorithm

Next we present a duplication-aware cache management algorithm, D-ARC, that is based on ARC [21], which is a major advance over LRU because of its scan-resistant nature. Similarly, the algorithm of D-ARC is more complex than D-LRU.

We start with a brief description of ARC. ARC assumes a Data Cache of size C. It uses four LRU lists, T_1, T_2, B_1, and B_2 to store metadata, i.e., cached source addresses, with total size $2C$. The key idea is to preserve information on frequently accessed data in T_2 and to let information on "scan" data (single-access data) pass through T_1. Together, the size of T_1 and T_2 cannot exceed C. When a new data block is added to the cache, its corresponding metadata is added to T_1, and it is moved to T_2 only when that address is referenced again. The relative sizes of the two lists, T_1 and T_2, are controlled by an adaptive parameter p. The algorithm strives to maintain the size of T_1 at p. When an item is evicted from T_1 or T_2, its data is also removed from the cache. ARC uses B_1 and B_2 to save metadata evicted from T_1 and T_2, respectively. Together they store an additional C metadata items, which help monitor the workload characteristics. When a source address from B_1 or B_2 is referenced, it is brought back into T_2, but triggers an adjustment of the adaptive parameter p.

Our ARC-inspired duplication-aware cache replacement algorithm is named D-ARC. The idea behind it is to maintain a duplication-free Data Cache D of maximum size C, and to use an ARC-based replacement scheme in the Metadata Cache M. If evictions are needed in D, only data blocks with no mappings in $T_1 \cup T_2$ are chosen for eviction. Figure 2 illustrates the architecture of D-ARC. The corresponding fingerprints are stored in the fingerprint store, which is omitted for clarity.

The first major difference from ARC is that the total size of T_1 and T_2 is not fixed and will vary depending on the duplication in the workload. If the workload has no duplicate blocks, then M will hold at most C source addresses, each mapped to a unique block in D, just as with the original ARC. In the presence of duplicates, D-ARC allows the total size of T_1 and T_2 to grow up to $C+X$, in order to store X more source addresses whose data duplicates the existing ones. A single block in D may be mapped from multiple source addresses in T_1 and T_2. X is a parameter that can be tuned by a system administrator to bound the size of M to store up to $2C+X$ items (source address/fingerprint pairs).

Second, when source addresses are evicted from T_1 or T_2 and moved into B_1 or B_2, as dictated by the ARC algorithm, D-ARC saves their fingerprints and data to opportunistically improve the performance of future references to these addresses. Moreover, D-ARC employs an additional LRU list B_3 to save source addresses (and their fingerprints) evicted from B_1 and B_2, as long as the lists T_1, T_2, and B_3 together store less than $C+X$ mappings. In essence, B_3 makes use of the space left available by T_1 and T_2. When a hit occurs in B_3, it is inserted into the MRU position of T_1, but does not affect the value of p. A future request to a source address retained in $B_1 \cup B_2 \cup B_3$ may result in a hit in D-ARC if its fingerprint shows that the data is in D. In contrast, any item found in B_1 or B_2 always results in a miss in the original ARC.

Third, when eviction is necessary in the Data Cache, D-ARC chooses an item with no mappings in $T_1 \cup T_2$. If no such data item is available, then items are evicted from $T_1 \cup T_2$ using the original ARC algorithm until such a data block is found. Note that at most $X+1$ items are evicted from $T_1 \cup T_2$ in the process.

The D-ARC pseudocode is shown in Algorithms 2 and

Algorithm 2: D-ARC(C, X) pseudocode

```
INPUT: The request stream $x_1, x_2, \ldots, x_t, \ldots$
PROCESSREQUEST ()
    if $x_i \in T_1 \cup T_2$ then
        Move $x_i$ to MRU position on $T_2$
    if $x_i \in B_1 \cup B_2$ then
        Increase $p$, if $x_i \in B_1$; else Decrease $p$.
        if $h'(x_i) \notin D$ then
            InsertInDataCache()
        CheckMetadataCache()
        Move $x_i$ to MRU position in $T_2$
    if $x_i \in B_3$ then
        if $h'(x_i) \notin D$ then
            InsertInDataCache()
        Move $x_i$ to MRU position in $T_1$
    if $x_i \notin T_1 \cup T_2 \cup B_1 \cup B_2 \cup B_3$ then
        if $h'(x_i) \notin D$ then
            InsertInDataCache()
        CheckMetadataCache()
        Move $x_i$ to MRU position in $T_1$
```

3. In the main program (ProcessRequest), we have four cases, of which only the third ($x_i \in B_3$) is not present in ARC. In each of the other cases, we have at most two independent operations—one to insert into the Data Cache (if needed) and second to insert into the Metadata Cache (if needed). In InsertInDataCache, an appropriate victim to evict from D is one with no references in $T_1 \cup T_2$. However, to find such a victim, several items may have to be deleted from M, as indicated by the while loop. In CheckMetadataCache, if $T_1 \cup T_2 \cup B_3$ exceeds $C + X$, an item from B_3 is always evicted, possibly after moving something from $T_1 \cup B_1 \cup B_2$ (as achieved in ManageMetadataCache). Finally, ReplaceInMetadataCache is similar to the Replace operation in original ARC and creates space for the new metadata item to be placed.

4.2.2 Analysis

To show that D-ARC complies with the no-wastage policy, we show that no orphans are created in the metadata contents of $T_1 \cup T_2$. (The B lists store historical metadata by design as they do in ARC, so we exclude them from the analysis.) On one hand, if a duplicated item is requested, it does not change the Data Cache, and therefore cannot create orphans in $T_1 \cup T_2$. On the other hand, every time a non-duplicated item is requested, it results in an insert into the Data Cache, causing some item to be evicted. As per the algorithms, the only evictions that are allowed involve items that have no source addresses in $T_1 \cup T_2$. If one exists, we are done and no orphans are created in the Metadata Cache by this insertion. If none exists, we "clear" items from $T_1 \cup T_2$ until we find an item in the Data Cache with no source address in $T_1 \cup T_2$, and then that item becomes the victim to be evicted. This victim cannot create an orphan in the Metadata Cache because of the way it is identified. Thus, D-ARC complies with the *no-wastage policy*.

Algorithm 3: D-ARC(C, X) subroutines

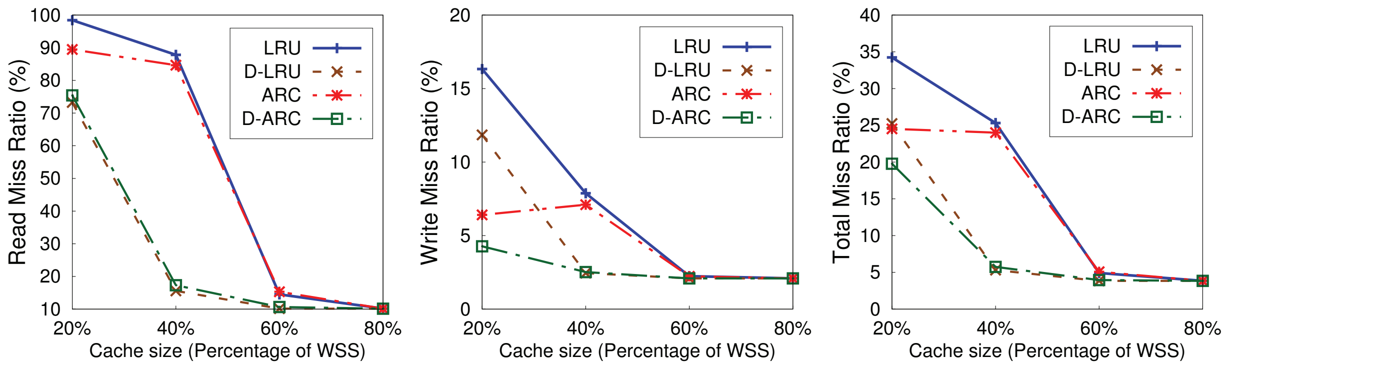
```
Subroutine InsertInDataCache()
    while no "victim" in $D$ with no references in
    $T_1 \cup T_2$ do
        ManageMetadataCache()
    replace LRU "victim" in $D$ with $h'(x_i)$
Subroutine CheckMetadataCache()
    if $|T_1| + |T_2| + |B_3| = C + X$ then
        if $|B_3| = 0$ then
            ManageMetadataCache()
        evict LRU item from $B_3$
Subroutine ManageMetadataCache()
    if $|T_1| + |B_1| \geq C$ then
        if $|T_1| < C$ then
            move LRU from $B_1$ to $B_3$
            ReplaceInMetadataCache()
        else
            if $|B_1| > 0$ then
                move LRU from $B_1$ to $B_3$
                move LRU from $T_1$ to $B_1$
            else
                move LRU from $T_1$ to $B_3$
    else
        if $|B_1| + |B_2| \geq C$ then
            move LRU from $B_2$ to $B_3$
        ReplaceInMetadataCache()
Subroutine ReplaceInMetadataCache()
    if $|T_1| > 0 \wedge (|T_1| > p \vee (x_i \in B_2 \wedge |T_1| = p))$
    then
        move LRU from $T_1$ to $B_1$
    else
        move LRU from $T_2$ to $B_2$
```

Complexity. The ARC-based insert to the Metadata Cache can be performed by D-ARC in constant time. However, an insert into the Data Cache may trigger repeated deletions from $T_1 \cup T_2$, which cannot be done in constant time. In fact, if $|T_1| + |T_2| = C + \delta$, for some number $\delta \leq X$, then at most δ evictions are needed for this operation. However, in order to have reached $C + \delta$ elements there must have been δ requests serviced in the past for which there were no evictions. So the amortized cost of each D-ARC request is still $O(1)$.

5 Evaluation

5.1 Methodology

Implementation: We created a practical prototype in Linux kernel space, based on block device virtualization [10]. It



(a) Read miss ratio

(b) Write miss ratio

(c) Total miss ratio

Figure 3: Miss ratio from WebVM

Name	Total I/Os I/Os (GB)	Working Set (GB)	Write-to -read ratio	Unique Data (GB)
WebVM	54.5	2.1	3.6	23.4
Homes	67.3	5.9	31.5	44.4
Mail	1741	57.1	8.1	171.3
Hadoop	23.6	14.4	0.4	3.7

Table 2: Trace statistics

can be deployed as a drop-in solution on existing systems that run Linux (including hypervisors that use the Linux I/O stack [6, 3]). It appears as a virtual block device to the applications/VMs if deployed on the client side, or to the storage services (e.g., iSCSI, NFS) if deployed on the server side.

Storage setup: We evaluated the real I/O performance of CacheDedup as the client-side flash cache for an iSCSI-based storage system, a widely used network storage protocol. The client and server each runs on a node with two six-core 2.4GHz Xeon CPUs and 24GB of RAM. The client uses a 120GB MLC SATA SSD as the cache, and the server uses a 1TB 7.2K RPM SAS disk as the target. Both nodes run Linux with kernel 3.2.20.

Traces: For our evaluation, we replayed the FIU traces [18]. These traces were collected from a VM hosting the departmental websites for webmail and online course management (*WebVM*), a file server used by a research group (*Homes*), and a departmental mail server (*Mail*). To study the support for concurrent workloads, we also collected traces (*Hadoop*) from a set of Hadoop VMs used to run MapReduce course projects. The characteristics of these traces are summarized in Table 2 where every I/O is of 4KB size. The working set size of a trace is calculated by counting the number of unique addresses of the I/Os in the trace and then multiplying it by the I/O size.

Metrics: We use the following metrics to compare the different cache-management approaches.

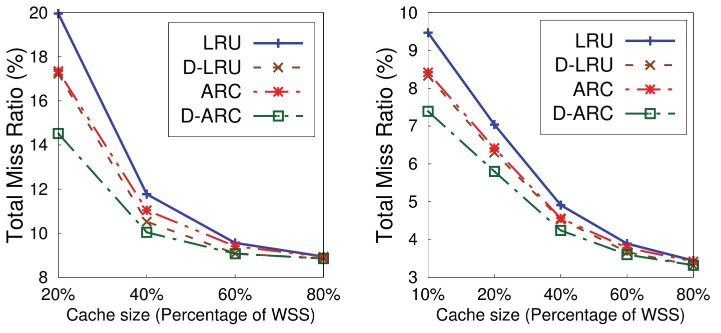
1) *Miss ratio:* We report both read miss ratio and write miss ratio, which are the numbers of reads and writes, respectively, over the total number of I/Os received by the cache. When the write-through policy is used, the read miss ratio is more important because writes always have to be performed on both cache and primary storage [11]. When the write-back policy is used, writes that hit the cache are absorbed by the cache, but the misses cause writes to the primary storage (when they are evicted). Therefore, the write miss ratio has a significant impact on the primary storage's I/O load and the performance of read misses [17, 5]. Therefore, we focus on the results from the write-back policy where the read miss ratio is also meaningful for the write-through policy. We omit the results from the write-invalidate policy which performs poorly for the write-intensive traces, although the deduplication-aware approaches still make substantial improvements as for the other two write policies.

2) *I/O latency/throughput:* To understand how the improvement in cache hits translates to application-perceived performance, we measure the latency of I/Os from replaying the traces on the storage system described above. To evaluate the overhead of CacheDeup, we measure both the I/O latency and throughput using an I/O benchmark, fio [1].

3) *Writes to flash ratio:* Without assuming knowledge of a flash device's internal garbage collection algorithm, we use the percentage of writes sent to the flash device, w.r.t. the total number of I/Os received by the cache manager, as an indirect metric of wear-out.

Cache configurations: We compare several different cache management approaches: 1) *LRU* (without deduplication); 2) *ARC* (without deduplication); 3) *D-LRU*; 4) *D-ARC*. To compare to the related work *Nitro* [19], we also created *CD-ARC*, a new ARC-based cache management approach that is aware of both duplication and compressibility.

For all the approaches that we considered, we show the results from the fault-tolerance configuration that keeps the entire metadata persistent (as discussed in Section 3.3). The



(a) Homes **(b) Mail**

Figure 4: Miss ratio from Homes and Mail

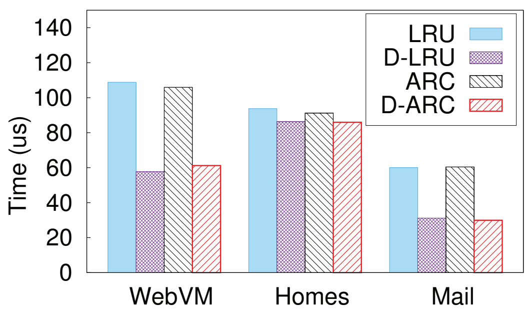


Figure 5: I/O latency from WebVM, Homes, and Mail with a cache size that is 40% of their respective WSS

same size of flash storage is used to store both data and metadata, so the comparison is fair. As discussed in Section 3.1, the Data Cache size can be traded for storing more historical fingerprints in the Metadata Cache as an optimization in the duplication-aware approaches. This tradeoff is studied in Section 5.4. In the other experiments, the Metadata Cache size is fixed at its minimum size (which is under 1% and 3% of the Data Cache size for D-LRU and D-ARC, respectively) plus an additional 2% taken from the Data Cache.

5.2 Performance

We evaluate the cache performance for each trace with different total cache sizes that are chosen at 20%, 40%, 60%, and 80% of its total working set size (WSS), which is listed in Table 2. Figure 3 compares the miss ratios of the WebVM trace. Results reveal that the duplication-aware approaches, D-LRU and D-ARC, outperform the alternatives by a significant margin for most cache sizes in terms of both read miss ratio and write miss ratio. Comparing the total miss ratio, D-LRU reduces it by up to 20% and D-ARC by up to 19.6%. Comparing LRU and ARC, ARC excels at small cache sizes, which is leveraged by D-ARC to also outperform D-LRU. For example, when the cache size is 20% of the trace's WSS, D-ARC has about 5% lower total miss ratio than D-LRU.

As discussed in Section 3, keeping historical source addresses and fingerprints in the Metadata Cache can help improve the hit ratio, because when the data block that a historical fingerprint maps to is brought back to the Data Cache, all the source addresses that map to this fingerprint can generate hits when they are referenced again. To quantify this benefit, we also measured the percentage of read hits that are generated by the historical metadata. For WebVM with a cache size that is 20% of its WSS, 83.25% of the read hits are produced by the historical metadata, which confirms the effectiveness of this optimization made by CacheDedup.

For the Homes and Mail traces, we show only the total miss ratio results to save space (Figure 4). Because the Mail trace is much more intensive than the other traces, we also show the results from the cache size that is 10% of its total WSS. Overall, D-ARC has the lowest total miss ratio, followed by D-LRU. D-ARC reduces the miss ratio by up to 5.4% and 3% compared to LRU and ARC, respectively in Homes and up to 2% and 1% in Mail. Compared to D-LRU, it reduces misses by up to 2.71% and 0.94% in Homes and Mail, respectively.

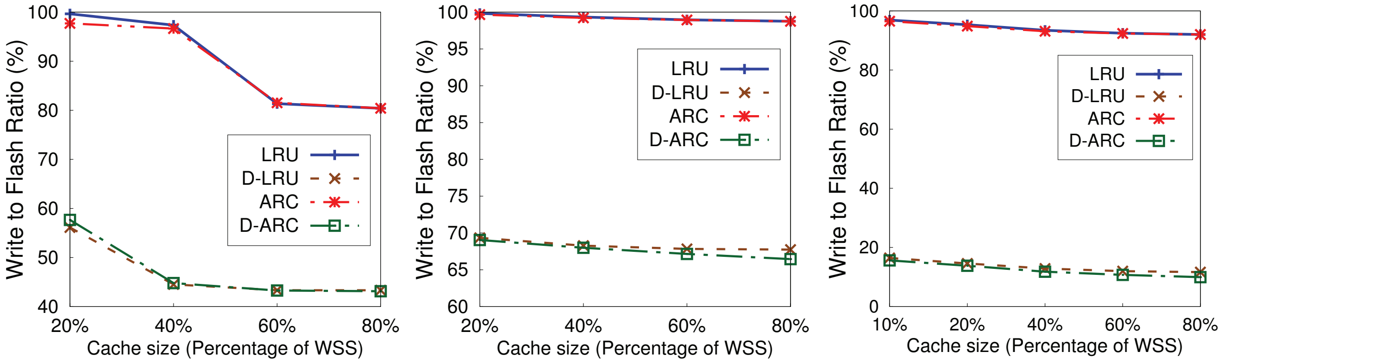
Figure 5 shows the average I/O latency from replaying the three traces with a cache size of 40% of their respective WSS. D-LRU and D-ARC deliver similar performance and reduce the latency by 47% and 42% compared to LRU and ARC, respectively for WebVM, 8% and 6% for Homes, and 48% and 51% for Mail. The improvement for Homes is smaller because of its much higher write-to-read ratio; the difference between a write hit and write miss is small when the storage server is not saturated. Note that the latency here does not include the fingerprinting time, which is $< 20\mu$s per fingerprint, since the fingerprints are taken directly from the traces. But we cannot simply add this latency to the results here because many cache hits do not require fingerprinting. Instead, we evaluate this overhead in Section 5.7 using a benchmark.

5.3 Endurance

The results in Figure 6 confirm that the two duplication-aware approaches can substantially improve flash cache endurance by reducing writes sent to the flash device by up to 54%, 33%, and 89% for the WebVM, Homes, and Mail traces, respectively, compared to the traditional approaches. The difference between D-LRU and D-ARC is small. This interesting observation suggests that the scan-resistant nature of ARC does not help as much on endurance as it does on hit ratio. It is also noticeable that the flash write ratio decreases with increasing cache size, but the difference is small for Homes and Mail and for WebVM after the cache size exceeds 40% of its WSS. This can be attributed to two opposite trends: 1) an increasing hit ratio reduces cache replacements and the corresponding flash writes; 2) but when write hits bring new data into the cache they still cause flash writes, which is quite common for these traces.

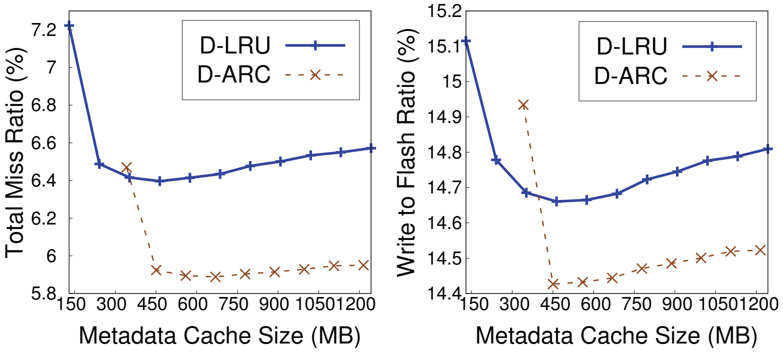
5.4 Sensitivity Study

We used the Mail trace to evaluate the impact of partitioning the shared flash cache space between the Data Cache and the



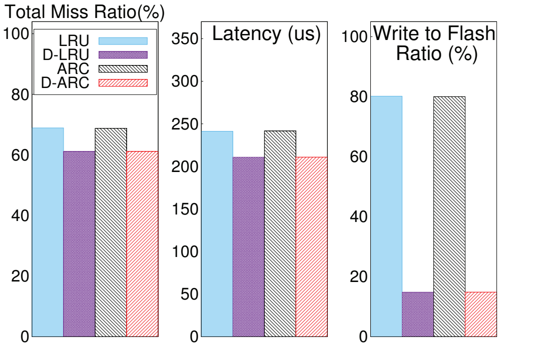
(a) WebVM **(b) Homes** **(c) Mail**

Figure 6: Writes to flash ratio



(a) Total miss ratio **(b) Writes to flash ratio**

Figure 7: D-LRU and D-ARC with varying Metadata/Data Cache space sharing for Mail



(a) **(b)** **(c)**

Figure 8: Results from concurrent Hadoop traces

Metadata Cache for the duplication-aware approaches. As discussed in Section 3.1, for a given Data Cache size, the minimum Metadata Cache size is what is required to store all the metadata of the cache data. But the Metadata Cache can take a small amount of extra space from the Data Cache for storing historical metadata and potentially improving performance. In this sensitivity study, we consider a cache size of 11GB which is 20% of Mail's WSS, and evaluate D-LRU and D-ARC while increasing the Metadata Cache size (and decreasing the Data Cache size accordingly). The results in Figure 7 show that both total miss ratio and wear-out initially improve with a growing Metadata Cache size. The starting points in the figure are the minimum Metadata Cache sizes for D-LRU and D-ARC (129MB for D-LRU and 341MB for D-ARC). Just by giving up 2% of the Data Cache space to hold more metadata (240MB of total Metadata Cache size for D-LRU and 450MB for D-ARC), the total miss ratio falls by 0.73% and 0.55% in D-LRU and D-ARC respectively, and the writes-to-flash ratio falls by 0.33% and 0.51%. The performance however starts to decay after having given up more than 3% of the Data Cache space, where the detrimental effect caused by having less data blocks starts to outweigh the benefit of being able to keep more historical metadata.

5.5 Concurrent Workloads

Next we evaluate CacheDedup's ability in handling concurrent workloads that share the flash cache and performing deduplication across them. We collected a set of VM I/O traces from a course where students conducted MapReduce programming projects using Hadoop. Each student group was given three datanode VMs and we picked three groups with substantial I/Os to replay. All the VMs were cloned from the same templates so we expect a good amount of duplicate I/Os to the Linux and Hadoop data. But the students worked on their projects independently so the I/Os are not identical. The statistics of these VM traces are listed as *Hadoop* in Table 2.

We replayed the last three days before the project submission deadline of these nine datanode VM traces concurrently, during which the I/Os are most intensive. The flash cache that they shared has a capacity of 40% of their total working set size. Figure 8 compares the performance of D-LRU and D-ARC to LRU and ARC. Overall, DLRU and DARC lower the miss ratio by 11% and the I/O latency by 12% while reducing the writes sent to the cache device by 81%. Notice that the reduction in flash write ratio is much higher than the reduction in miss ratio because, owing to the use of deduplication, a cache miss does not cause a write to the cache if the

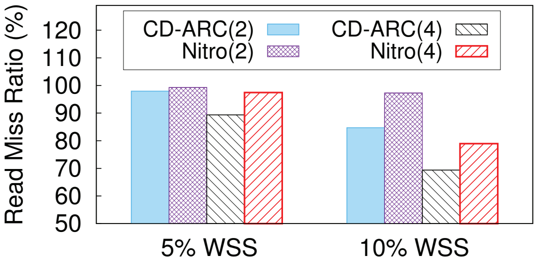


Figure 9: CD-ARC and Nitro for WebVM with a compression ratio of 2 and 4

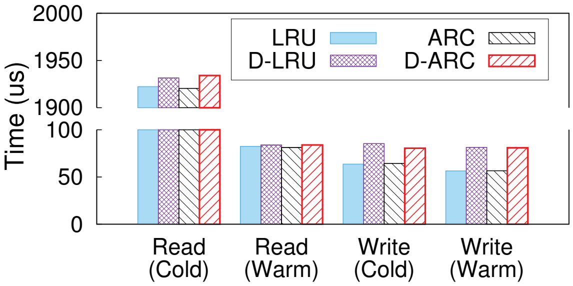


Figure 10: FIO latency with random reads and writes

requested data is a duplicate to the existing cached data.

5.6 Compression- and Duplication-aware Cache Management

Compression can be employed in addition to deduplication to further reduce the volume of data written to cache and improve cache performance and endurance. The recent work Nitro [19] was the first to combine these two techniques. It performs first deduplication and then compression on the data blocks. The compressed, variable-length data chunks (named extents) are packed into fixed-size blocks, named Write-Evict Units (WEU), and stored in cache. Cache replacement uses LRU at the granularity of WEUs. The size of a WEU is made the same as the flash device's erase block size to reduce garbage collection (GC) overhead. The fingerprints are managed using LRU separately from data replacement. If the primary storage also employs deduplication, Nitro can prefetch the fingerprints for identifying duplicates in future accesses.

CacheDedup is complementary to Nitro. On one hand, it can use the concept of WEU to manage compressed data in cache and reduce the flash device's internal GC overhead. On the other hand, CacheDedup can improve Nitro with its integrated cache management to reduce cache wastage and improve its performance and endurance. To prove this point, we created a version of D-ARC that is also compression-aware, named *CD-ARC*. It is still based on the integrated Metadata and Data Cache management architecture of CacheDedup. The main differences from D-ARC are that 1) the Data Cache stores WEUs; 2) the fingerprints in the Metadata Cache point to the extents in the WEUs; and 3) replacement in the Data Cache preferably uses a WEU with no mappings in $T_1 \cup T_2$.

We compare CD-ARC to a Nitro implementation without fingerprint prefetching because prefetching is an orthogonal technique that can be used by both approaches. But we extend Nitro to also cache historical fingerprints, so the comparison is fair. We also set the same limits on the two algorithms' data cache capacity and metadata space usage. We present the results from a 2MB WEU size with a compression ratio of 2 and 4, and report the read miss ratios in Figure 9 for the WebVM trace. CD-ARC improves the read hit ratio by up to 12.56% compared to Nitro. This improvement can be largely attributed to CD-ARC's scan-resistant and adaptive properties inherited from ARC, which is possible only because of the integrated cache and deduplication design of our approach.

Although compression can further reduce a workload's cache footprint and wear-out, it also adds additional overhead. Moreover, for a write-intensive workload which has a large number of updates, the use of compression also introduces wastage, because the updated data cannot be written in place in the cache. For example, for WebVM with a cache size that is 20% of its WSS and a compression ratio of 2, CD-ARC achieves only 8% higher read hit ratio, because in average 46% of the cache capacity is occupied by invalid data caused by updates.

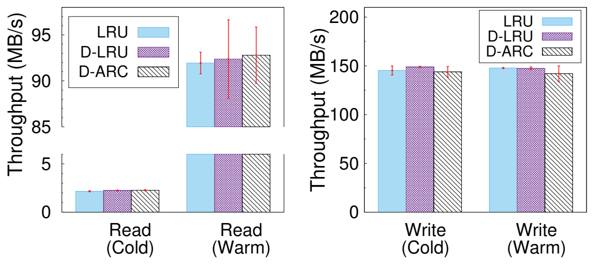
5.7 Overhead

Finally, we used an I/O benchmark fio [1] to measure CacheDedup's overhead compared to a stand-alone caching layer that does not use deduplication. The benchmark issued random reads or writes with no data duplication, so the results reveal the worst-case performance of CacheDedup. Direct I/O is used to bypass the main memory cache. First, we used a single fio thread with 1GB of random I/Os. Figure 10 shows that D-LRU and D-ARC adds a $10-20\mu$s latency to LRU and ARC for writes and for reads when the cache is cold, which is mainly the overhead of creating the fingerprint.

Although this fingerprinting overhead is considerable, in typical workloads, concurrent I/Os' fingerprinting operations can be overlapped by their I/Os and become insignificant in the overall performance. To demonstrate this, in the next experiment, we used eight concurrent fio threads each issuing 512MB of I/Os to evaluate the throughput overhead. Figure 11 shows that CacheDedup does not have significant overhead in terms of throughput. Moreover, CacheDedup's hit ratio gain and the corresponding performance improvement (as shown in the previous experiments) will significantly outweigh the fingerprinting overhead.

6 Related Work

A variety of research and commercial solutions have shown the effectiveness of flash caching for improving the performance of storage systems [7, 2, 5]. Compared to traditional main-memory-based caching, flash caching differs in its rel-



(a) Read throughput **(b) Write throughput**

Figure 11: FIO throughput with 8 concurrent threads

atively larger capacity and ability to store data persistently. Therefore, related work revisited several key design decisions for flash caching [11, 5]. In particular, it has been observed that write-through caching can provide good performance when the writes are submitted asynchronously to the backend storage, whereas write-back caching can reduce the I/O load on the backend and further improve the performance. The solution proposed here, CacheDedup, supports both write policies. Related work on the allocation of shared cache capacity among concurrent workloads [23, 20] is also complementary to CacheDedup's support for reducing each workload's footprint through in-line deduplication.

Deduplication has been commonly used to reduce the data footprint in order to save the bandwidth needed for data transfer and the capacity needed for data storage. Related solutions have been proposed to apply deduplication at different levels of a storage hierarchy, including backup and archival storage [27, 22], primary storage [12, 15], and main memory [33]. The wear-out issue of flash devices has motivated several flash deduplication studies [8, 9, 16] which show that it is a viable solution to improving endurance. Compared to these related efforts, CacheDedup addresses the unique challenges presented by caching. First, caching needs to process both reads and writes, while for primary storage only writes create new data and need to be considered. Second, the management of deduplication cannot be dissociated from cache management issues if localities have to be captured.

As discussed in Section 2, although a solution that simply stacks a standalone deduplication layer upon a standalone caching layer could also work, it would have a much higher metadata space overhead because both layers have to maintain the source block addresses, and the deduplication layer also has to manage the fingerprints for the entire source device. Moreover, such simple stacking cannot support a more sophisticated cache replacement algorithm such as ARC, which is made possible in D-ARC because of its integrated cache and deduplication management.

Nitro [19] is a closely related work which combines deduplication and compression to manage a flash cache employed at the server-side of a network file system. As discussed in Section 5.6, CacheDedup is complementary to Nitro in that our proposed architecture and algorithms can be incorporated to create a compression- and duplication-aware caching solution with further improved cache hit ratio and endurance.

As small random writes can decrease the throughput and device lifespan of a flash cache, related work RIPQ [32] proposed several techniques to address this problem, including aggregating the small random writes, which is similar to the WEU technique of Nitro, and our CD-ARC. It is also conceivable to apply WEU to D-LRU and D-ARC to aggregate small writes in order to sustain cache throughput and further improve flash endurance.

Related work studied cache admission policies to reduce flash wear-out [34, 14]. By not caching data with weak temporal locality, they showed improvements in endurance. Suei *et al.* [31] created a device-level cache partition design to distribute frequently-updated data into different erase blocks and lower the chances of blocks to be worn-out soon. These solutions are complementary to CacheDedup's focus on optimizing the use of deduplication for improving endurance.

7 Conclusions

This paper presents CacheDedup, a first study on integrating deduplication with flash caching using duplication-aware cache management. The novelties lie in a new architectural design that seamlessly integrates the caching of metadata and data, and new cache replacement algorithms D-LRU and D-ARC that allow the optimization for both performance and endurance. The paper offers an in-depth study of these algorithms with both theoretical analysis and experimental evaluation, which proves their no-cache-wastage property and shows the improvement on cache hit ratio, I/O latency, and the amount of writes sent to the cache device.

Between the two algorithms, D-ARC achieves the best performance, and D-LRU is attractive because of its simplicity. Both are efficient in terms of time and space usage. CacheDedup is a versatile framework for enabling various algorithms, including one (CD-ARC) that improves the use of compression with deduplication. As its design is not specific to flash devices, we believe that the CacheDedup approach can be also applied to new non-volatile memory technologies and improve their performance and endurance when used for caching.

8 Acknowledgements

We thank the anonymous reviewers and our shepherd, Geoff Kuenning, for their thorough reviews and insightful suggestions, and our colleagues at the VISA Research Lab, Dulcardo Arteaga, for his help with the caching framework, and Saman Biook Aghazadeh for his support of this paper including collecting the Hadoop traces. This research is sponsored by National Science Foundation CAREER award CNS-125394 and Department of Defense award W911NF-13-1-0157.

References

[1] Fio — Flexible I/O Tester Synthetic Benchmark. http://git.kernel.dk/?p=fio.git.

[2] Fusion-io ioCache. http://www.fusionio.com/products/iocache/.

[3] Kernel Based Virtual Machine. http://www.linux-kvm.org/page/Main_Page.

[4] Federal Information Processing Standards (FIPS) publication 180-1: Secure Hash Standard. National Institute of Standards and Technology (NIST), April 17, 1995.

[5] D. Arteaga and M. Zhao. Client-side flash caching for cloud systems. In *Proceedings of International Conference on Systems and Storage (SYSTOR)*, pages 7:1–7:11, New York, NY, USA, 2014. ACM.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, New York, 2003.

[7] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In *Proceedings of the 28th IEEE Conference on Massive Data Storage (MSST)*, Pacific Grove, CA, USA, 2012. IEEE.

[8] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, volume 11, 2011.

[9] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, pages 91–103, 2011.

[10] E. V. Hensbergen and M. Zhao. Dynamic policy disk caching for storage networking. Technical Report RC24123, IBM, November 2006.

[11] D. A. Holland, E. L. Angelino, G. Wald, and M. I. Seltzer. Flash caching on the storage client. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference (ATC)*. USENIX Association, 2013.

[12] B. Hong, D. Plantenberg, D. D. Long, and M. Sivan-Zimet. Duplicate data elimination in a SAN file system. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, pages 301–314, 2004.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[14] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with lazy adaptive replacement. In *Proceeding of 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2013.

[15] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of International Conference on Systems and Storage (SYSTOR)*, page 7. ACM, 2009.

[16] J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H.-u. Lee, S. Kang, Y. Won, and J. Cha. Deduplication in SSDs: Model and quantitative analysis. In *Proceedings of IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2012.

[17] R. Koller, L. Marmol, R. Ranganswami, S. Sundararaman, N. Talagala, and M. Zhao. Write policies for host-side flash caches. In *Proceedings of the 11th USENIX conference on File and Storage Technologies*, 2013.

[18] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage (TOS)*, 6(3):13, 2010.

[19] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 501–512. USENIX Association, 2014.

[20] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou. S-CAVE: Effective SSD caching to improve virtual machine storage performance. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 103–112. IEEE Press, 2013.

[21] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[22] D. Meister and A. Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of The Israeli Experimental Systems Conference (SYSTOR)*, page 8. ACM, 2009.

[23] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu. vCacheShare: Automated server flash cache space management in a virtualization environment. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2014.

[24] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):134–154, 1988.

[25] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. 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)*, pages 471–484. ACM, 2014.

[26] D. Qin, A. D. Brown, and A. Goel. Reliable write-

back for client-side flash caches. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 451–462. USENIX Association, 2014.

[27] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, volume 2, pages 89–101, 2002.

[28] R. Rivest. The MD5 message-digest algorithm. 1992.

[29] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, pages 267–280, New York, NY, USA, 2012. ACM.

[30] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communication. ACM*, 28(2):202–208, Feb. 1985.

[31] P.-L. Suei, M.-Y. Yeh, and T.-W. Kuo. Endurance-aware flash-cache management for storage servers. *IEEE Transactions on Computers (TOC)*, 63:2416–2430, 2013.

[32] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[33] C. A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[34] J. Yang, N. Plasson, G. Gillis, and N. Talagala. HEC: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)*, page 10. ACM, 2013.

Using Hints to Improve Inline Block-Layer Deduplication

Sonam Mandal,[1] Geoff Kuenning,[3] Dongju Ok,[1] Varun Shastry,[1] Philip Shilane,[4] Sun Zhen,[1,5]
Vasily Tarasov,[2] and Erez Zadok[1]
[1]*Stony Brook University,* [2]*IBM Research—Almaden,* [3]*Harvey Mudd College,*
[4]*EMC Corporation, and* [5]*HPCL, NUDT, China*

Abstract

Block-layer data deduplication allows file systems and applications to reap the benefits of deduplication without requiring per-system or per-application modifications. However, important information about data context (e.g., data vs. metadata writes) is lost at the block layer. Passing such context to the block layer can help improve deduplication performance and reliability. We implemented a hinting interface in an open-source block-layer deduplication system, *dmdedup,* that passes relevant context to the block layer, and evaluated two hints, NODEDUP and PREFETCH. To allow upper storage layers to pass hints based on the available context, we modified the VFS and file system layers to expose a hinting interface to user applications. We show that passing the NODEDUP hint speeds up applications by up to 5.3× on modern machines because the overhead of deduplication is avoided when it is unlikely to be beneficial. We also show that the PREFETCH hint accelerates applications up to 1.8× by caching hashes for data that is likely to be accessed soon.

1 Introduction

The amount of data that organizations store is growing rapidly [3]. Decreases in hard drive and SSD prices do not compensate for this growth; as a result companies are spending more and more on storage [26]. One technique to reduce storage costs is deduplication, which allows sites to store less raw data. At its core, deduplication systematically replaces duplicate data chunks with references. For many real-world datasets, deduplication significantly reduces raw storage usage [10, 19, 22].

Deduplication can be implemented at several layers in the storage stack. Most existing solutions are built into file systems [4, 28, 32] because they have enough information to deduplicate efficiently without jeopardizing reliability. For example, file sizes, metadata, and on-disk layout are known to the file system; often file systems are aware of the processes and users that perform I/O. This information can be leveraged to avoid deduplicating certain blocks (e.g., metadata), or to prefetch dedup metadata (e.g., for blocks likely to be accessed together).

An alternative is to add deduplication to the block layer, which provides a simple read/write interface. Because of this simplicity, adding features to the block layer is easier than changing file systems. This observation is equally applicable to systems that work directly at the block layer, such as databases and object stores.

However, a block-level deduplication system is unaware of the context of the data it operates on. A typical I/O request contains only the operation type (read or write), size, and offset, without attached semantics such as the difference between metadata and user data. Deduplicating metadata can (1) harm reliability [25], e.g., because many file systems intentionally save several copies of critical data such as superblocks, and (2) waste computational resources because typical metadata (inode tables, directory entries, etc.) exhibits low redundancy. In particular, in-line deduplication is expensive because forming chunks (fixed or variable-length), hash calculation, and hash searches are performed before writing data to disk; it is undesirable to expend resources on data that may not benefit from deduplication.

To allow block-layer deduplication to take context into account, we propose an interface that allows file systems and applications to provide simple *hints* about the context in which the deduplication is being performed. Such hints require only minor file system changes, making them practical to add to existing, mature file systems. We implemented two hints: NODEDUP and PREFETCH, which we found useful in a wide range of cases.

To evaluate the potential benefits of hints, we used an open-source block-layer deduplication system, *dmdedup* [34]. *Dmdedup* is meant for in-line primary-storage deduplication and is implemented as a stackable block device in the Linux kernel. We evaluated our hints under a variety of workloads and mixes of unique vs. deduplicable data. Our results demonstrate that by not deduplicating data that is likely to be unique, the NODEDUP hint can speed up applications by as much as 5.3× over vanilla *Dmdedup*. We also show that by preloading hashes for data that is likely to be deduplicated soon, the PREFETCH hint can speed up applications by as much as 1.8× over vanilla *Dmdedup*.

2 Background

Context recovery. Previous research has addressed the semantic gap between the block layer and a file system and has demonstrated that restoring all or part of the context can substantially improve block-level performance and reliability [2, 18, 29–31, 36]. We build on this observation by recovering partial file system and application context to improve block-level deduplication.

Context recovery can be achieved either by *introspection* or via *hinting*. Introspection relies on block-layer intelligence to infer file-system or application operations. The benefit of introspection is that it does not require any file-system changes; the disadvantage is that a successful implementation can be difficult [33, 35]. In contrast, *hinting* asks higher layers to provide small amounts of extra information to the deduplication system. Although file systems and perhaps applications must be changed, the necessary revisions are small compared to the benefits. Furthermore, application changes can be minimized by interposing a library that can deduce hints from information such as the file or program name, file format, etc. In this work, we use hinting to recover context at the block layer.

Dmdedup. We used an open-source block-layer deduplication system, *dmdedup* [34], to evaluate the benefits of hints. *Dmdedup* uses fixed-size chunking and relies on Linux's crypto API for hashing. It can use one of two metadata back ends: `inram` and `cowbtree`; the former stores the metadata only in RAM (if it is battery-backed), and the latter writes it durably to disk. Others also proposed a soft-update based metadata backend [5].

Figure 1 depicts *dmdedup's* main components and its position in a typical setup. *Dmdedup* rests on top of physical block devices (e.g., disk drives, RAIDs, SSDs), or other logical devices (e.g., dm-crypt for encryption). It typically requires two block devices to operate: one a *data* device that stores actual user data, and a *metadata* device that keeps track of the organizational information (e.g., the hash index). In our experiments, we used an HDD for data and SSD for metadata. Placing metadata on an SSD makes sense because it is much smaller than the data itself—often less than 1% of the data—but is critical enough to require low-latency access. To upper layers, *dmdedup* provides a conventional read/write block interface. Normally, every write to a *dmdedup* instance is hashed and checked against all existing data; if a duplicate is detected, the corresponding metadata is updated and no data is written. New non-duplicate content is passed to the data device and tracked in the metadata. Since only one instance of a given block is stored, multiple files may be affected if it gets corrupted. Therefore, *dmdedup* can be run over RAID or a replication system to minimize the risk of data loss.

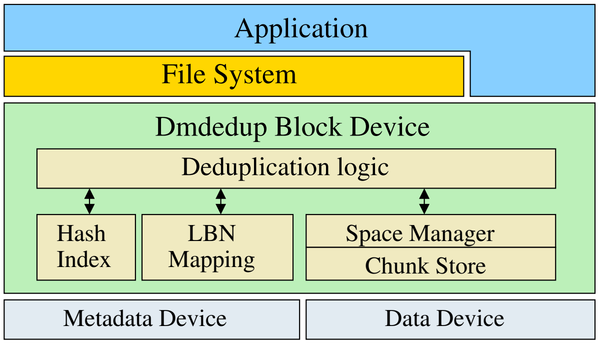


Figure 1: Dmdedup high-level design.

	Ext2	Ext3	Ext4	Nilfs2
% of writes that are metadata	11.6	28.0	18.9	12.1
% of unique metadata writes	98.5	57.6	61.2	75.0

Table 1: Percentage of metadata writes and unique metadata in different file systems.

Internally, *dmdedup* has five components (Figure 1): **(1)** the *deduplication logic* that chunks data, computes hashes, and coordinates other components; **(2)** a *hash index* that tracks the hashes and locations of all currently stored chunks; **(3)** a *mapping* between Logical Block Numbers (LBNs) visible to the upper layers and the Physical Block Numbers (PBNs) where the actual data is stored; **(4)** a *space manager* that tracks space on the data device, maintains reference counts, allocates new blocks, and reclaims unreferenced data; and **(5)** a *chunk store* that saves user data to the data device.

3 Potential Hints

Bypass deduplication. Some writes are known *a priori* to be likely to be unique. Applications might generate data that should not or cannot be deduplicated. For example, some applications write random, compressed, or encrypted data; others write complex formats (e.g., virtual disk images) with internal metadata that tends to be unique [8]. HPC simulations often generate massive checkpoints with unique data, and high-resolution sensors produce unique data streams.

Attempting to deduplicate unique writes wastes CPU time on hash computation and I/O bandwidth on maintaining the hash index. Unique hashes also increase the index size, requiring more RAM space and bandwidth for lookup, insertion, and garbage collection.

Most file system metadata is unique—e.g., inodes (which have varying timestamps and block pointers), directory entries, and indirect blocks. Table 1 shows the percentage of 4KB metadata writes (unique and overall) for several file systems, using Filebench's [7] Fileserver workload adjusted to write 4KB blocks instead of 1KB (so as to match the deduplication system's chunk size). About 12–28% of the total writes across all file systems were metadata; in all cases at least 57% of the metadata was unique. Ext3 and Ext4 have more metadata duplicates than Ext2 and Nilfs2 (43% vs. 1–25%), a phenomenon caused by journaling: Ext4 initially writes metadata blocks to the journal and then writes the same blocks to their proper location on the disk.

Metadata writes are more important to overall system performance than data writes because the former are often synchronous. Adding extra deduplication overhead might increase the latency of those critical metadata writes. Avoiding excessive metadata deduplication also helps reliability because many file systems store redundant copies of their metadata (e.g., Ext2/3/4 keeps multiple superblocks; ZFS explicitly duplicates metadata to avoid corruption). Deduplicating those copies would obviate this feature. Likewise, file system journals enhance reliability, so deduplicating their blocks might be counterproductive. A deduplicated journal would also lose sequentiality, which could harm performance.

In summary, if a block-level deduplication system can know when it is unwise to deduplicate a write, it can optimize its performance and reliability. We implemented a NODEDUP hint that informs our system that a corresponding request should not be deduplicated.

Prefetch hashes. When a deduplication system knows what data is about to be written, it can prefetch the corresponding hashes from the index, accelerating future data writes by reducing lookup delays. For example, a copying process first reads source data and then writes it back. If a deduplication system can identify that behavior at read time, it can prefetch the corresponding hash entries from the index to speed up the write path. We implemented this hint and refer to it as PREFETCH. Another interesting use case for this hint is segment cleaning in log-structured file systems (e.g., Nilfs2) that migrate data between segments during garbage collection.

Bypass compression. Some deduplication systems compress chunks to save further space. However, if a file is already compressed (easily determined), additional compression consumes CPU time with no benefit.

Cluster hashes. Files that reside in the same directory tend to be accessed together [12]. In a multi-user environment, a specific user's working set is normally far smaller than the whole file system tree [13]. Based on file ownership or on which directories contain files, a deduplication system could group hashes in the index and pre-load the cache more efficiently.

Partitioned hash index. Partitioning the hash index based on incoming chunk properties is a popular technique for improving deduplication performance [1]. The chance of finding a duplicate in files of the same type is higher than across all files, so one could define partitions using, for example, file extensions.

Intelligent chunking. Knowing file boundaries allows a deduplication system to efficiently chunk data. Certain large files (e.g., tarballs) contain many small ones. Passing information about content boundaries to the block layer would enable higher deduplication ratios [15].

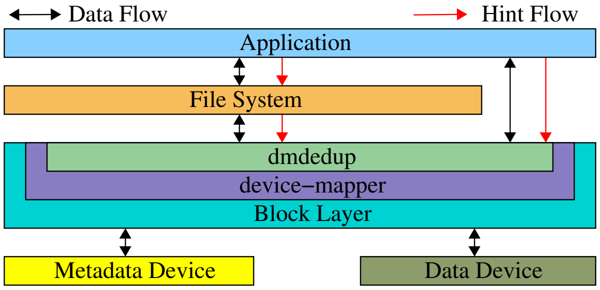


Figure 2: Flow of hints across the storage layers.

4 Design and Implementation

To allow the block layer to be aware of context, we designed a system that lets hints flow from higher to lower layers in the storage stack. Applications and file systems can then communicate important information about their data to lower layers. The red arrows in Figure 2 show how hints are passed to the block layer. We have implemented two important hints: NODEDUP and PREFETCH.

Nodedup. Since deduplication uses computational resources and may increase latency, it should only be performed when there is a potential benefit. The NODEDUP hint instructs the block layer not to deduplicate a particular chunk (block) on writes. It has two use cases: **(1)** unique data: there is no point in wasting resources on deduplicating data that is unlikely to have duplicates, such as sensor or encrypted data; **(2)** reliability: maintaining multiple copies of certain blocks may be necessary, e.g., superblock replicas in many file systems.

Prefetch. One of the most time-consuming operations in a deduplication system is hash lookup, because it often requires extra I/O operations. Worse, hashes are randomly distributed by their very nature. Hence, looking up a hash often requires random I/O, which is the slowest operation in most storage systems. Also, as previous studies have shown [38], it is impractical to keep all the hashes in memory because the hash index is far too large.

The PREFETCH hint is used to inform the deduplication system of I/O operations that are likely to generate further duplicates (e.g., during a file copy) so that their hashes can be prefetched and cached to minimize random accesses. This hint can be set on the read path for applications that expect to access the same data again. (Note that reads normally only need to access the LBN→PBN index, bypassing the hash index.)

4.1 Implementation

To add support for hints, we modified various parts of the storage stack. The generic changes to support propagation of hints from higher levels to the block layer modified about 77 lines of code in the kernel. We also modified the OPEN system call to take two new flags, O_NODEDUP and O_PREFETCH. User-space applications can use these flags to pass hints to the underlying deduplication block device. If the block layer does not support the flags, they are ignored. Applications that require redundancy or have a small number of duplicates can pass the O_NODEDUP hint when `open`ing for write. Similarly, applications that are aware of popular data blocks, or that know some data will be accessed again, can pass O_PREFETCH when `open`ing for read. Hashes of the blocks being read can then be prefetched, so that on a later write they can be found in the prefetch cache.

We modified *dmdedup* to support the NODEDUP and PREFETCH hints by adding and changing about 741 LoC. In *dmdedup*, if a request has the NODEDUP flag set, we skip lookups and updates in the hash index. Instead, we add an entry only to the LBN→PBN mapping. The read path needs no changes to support NODEDUP.

On the read path in *dmdedup*, the LBN→PBN map is consulted to find whether the given location is known, but no hash calculation is normally necessary because a previous write would have already added the block to the hash→PBN map. If a request has the PREFETCH hint set on the read path then *dmdedup* hashes the data after it is read and puts the corresponding hash→PBN tuple in a prefetch cache. Upon writes, our code saves execution time by checking the cache before searching the metadata backend. When a hash is found in the prefetch cache, it is evicted, since after the copy there is little reason to believe that it will be used again soon.

We also modified some specific file systems to pass the NODEDUP hint for their metadata and also pass the OPEN flags to the block layer if set. For Linux's Nilfs2, we changed about 371 kernel LoC to mark its metadata with hints and propagate them, along with the OPEN flags, from the upper levels to the block layer. Similar changes to Ext4 changed 16 lines of code; in Ext3 we modified 6 lines (which also added support for Ext2). The Ext2/3/4 changes were small because we were able to leverage the (newer) REQ_META flag being set on the file system metadata to decide whether to deduplicate based on data type. The rest of the metadata-related hints are inferred; we identify journal writes from the process name, *jbd2*.

5 Evaluation

Experimental Setup. In our experiments we used a Dell PowerEdge R710, equipped with an Intel Xeon E5540 2.4GHz 4-core CPU and 24GB of RAM. The machine ran Ubuntu Linux 14.04 x86_64, upgraded to a Linux 3.17.0 kernel. We used an Intel DC S3700 series 200GB SSD as the *dmdedup* metadata device and a Seagate Savvio 15K.2 146GB disk drive for the data. Both drives were connected to the host using Dell's PERC 6/i controller. Although the SSD is large, in all our experiments we used 1.5GB or less for *dmdedup* metadata.

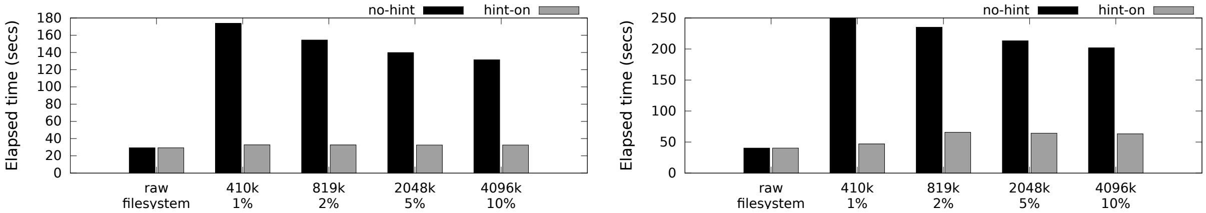
We ran all experiments at least three times and ensured that standard deviations were less than 5% of the mean. To ensure that all dirty data reached stable media in the micro-workload experiments, we called `sync` at the end of each run and then unmounted the file system; our time measurements include these two steps.

For all experiments we used *dmdedup*'s *cowbtree* transactional metadata backend, since it helps avoid inconsistent metadata states on crashes. *Cowbtree* allows users to specify different metadata cache sizes; we used sizes of 1%, 2%, 5%, and 10% of the deduplication metadata for each experiment. These ratios are typical in real deduplication systems. *Dmdedup* also allows users to specify the granularity at which they want to flush metadata. We ran all experiments with two settings: flush metadata on every write, or flush after every 1,000 writes. In our results we focus on the latter case because it is a more realistic setting. Flushing after every write is like using the O_SYNC flag for every operation and is uncommon in real systems; we used that setting to achieve a worst-case estimate. *Dmdedup* also flushes its metadata when it receives any flush request from the layers above. Thus, *dmdedup*'s data persistency semantics are the same as those of a regular block device.

5.1 Experiments

We evaluated the NODEDUP and PREFETCH hints for four file systems: Ext2, Ext3, Ext4, and Nilfs2. Ext2 is a traditional FFS-like file system that updates metadata in place; Ext3 adds journaling and Ext4 further adds extent support. Nilfs2 is a log-structured file system: it sequentializes all writes and has a garbage-collection phase to remove redundant blocks. We show results only for Ext4 and Nilfs2, because we obtained similar results from the other file systems. In all cases we found that the performance of Nilfs2 is lower than that of Ext4; others have seen similar trends [27].

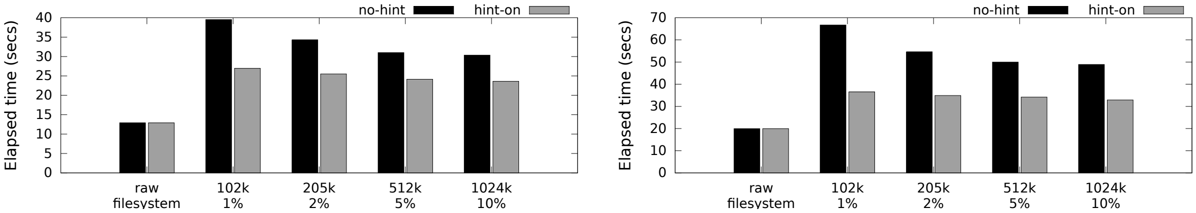
NODEDUP hint. To show the effectiveness of application-layer hints, we added the NODEDUP hint as an open flag on `dd`'s write path. We then created a 4GB file with unique data, testing with the hint both on and off. This experiment shows the benefit of the NODEDUP hint on a system where unique data is being written (i.e., where deduplication is not useful), or where reliability considerations trump deduplicating. This hint might not be as helpful in workloads that produce many duplicates. Figure 3 shows the benefit of the NODEDUP hint for Ext4 and Nilfs2 when metadata was flushed every 1,000 writes; results for other file systems were similar. We found that the NODEDUP hint decreased unique-data write times by 2.2–5.3×. Flushing *dmdedup's* metadata after every write reduced the benefit of the NODEDUP hint, since the I/O overhead was high, but we still observed improvements of 1.3–1.6×.



(a) Ext4

(b) Nilfs2

Figure 3: Performance of using `dd` *to create a 4GB file with unique content, both with and without the* NODEDUP *hint, for different file systems. The X axis lists the metadata cache size used by* dmdedup*, in both absolute values and as a percentage of the total metadata required by the workload. Dmdedup metadata was flushed after every 1,000 writes. Lower is better.*



(a) Ext4

(b) Nilfs2

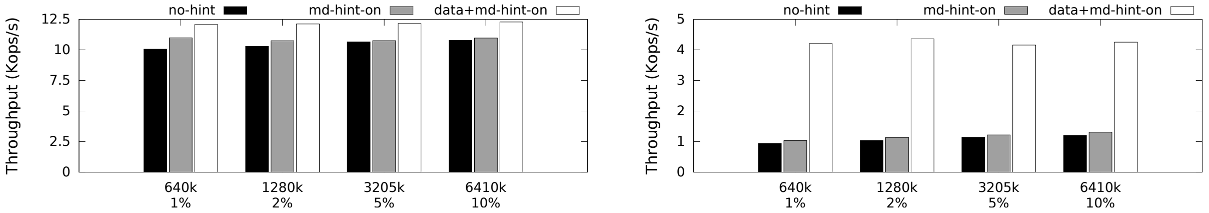
Figure 4: Performance of using `dd` *to copy a 1GB file, both with and without the* PREFETCH *hint, for different file systems. The X axis lists the metadata cache size used by* dmdedup*, in both absolute values and as a percentage of the total metadata required by workload. Dmdedup metadata was flushed after every 1,000 writes. Lower is better.*

PREFETCH hint. To evaluate the PREFETCH hint we modified `dd` to use the O_PREFETCH open flag on the read path so that writes could benefit from caching hashes. We then used the modified `dd` to repeatedly copy a 1GB file with unique content within a single file system. We used unique content so that we could measure the worst-case performance where no deduplication can happen, and to ensure that the prefetch cache was heavily used. We also performed studies on a locally collected dataset of the hashes of the home directories of a small research group. We analyzed the hashes to learn how many duplicate blocks are seen within a file using 4KB chunk sizes, and found that 99% of the files had unique chunks. Thus testing the PREFETCH hint with unique content makes sense. For all four file systems, the results were similar because most file systems manage single-file data-intensive workloads similarly. Figure 4 shows results for Ext4 and Nilfs2. When flushing *dmdedup*'s metadata every 1,000 writes, the reduction in copy time compared to the no-hint configuration was 1.2–1.8×. When we flushed the metadata after every write, the copy times ranged from 16% *worse* to 16% better. The improvement from hints was less significant here because the overhead of flushing was higher than the benefit obtained from prefetching the hashes.

Not suprisingly, the effectiveness of PREFETCH hint depends on the deduplication ratio. For example, when we changed the deduplication ratio to 8:1 in the above experiment, the copy times ranged from 9% *worse* to 55% better depending on file system type and *dmdedup* settings.

Macro workload. We modified *Filebench* to generate data in the form of a given duplicate distribution instead of arbitrary data. We then ran *Filebench's* `Fileserver` workload, modified to write 4KB blocks, to assess the benefit of setting the NODEDUP hint for: (1) file metadata writes, where we mark the metadata blocks and the journal writes with this hint, and (2) file data writes along with the metadata writes. We used a unique-write workload to show the benefits of applying the NODEDUP hint for applications writing unique data. Figure 5 shows the maximal benefit of setting the NODEDUP hint on for file metadata writes alone, and for data and metadata writes. We ran Filebench, with the all-unique writes being flushed after 1,000 writes. When setting the NODEDUP hint only for metadata writes, we saw an increase in throughput of 1–10%. When we set the hint for both data and metadata writes, we saw an improvement in throughput of 1.1–1.2× for Ext4, and 3.5–4.5× for Nilfs2. When we set the NODEDUP hint for metadata only, we observed an increase in performance but a decrease in deduplication. As calculated from Table 1, about 7.3% of all writes in Ext4 and 3.0% of all writes in Nilfs2 are duplicated file-system metadata writes. *Dmdedup* would save extra space by deduplicating these writes if the NODEDUP hint was not set. In other words, the hint trades higher throughput and reliability for a lower deduplication ratio.

We also ran a similar experiment (not shown for brevity) where Filebench generated data with a dedup ratio of 4:1 (3 duplicate blocks for every unique one). We set the NODEDUP hint for metadata writes only (because Filebench generated unique data on a per-write ba-



(a) Ext4

(b) Nilfs2

Figure 5: Throughput obtained using Filebench's `Fileserver` *workload modified to write all-unique content, for different file systems. Throughput is shown with the* NODEDUP *hint off (no-hint); with the hint on for file system metadata only (md-hint-on); and with the hint on for both file system metadata and data (data+md-hint on). The X axis lists the* dmdedup *metadata cache size, in both absolute values and as a percentage of an estimate of the total metadata required by the workload. Dmdedup metadata was flushed after every 1,000 writes. Higher is better.*

sis whereas our hint works on a per-file basis), and compared this to the case where the NODEDUP hint was off. We saw a modest improvement in throughput, ranging from 4–7% for Ext4 and 6–10% for Nilfs2.

6 Related Work

The semantic divide between the block layer and file systems has been addressed previously [2,6,29–31] and has received growing attention because of the widespread use of virtualization and the cloud, which places storage further away from applications [9,11,17,23,24,35].

An important approach to secondary storage is ADMAD, which performs application-aware file chunking before deduplicating a backup [16]. This is similar to our hints interface, which can be easily extended to pass application-aware chunk-boundary information.

Many researchers have proposed techniques to prefetch fingerprints and accelerate deduplication filtering [14,37,38]. While these techniques could be added to *dmdedup* in the future, our current focus is on providing semantic hints from higher layers, which we believe is an effective complementary method for accelerating performance. In addition, some of these past techniques rely on workload-specific data patterns (e.g., backups) that might not be beneficial in general-purpose in-line primary-storage deduplication systems.

Studies of memory deduplication in virtualized environments [20,21] show a benefit of closing the semantic gap caused by multiple virtualization layers. There, memory is scanned by the host OS to identify and merge duplicate pages. Such scanning is expensive, misses short-lived pages, and is slow to identify longer-lived duplicates. However, these studies found that pages in the guest's unified buffer cache are good sharing candidates, so marking requests from the guest OS with a dedup hint can help to quickly identify potential duplicates. This approach is specific to memory deduplication and may not apply to storage systems where we identify duplicates before writing to the disk.

Lastly, others have demonstrated a loss of potential deduplication opportunities caused by intermixing metadata and data [15], showing that having hints to avoid unnecessary deduplication might be beneficial.

7 Conclusions and Future Work

Deduplication at the block layer has two main advantages: (1) allowing any file system and application to benefit from deduplication, and (2) ease of implementation [34]. Unfortunately, application and file system context is lost at the block layer, which can harm deduplication's effectiveness. However, by adding simple yet powerful hints, we were able to provide the missing semantics to the block layer, allowing the dedup system to improve performance and possibly also reliability. Our experiments show that adding the NODEDUP hint to applications like `dd` can improve performance by up to 5.3× when copying unique data, since we avoid the overhead of deduplication for data that is unlikely to have duplicates. This hint can be extended to other applications, such as those that compress or encrypt. Adding the PREFETCH hint to applications like `dd` improved copying time by as much as 1.8× because we cache the hashes and do not need to access the metadata device to fetch them on the write path. Adding hints to macro workloads like *Filebench's* `Fileserver` workload improved throughput by as much as 4.5×. Another important note is that the effectiveness of hints depends on both the overhead added by the deduplication system, the nature of the data being written (e.g., deduplication ratio), and the workload, so all factors need to be considered when choosing to use hints.

Future work. Because of the success of our initial experiments, we intend to add hint support to other file systems, such as Btrfs and XFS. We also plan to implement other hints, discussed in Section 3, to provide richer context to the block layer, along with support to pass additional information (e.g. inode numbers) that can be used to enhance hints. We also plan to add the PREFETCH hint to Nilfs2 for segment cleaning.

Acknowledgments. We thank the anonymous FAST reviewers for their useful feedback. This work was made possible in part thanks to EMC support and NSF awards CNS-1251137 and CNS-1302246.

References

[1] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. Klein. The design of similarity based deduplication system. In *Proceedings of the Israeli Experimental Systems Conference (SYSTOR)*, 2009.

[2] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A non-invasive exclusive caching mechanism for RAIDs. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 176–187, Washington, DC, USA, 2004. IEEE Computer Society.

[3] R. E. Bohn and J. E. Short. How much information? 2009 report on American consumers. `http://hmi.ucsd.edu/pdf/HMI_2009_ConsumerReport_Dec9_2009.pdf`, December 2009.

[4] J. Bonwick. ZFS deduplication, November 2009. `http://blogs.oracle.com/bonwick/entry/zfs_dedup`.

[5] Zhuan Chen and Kai Shen. Ordermergededup: Efficient, failure-consistent deduplication on flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, Santa Clara, CA, February 2016. USENIX Association.

[6] A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the USENIX Annual Technical Conference*, 2009.

[7] Filebench. `http://filebench.sf.net`.

[8] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP '11)*, Cascais, Portugal, October 2011. ACM Press.

[9] X. Jiang and X. Wang. "Out-of-the-box" monitoring of VM-based high-interaction honeypots. In *Proceedings of the International Conference on Recent Advances in Intrusion Detection (RAID)*, 2007.

[10] K. Jin and E. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of the Israeli Experimental Systems Conference (SYSTOR)*, 2009.

[11] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 14–24, New York, NY, USA, 2006. ACM Press.

[12] T. M. Kroeger and D. D. E. Long. Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the Annual USENIX Technical Conference (ATC)*, pages 105–118, Boston, MA, June 2001. USENIX Association.

[13] G. H. Kuenning, G. J. Popek, and P. Reiher. An analysis of trace data for predictive file caching in mobile computing. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 291–303, June 1994.

[14] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, 2009.

[15] Xing Lin, Fred Douglis, Jim Li, Xudong Li, Robert Ricci, Stephen Smaldone, and Grant Wallace. Metadata considered harmful ... to deduplication. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'15, pages 11–11, Berkeley, CA, USA, 2015. USENIX Association.

[16] C. Liu, Y. Lu, C. Shi, G. Lu, D. Du, and D.-S. Wang. ADMAD: Application-driven metadata aware de-duplication archival storage system. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2008.

[17] Bo Mao, Hong Jiang, Suzhen Wu, and Lei Tian. POD: Performance oriented I/O deduplication for primary storage systems in the cloud. In *28th International IEEE Parallel and Distributed Processing Symposium*, 2014.

[18] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 57–70, New York, NY, USA, 2011. ACM.

[19] D. Meyer and W. Bolosky. A study of practical deduplication. In *Proceedings of the Ninth USENIX Conference on File and Storage Technologies (FAST '11)*, 2011.

[20] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. KSM++: Using I/O-based hints to make memory-deduplication scanners more efficient. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized*

Environments (RESoLVE'12), London, UK, March 2012.

[21] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 279–290, San Jose, CA, 2013. USENIX.

[22] N. Park and D. Lilja. Characterizing datasets for data deduplication in backup applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2010.

[23] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening black boxes: Using semantic information to combat virtual machine image sprawl. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, Seattle, WA, March 2008. ACM.

[24] W. Richter, G. Ammons, J. Harkes, A. Goode, N. Bila, E. de Lara, V. Bala, and M. Satyanarayanan. Privacy-sensitive VM retrospection. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[25] David Rosenthal. Deduplicating devices considered harmful. *Queue*, 9(5):30:30–30:31, May 2011.

[26] J. Rydningcom and M. Shirer. Worldwide hard disk drive 2010-2014 forecast: Sowing the seeds of change for enterprise applications. IDC Study 222797, `www.idc.com`, May 2010.

[27] Ricardo Santana, Raju Rangaswami, Vasily Tarasov, and Dean Hildebrand. A fast and slippery slope for file systems. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pages 5:1–5:8, New York, NY, USA, 2015. ACM.

[28] Opendedup, January 2012. `www.opendedup.org`.

[29] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or death at block-level. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 379–394, San Francisco, CA, December 2004. ACM SIGOPS.

[30] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 15–30, San Francisco, CA, March/April 2004. USENIX Association.

[31] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 73–88, San Francisco, CA, March 2003. USENIX Association.

[32] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

[33] Sahil Suneja, Canturk Isci, Eyal de Lara, and Vasanth Bala. Exploring VM introspection: Techniques and trade-offs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2015.

[34] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, and S. Trehan. Dmdedup: Device-mapper deduplication target. In *Proceedings of the Linux Symposium*, pages 83–95, Ottawa, Canada, July 2014.

[35] Vasily Tarasov, Deepak Jain, Dean Hildebrand, Renu Tewari, Geoff Kuenning, and Erez Zadok. Improving I/O performance using virtual disk introspection. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, June 2013.

[36] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. ACM.

[37] W. Xia, H. Jiang, D. Feng, and Y. Hua. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[38] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, 2008.

NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories

Jian Xu Steven Swanson
University of California, San Diego

Abstract

Fast non-volatile memories (NVMs) will soon appear on the processor memory bus alongside DRAM. The resulting hybrid memory systems will provide software with sub-microsecond, high-bandwidth access to persistent data, but managing, accessing, and maintaining consistency for data stored in NVM raises a host of challenges. Existing file systems built for spinning or solid-state disks introduce software overheads that would obscure the performance that NVMs should provide, but proposed file systems for NVMs either incur similar overheads or fail to provide the strong consistency guarantees that applications require.

We present NOVA, a file system designed to maximize performance on hybrid memory systems while providing strong consistency guarantees. NOVA adapts conventional log-structured file system techniques to exploit the fast random access that NVMs provide. In particular, it maintains separate logs for each inode to improve concurrency, and stores file data outside the log to minimize log size and reduce garbage collection costs. NOVA's logs provide metadata, data, and mmap atomicity and focus on simplicity and reliability, keeping complex metadata structures in DRAM to accelerate lookup operations. Experimental results show that in write-intensive workloads, NOVA provides 22% to 216× throughput improvement compared to state-of-the-art file systems, and 3.1× to 13.5× improvement compared to file systems that provide equally strong data consistency guarantees.

1. Introduction

Emerging non-volatile memory (NVM) technologies such as spin-torque transfer, phase change, resistive memories [2, 28, 52] and Intel and Micron's 3D XPoint [1] technology promise to revolutionize I/O performance. Researchers have proposed several approaches to integrating NVMs into computer systems [11, 13, 19, 31, 36, 41, 58, 67], and the most exciting proposals place NVMs on the processor's memory bus alongside conventional DRAM, leading to hybrid volatile/non-volatile main memory systems [4, 51, 72, 78]. Combining faster, volatile DRAM with slightly slower, denser non-volatile main memories (NVMMs) offers the possibility of storage systems that combine the best characteristics of both technologies.

Hybrid DRAM/NVMM storage systems present a host of opportunities and challenges for system designers. These systems need to minimize software overhead if they are to fully exploit NVMM's high performance and efficiently support more flexible access patterns, and at the same time they must provide the strong consistency guarantees that applications require and respect the limitations of emerging memories (e.g., limited program cycles).

Conventional file systems are not suitable for hybrid memory systems because they are built for the performance characteristics of disks (spinning or solid state) and rely on disks' consistency guarantees (e.g., that sector updates are atomic) for correctness [47]. Hybrid memory systems differ from conventional storage systems on both counts: NVMMs provide vastly improved performance over disks while DRAM provides even better performance, albeit without persistence. And memory provides different consistency guarantees (e.g., 64-bit atomic stores) from disks.

Providing strong consistency guarantees is particularly challenging for memory-based file systems because maintaining data consistency in NVMM can be costly. Modern CPU and memory systems may reorder stores to memory to improve performance, breaking consistency in case of system failure. To compensate, the file system needs to explicitly flush data from the CPU's caches to enforce orderings, adding significant overhead and squandering the improved performance that NVMM can provide [6, 76].

Overcoming these problems is critical since many applications rely on atomic file system operations to ensure their own correctness. Existing mainstream file systems use journaling, shadow paging, or log-structuring techniques to provide atomicity. However, journaling wastes bandwidth by doubling the number of writes to the storage device, and shadow paging file systems require a cascade of updates from the affected leaf nodes to the root. Implementing either technique imposes strict ordering requirements that reduce performance.

Log-structured file systems (LFSs) [55] group small random write requests into a larger sequential write that hard disks and NAND flash-based solid state drives (SSDs) can process efficiently. However, conventional LFSs rely on the availability of contiguous free regions, and maintaining those regions requires expensive garbage collection operations. As a result, recent research [59] shows that LFSs perform worse than journaling file systems on NVMM.

To overcome all these limitations, we present the *NOn-Volatile memory Accelerated (NOVA)* log-structured file system. NOVA adapts conventional log-structured file system techniques to exploit the fast random access provided by hybrid memory systems. This allows NOVA to support massive concurrency, reduce log size, and minimize garbage collection costs while providing strong consistency guarantees for conventional file operations and mmap-based load/store accesses.

Several aspects of NOVA set it apart from previous log-structured file systems. NOVA assigns each inode a separate log to maximize concurrency during normal operation and recovery. NOVA stores the logs as linked lists, so they do not need to be contiguous in memory, and it uses atomic updates to a log's tail pointer to provide atomic log append. For operations that span multiple inodes, NOVA uses lightweight journaling.

NOVA does not log data, so the recovery process only needs to scan a small fraction of the NVMM. This also allows NOVA to immediately reclaim pages when they become stale, significantly reducing garbage collection overhead and allowing NOVA to sustain good performance even when the file system is nearly full.

In describing NOVA, this paper makes the following contributions:

- It extends existing log-structured file system techniques to exploit the characteristics of hybrid memory systems.
- It describes *atomic mmap*, a simplified interface for exposing NVMM directly to applications with a strong consistency guarantee.
- It demonstrates that NOVA outperforms existing journaling, shadow paging, and log-structured file systems running on hybrid memory systems.
- It shows that NOVA provides these benefits across a range of proposed NVMM technologies.

We evaluate NOVA using a collection of micro- and macro-benchmarks on a hardware-based NVMM emulator. We find that NOVA is significantly faster than existing file systems in a wide range of applications and outperforms file systems that provide the same data consistency guarantees by between 3.1× and 13.5× in write-intensive workloads. We also measure garbage collection and recovery overheads, and we find that NOVA provides stable performance under high NVMM utilization levels and fast recovery in case of system failure.

The remainder of the paper is organized as follows. Section 2 describes NVMMs, the challenges they present, and related work on NVMM file system design. Section 3 gives a overview of NOVA architecture and Section 4 describes the implementation in detail. Section 5 evaluates NOVA, and Section 6 concludes.

2. Background

NOVA targets memory systems that include emerging non-volatile memory technologies along with DRAM. This section first provides a brief survey of NVM technologies and the opportunities and challenges they present to system designers. Then, we discuss how other file systems have provided atomic operations and consistency guarantees. Finally, we discuss previous work on NVMM file systems.

2.1. Non-volatile memory technologies

Emerging non-volatile memory technologies, such as spin-torque transfer RAM (STT-RAM) [28, 42], phase change memory (PCM) [10, 18, 29, 52], resistive RAM (ReRAM) [22, 62], and 3D XPoint memory technology [1], promise to provide fast, non-volatile, byte-addressable memories. Suzuki *et al.* [63] provides a survey of these technologies and their evolution over time.

These memories have different strengths and weaknesses that make them useful in different parts of the memory hierarchy. STT-RAM can meet or surpass DRAM's latency and it may eventually appear in on-chip, last-level caches [77], but its large cell size limits capacity and its feasibility as a DRAM replacement. PCM and ReRAM are denser than DRAM, and may enable very large, non-volatile main memories. However, their relatively long latencies make it unlikely that they will fully replace DRAM as main memory. The 3D XPoint memory technology recently announced by Intel and Micron is rumored to be one of these and to offer performance up to 1,000 times faster than NAND flash [1]. It will appear in both SSDs and on the processor memory bus. As a result, we expect to see hybrid volatile/non-volatile memory hierarchies become common in large systems.

2.2. Challenges for NVMM software

NVMM technologies present several challenges to file system designers. The most critical of these focus on balancing the memories' performance against software overheads, enforcing ordering among updates to ensure consistency, and providing atomic updates.

Performance The low latencies of NVMMs alters the trade-offs between hardware and software latency. In conventional storage systems, the latency of slow storage devices (e.g., disks) dominates access latency, so software efficiency is not critical. Previous work has shown that with fast NVMM, software costs can quickly dominate memory latency, squandering the performance that NVMMs could provide [7, 12, 68, 74].

Since NVMM memories offer low latency and will be on the processor's memory bus, software should be able to access them directly via loads and stores. Recent NVMM-based file systems [21, 71, 73] bypass the DRAM page cache and access NVMM directly using a technique called *Direct Access (DAX)* or *eXecute In Place (XIP)*, avoiding extra copies between NVMM and DRAM in the storage stack. NOVA is a DAX file system and we expect that all NVMM file systems will provide these (or similar) features. We describe currently available DAX file systems in Section 2.4.

Write reordering Modern processors and their caching hierarchies may reorder store operations to improve performance. The CPU's memory consistency protocol makes guarantees about the ordering of memory updates, but existing models (with the exception of research proposals [20, 46]) do not provide guarantees on when updates will reach NVMMs. As a result, a power failure may leave the data in an inconsistent state.

NVMM-aware software can avoid this by explicitly flushing caches and issuing memory barriers to enforce write ordering. The x86 architecture provides the `clflush` instruction to flush a CPU cacheline, but `clflush` is strictly ordered and needlessly invalidates the cacheline, incurring a significant performance penalty [6, 76]. Also, `clflush` only sends data to the memory controller; it does not guarantee the data will reach memory. Memory barriers such as Intel's `mfence` instruction enforce order on memory operations before and after the barrier, but `mfence` only guarantees all CPUs have the same view of the memory. It does not impose any constraints on the order of data writebacks to NVMM.

Intel has proposed new instructions that fix these problems, including `clflushopt` (a more efficient version of `clflush`), `clwb` (to explicitly write back a cache line without invalidating it) and `PCOMMIT` (to force stores out to NVMM) [26, 79]. NOVA is built with these instructions in mind. In our evaluation we use a hardware NVMM emulation system that approximates the performance impacts of these instructions.

Atomicity POSIX-style file system semantics require many operations to be atomic (i.e., to execute in an "all or nothing" fashion). For example, the POSIX `rename` requires that if the operation fails, neither the file with the old name nor the file with the new name shall be changed or created [53]. Renaming a file is a metadata-only operation, but some atomic updates apply to both file system metadata and data. For instance, appending to a file atomically updates the file data and changes the file's length and modification time. Many applications rely on atomic file system operations for their own correctness.

Storage devices typically provide only rudimentary guarantees about atomicity. Disks provide atomic sector writes and processors guarantee only that 8-byte (or smaller), aligned stores are atomic. To build the more complex atomic updates that file systems require, programmers must use more complex techniques.

2.3. Building complex atomic operations

Existing file systems use a variety of techniques like journaling, shadow paging, or log-structuring to provide atomicity guarantees. These work in different ways and incur different types of overheads.

Journaling Journaling (or write-ahead logging) is widely used in journaling file systems [24, 27, 32, 71] and databases [39, 43] to ensure atomicity. A journaling system records all updates to a journal before applying them and, in case of power failure, replays the journal to restore the system to a consistent state. Journaling requires writing data twice: once to the log and once to the target location, and to improve performance journaling file systems usually only journal metadata. Recent work has proposed back pointers [17] and decoupling ordering from durability [16] to reduce the overhead of journaling.

Shadow paging Several file systems use a copy-on-write mechanism called shadow paging [20, 8, 25, 54]. Shadow paging file systems rely heavily on their tree structure to provide atomicity. Rather than modifying data in-place during a write, shadow paging writes a new copy of the affected page(s) to an empty portion of the storage device. Then, it splices the new pages into the file system tree by updating the nodes between the pages and root. The resulting cascade of updates is potentially expensive.

Log-structuring Log-structured file systems (LFSs) [55, 60] were originally designed to exploit hard disk drives' high performance on sequential accesses. LFSs buffer random writes in memory and convert them into larger, sequential writes to the disk, making the best of hard disks' strengths.

Although LFS is an elegant idea, implementing it efficiently is complex, because LFSs rely on writing sequentially to contiguous free regions of the disk. To ensure a consistent supply of such regions, LFSs constantly clean and compact the log to reclaim space occupied by stale data.

Log cleaning adds overhead and degrades the performance of LFSs [3, 61]. To reduce cleaning overhead, some LFS designs separate hot and cold data and apply different cleaning policies to each [69, 70]. SSDs also perform best under sequential workloads [9, 14], so LFS techniques have been applied to SSD file systems as well. SFS [38] classifies file blocks based on their update likelihood, and writes blocks with similar "hotness" into the same log segment to reduce cleaning overhead. F2FS [30] uses multi-head logging, writes metadata and data to separate logs, and writes new data directly to free space in dirty segments at high disk utilization to avoid frequent garbage collection.

RAMCloud [44] is a DRAM-based storage system that keeps all its data in DRAM to service reads and maintains a persistent version on hard drives. RAMCloud applies log structure to both DRAM and disk: It allocates DRAM in a log-structured way, achieving higher DRAM utilization than other memory allocators [56], and stores the back up data in logs on disk.

2.4. File systems for NVMM

Several groups have designed NVMM-based file systems that address some of the issues described in Section 2.2 by applying one or more of the techniques discussed in Section 2.3, but none meet all the requirements that modern applications place on file systems.

BPFS [20] is a shadow paging file system that provides metadata and data consistency. BPFS proposes a hardware mechanism to enforce store durability and ordering. BPFS uses short-circuit shadow paging to reduce shadow paging overheads in common cases, but certain operations that span a large portion of the file system tree (e.g., a move between directories) can still incur large overheads.

PMFS [21, 49] is a lightweight DAX file system that bypasses the block layer and file system page cache to improve performance. PMFS uses journaling for metadata updates. It performs writes in-place, so they are not atomic.

Ext4-DAX [71] extends Ext4 with DAX capabilities to directly access NVMM, and uses journaling to guarantee metadata update atomicity. The normal (non-DAX) Ext4 file system has a data-journal mode to provide data atomicity. Ext4-DAX does not support this mode, so data updates are not atomic.

SCMFS [73] utilizes the operating system's virtual memory management module and maps files to large contiguous virtual address regions, making file accesses simple and lightweight. SCMFS does not provide any consistency guarantee of metadata or data.

Aerie [66] implements the file system interface and functionality in user space to provide low-latency access to data in NVMM. It has an optimization that improves performance by relaxing POSIX semantics. Aerie journals metadata but does not support data atomicity or mmap operation.

3. NOVA Design Overview

NOVA is a log-structured, POSIX file system that builds on the strengths of LFS and adapts them to take advantage of hybrid memory systems. Because it targets a different storage technology, NOVA looks very different from conventional log-structured file systems that are built to maximize disk bandwidth.

We designed NOVA based on three observations. First, logs that support atomic updates are easy to implement correctly in NVMM, but they are not efficient for search operations (e.g., directory lookup and random-access within a file). Conversely, data structures that support fast search (e.g., tree structures) are more difficult to implement correctly and efficiently in NVMM [15, 40, 65, 75]. Second, the complexity of cleaning logs stems primarily from the need to supply contiguous free regions of storage, but this is not necessary in NVMM, because random access is cheap. Third, using a single log makes sense for disks (where there is a single disk head and improving spatial locality is paramount), but it limits concurrency. Since NVMMs support fast, highly concurrent random accesses, using multiple logs does not negatively impact performance.

Based on these observations, we made the following design decisions in NOVA.

Keep logs in NVMM and indexes in DRAM. NOVA keeps log and file data in NVMM and builds radix trees [35] in DRAM to quickly perform search operations, making the in-NVMM data structures simple and efficient. We use a radix tree because there is a mature, well-tested, widely-used implementation in the Linux kernel. The leaves of the radix tree point to entries in the log which in turn point to file data.

Give each inode its own log. Each inode in NOVA has its own log, allowing concurrent updates across files without synchronization. This structure allows for high concurrency both in file access and during recovery, since NOVA can replay multiple logs simultaneously. NOVA also guarantees that the number of valid log entries is small (on the order of the number of extents in the file), which ensures that scanning the log is fast.

Use logging and lightweight journaling for complex atomic updates. NOVA is log-structured because this provides cheaper atomic updates than journaling and shadow paging. To atomically write data to a log, NOVA first appends data to the log and then atomically updates the log tail to commit the updates, thus avoiding both the duplicate writes overhead of journaling file systems and the cascading update costs of shadow paging systems.

Some directory operations, such as a move between directories, span multiple inodes and NOVA uses journaling to atomically update multiple logs. NOVA first writes data at the end of each inode's log, and then journals the log tail updates to update them atomically. NOVA journaling is lightweight since it only involves log tails (as opposed to file data or metadata) and no POSIX file operation operates on more than four inodes.

Implement the log as a singly linked list. The locality benefits of sequential logs are less important in NVMM-based storage, so NOVA uses a linked list of 4 KB NVMM pages to hold the log and stores the next page pointer in the end of each log page.

Allowing for non-sequential log storage provides three advantages. First, allocating log space is easy since NOVA does not need to allocate large, contiguous regions for the log. Second, NOVA can perform log cleaning at fine-grained, page-size granularity. Third, reclaiming log pages that contain only stale entries requires just a few pointer assignments.

Do not log file data. The inode logs in NOVA do not contain file data. Instead, NOVA uses copy-on-write for modified pages and appends metadata about the write to the log. The metadata describe the update and point to the data pages. Section 4.4 describes file write operation in more detail.

Using copy-on-write for file data is useful for several reasons. First, it results in a shorter log, accelerating the recovery process. Second, it makes garbage collection simpler and more efficient, since NOVA never has to copy file data out of the log to reclaim a log page. Third, reclaiming stale pages and allocating new data pages are both easy, since they just require adding and removing pages from in-DRAM free lists. Fourth, since it can reclaim stale data pages immediately, NOVA can sustain performance even under heavy write loads and high NVMM utilization levels.

The next section describes the implementation of NOVA in more detail.

4. Implementing NOVA

We have implemented NOVA in the Linux kernel version 4.0. NOVA uses the existing NVMM hooks in the kernel and has passed the Linux POSIX file system test suite [50]. The source code is available on GitHub: https://github.com/NVSL/NOVA. In this section we first describe the overall file system layout and its atomicity and write ordering mechanisms. Then, we describe how NOVA performs atomic directory, file, and mmap operations. Finally we discuss garbage collection, recovery, and memory protection in NOVA.

4.1. NVMM data structures and space management

Figure 1 shows the high-level layout of NOVA data structures in a region of NVMM it manages. NOVA divides the NVMM into four parts: the superblock and recovery inode, the inode tables, the journals, and log/data pages. The superblock contains global file system information, the recovery inode stores recovery information that accelerates NOVA remount after a clean shutdown (see Section 4.7), the inode tables contain inodes, the journals provide atomicity to directory operations, and the remaining area contains NVMM log and data pages. We designed NOVA with scalability in mind: NOVA maintains an inode table, journal, and NVMM free page list at each CPU to avoid global locking and scalability bottlenecks.

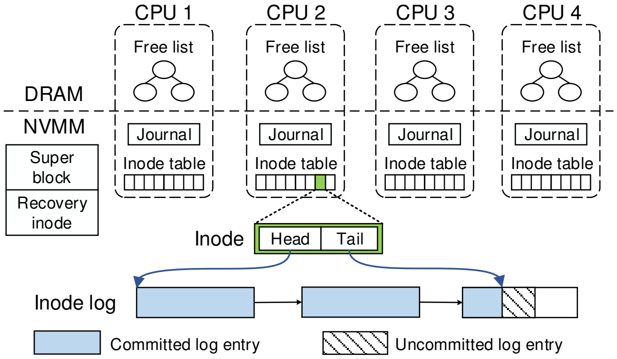


Figure 1: NOVA data structure layout. NOVA has per-CPU free lists, journals and inode tables to ensure good scalability. Each inode has a separate log consisting of a singly linked list of 4 KB log pages; the tail pointer in the inode points to the latest committed entry in the log.

Inode table NOVA initializes each inode table as a 2 MB block array of inodes. Each NOVA inode is aligned on 128-byte boundary, so that given the inode number NOVA can easily locate the target inode. NOVA assigns new inodes to each inode table in a round-robin order, so that inodes are evenly distributed among inode tables. If the inode table is full, NOVA extends it by building a linked list of 2 MB sub-tables. To reduce the inode table size, each NOVA inode contains a valid bit and NOVA reuses invalid inodes for new files and directories. Per-CPU inode tables avoid the inode allocation contention and allow for parallel scanning in failure recovery.

A NOVA inode contains pointers to the head and tail of its log. The log is a linked list of 4 KB pages, and the tail always points to the latest committed log entry. NOVA scans the log from head to tail to rebuild the DRAM data structures when the system accesses the inode for the first time.

Journal A NOVA journal is a 4 KB circular buffer and NOVA manages each journal with a <enqueue, dequeue> pointer pair. To coordinate updates that across multiple inodes, NOVA first appends log entries to each log, and then starts a transaction by appending all the affected log tails to the current CPU's journal enqueue, and updates the enqueue pointer. After propagating the updates to the target log tails, NOVA updates the dequeue equal to enqueue to commit the transaction. For a `create` operation, NOVA journals the directory's log tail pointer and new inode's valid bit. During power failure recovery, NOVA checks each journal and rolls back any updates between the journal's dequeue and enqueue. NOVA only allows one open transaction at a time on each core and per-CPU journals allow for concurrent transactions. For each directory operation, the kernel's virtual file system

(VFS) layer locks all the affected inodes, so concurrent transactions never modify the same inode.

NVMM space management To make NVMM allocation and deallocation fast, NOVA divides NVMM into pools, one per CPU, and keeps lists of free NVMM pages in DRAM. If no pages are available in the current CPU's pool, NOVA allocates pages from the largest pool, and uses per-pool locks to provide protection. This allocation scheme is similar to scalable memory allocators like Hoard [5]. To reduce the allocator size, NOVA uses a red-black tree to keep the free list sorted by address, allowing for efficient merging and providing $O(\log n)$ deallocation. To improve performance, NOVA does not store the allocator state in NVMM during operation. On a normal shutdown, it records the allocator state to the recovery inode's log and restores the allocator state by scanning the all the inodes' logs in case of a power failure.

NOVA allocates log space aggressively to avoid the need to frequently resize the log. Initially, an inode's log contains one page. When the log exhausts the available space, NOVA allocates sufficient new pages to double the log space and appends them to the log. If the log length is above a given threshold, NOVA appends a fixed number of pages each time.

4.2. Atomicity and enforcing write ordering

NOVA provides fast atomicity for metadata, data, and mmap updates using a technique that combines log structuring and journaling. This technique uses three mechanisms.

64-bit atomic updates Modern processors support 64-bit atomic writes for volatile memory and NOVA assumes that 64-bit writes to NVMM will be atomic as well. NOVA uses 64-bit in-place writes to directly modify metadata for some operations (e.g., the file's *atime* for reads) and uses them to commit updates to the log by updating the inode's log tail pointer.

Logging NOVA uses the inode's log to record operations that modify a single inode. These include operations such as `write`, `msync` and `chmod`. The logs are independent of one another.

Lightweight journaling For directory operations that require changes to multiple inodes (e.g., `create`, `unlink` and `rename`), NOVA uses lightweight journaling to provide atomicity. At any time, the data in any NOVA journal are small—no more than 64 bytes: The most complex POSIX `rename` operation involves up to four inodes, and NOVA only needs 16 bytes to journal each inode: 8 bytes for the address of the log tail pointer and 8 bytes for the value.

Enforcing write ordering NOVA relies on three write ordering rules to ensure consistency. First, it commits data and log entries to NVMM before updating the log tail. Second, it commits journal data to NVMM before propagating the updates. Third, it commits new versions of data pages to NVMM before recycling the stale versions. If NOVA is running on a system that supports `clflushopt`, `clwb` and `PCOMMIT` instructions, it uses the code in Figure 2 to enforce the write ordering.

```
new_tail = append_to_log(inode->tail, entry);
// writes back the log entry cachelines
clwb(inode->tail, entry->length);
sfence();        // orders subsequent PCOMMIT
PCOMMIT();       // commits entry to NVMM
sfence();        // orders subsequent store
inode->tail = new_tail;
```

Figure 2: Pseudocode for enforcing write ordering. NOVA commits the log entry to NVMM strictly before updating the log tail pointer. The persistency of the tail update is not shown in the figure.

First, the code appends the entry to the log. Then it flushes the affected cache lines with `clwb`. Next, it issues a `sfence` and a `PCOMMIT` instruction to force all previous updates to the NVMM controller. A second `sfence` prevents the tail update from occurring before the `PCOMMIT`. The write-back and commit of the tail update are not shown in the figure.

If the platform does not support the new instructions, NOVA uses `movntq`, a non-temporal move instruction that bypasses the CPU cache hierarchy to perform direct writes to NVMM and uses a combination of `clflush` and `sfence` to enforce the write ordering.

4.3. Directory operations

NOVA pays close attention to directory operations because they have a large impact on application performance [37, 33, 64]. NOVA includes optimizations for all the major directory operations, including `link`, `symlink` and `rename`.

NOVA directories comprise two parts: the log of the directory's inode in NVMM and a radix tree in DRAM. Figure 3 shows the relationship between these components. The directory's log holds two kinds of entries: directory entries (dentry) and inode update entries. Dentries include the name of the child file/directory, its inode number, and timestamp. NOVA uses the timestamp to atomically update the directory inode's *mtime* and *ctime* with the operation. NOVA appends a dentry to the log when it creates, deletes, or renames a file or subdirectory under that directory. A dentry for a `delete` operation has its inode number set to zero to distinguish it from a create dentry.

NOVA adds inode update entries to the directory's log to record updates to the directory's inode (e.g., for `chmod` and `chown`). These operations modify multiple fields of the inode, and the inode update entry provides atomicity.

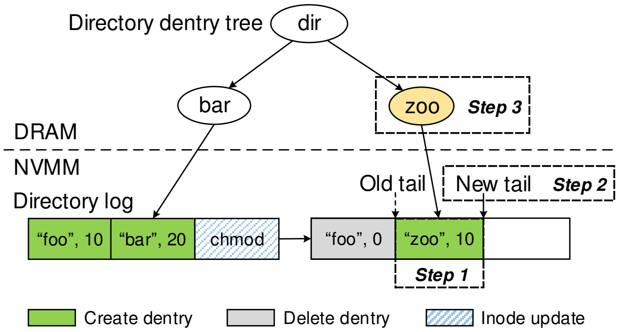


Figure 3: NOVA directory structure. Dentry is shown in <name, inode_number> format. To create a file, NOVA first appends the dentry to the directory's log (step 1), updates the log tail as part of a transaction (step 2), and updates the radix tree (step 3).

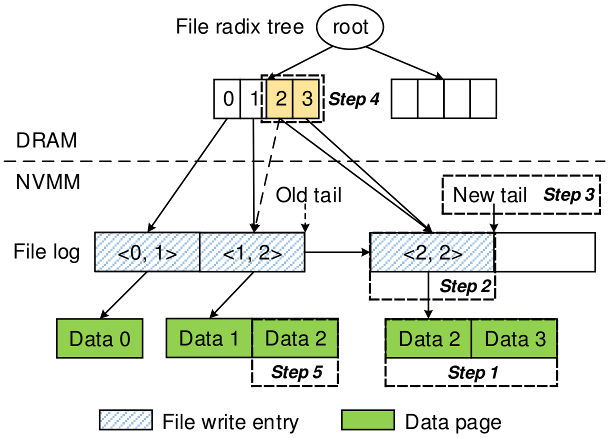


Figure 4: NOVA file structure. An 8 KB (i.e., 2-page) write to page two (<2, 2>) of a file requires five steps. NOVA first writes a copy of the data to new pages (step 1) and appends the file write entry (step 2). Then it updates the log tail (step 3) and the radix tree (step 4). Finally, NOVA returns the old version of the data to the allocator (step 5).

To speed up dentry lookups, NOVA keeps a radix tree in DRAM for each directory inode. The key is the hash value of the dentry name, and each leaf node points to the corresponding dentry in the log. The radix tree makes search efficient even for large directories. Below, we use file creation and deletion to illustrate these principles.

Creating a file Figure 3 illustrates the creation of file *zoo* in a directory that already contains file *bar*. The directory has recently undergone a chmod operation and used to contain another file, *foo*. The log entries for those operations are visible in the figure. NOVA first selects and initializes an unused inode in the inode table for *zoo*, and appends a create dentry of *zoo* to the directory's log. Then, NOVA uses the current CPU's journal to atomically update the directory's log tail and set the valid bit of the new inode. Finally NOVA adds the file to the directory's radix tree in DRAM.

Deleting a file In Linux, deleting a file requires two updates: The first decrements the link count of the file's inode, and the second removes the file from the enclosing directory. NOVA first appends a delete dentry log entry to the directory inode's log and an inode update entry to the file inode's log and then uses the journaling mechanism to atomically update both log tails. Finally it propagates the changes to the directory's radix tree in DRAM.

4.4. Atomic file operations

The NOVA file structure uses logging to provide metadata and data atomicity with low overhead, and it uses copy-on-write for file data to reduce the log size and make garbage collection simple and efficient. Figure 4 shows the structure of a NOVA file. The file inode's log records metadata changes, and each file has a radix tree in DRAM to locate data in the file by the file offset.

A file inode's log contains two kinds of log entries: inode update entries and file write entries that describe file write operations and point to data pages the write modified. File write entries also include timestamp and file size, so that write operations atomically update the file's metadata. The DRAM radix tree maps file offsets to file write entries.

If the write is large, NOVA may not be able to describe it with a single write entry. If NOVA cannot find a large enough set of contiguous pages, it breaks the write into multiple write entries and appends them all to the log to satisfy the request. To maintain atomicity, NOVA commits all the entries with a single update to the log tail pointer.

For a read operation, NOVA updates the file inode's access time with a 64-bit atomic write, locates the required page using the file's radix tree, and copies the data from NVMM to the user buffer.

Figure 4 illustrates a write operation. The notation <*file pgoff*, *num pages*> denotes the page offset and number of pages a write affects. The first two entries in the log describe two writes, <0, 1> and <1, 2>, of 4 KB and 8 KB (i.e., 1 and 2 pages), respectively. A third, 8 KB write, <2, 2>, is in flight.

To perform the <2, 2> write, NOVA fills data pages and then appends the <2, 2> entry to the file's inode log. Then NOVA atomically updates the log tail to commit the write, and updates the radix tree in DRAM, so that offset "2" points to the new entry. The NVMM page that holds the old contents of page 2 returns to the free list immediately. During the operation, a per-inode lock protects the log and the radix tree from concurrent updates. When the write system call returns, all the updates are persistent in NVMM.

4.5. Atomic mmap

DAX file systems allow applications to access NVMM directly via load and store instructions by mapping the physical NVMM file data pages into the application's address space. This *DAX-mmap* exposes the NVMM's raw performance to the applications and is likely to be a critical interface in the future.

While DAX-mmap bypasses the file system page cache and avoids paging overheads, it presents challenges for programmers. DAX-mmap provides raw NVMM so the only atomicity mechanisms available to the programmer are the 64-bit writes, fences, and cache flush instructions that the processor provides. Using these primitives to build robust non-volatile data structures is very difficult [19, 67, 34], and expecting programmers to do so will likely limit the usefulness of direct-mapped NVMM.

To address this problem, NOVA proposes a direct NVMM access model with stronger consistency called `atomic-mmap`. When an application uses `atomic-mmap` to map a file into its address space, NOVA allocates *replica pages* from NVMM, copies the file data to the replica pages, and then maps the replicas into the address space. When the application calls `msync` on the replica pages, NOVA handles it as a `write` request described in the previous section, uses `movntq` operation to copy the data from replica pages to data pages directly, and commits the changes atomically.

Since NOVA uses copy-on-write for file data and reclaims stale data pages immediately, it does not support DAX-mmap. `Atomic-mmap` has higher overhead than DAX-mmap but provides stronger consistency guarantee. The normal DRAM mmap is not atomic because the operating system might eagerly write back a subset of dirty pages to the file system, leaving the file data inconsistent in event of a system failure [45]. NOVA could support atomic mmap in DRAM by preventing the operating system from flushing dirty pages, but we leave this feature as future work.

4.6. Garbage collection

NOVA's logs are linked lists and contain only metadata, making garbage collection simple and efficient. This structure also frees NOVA from the need to constantly move data to maintain a supply of contiguous free regions.

NOVA handles garbage collection for stale data pages and stale log entries separately. NOVA collects stale data pages immediately during `write` operations (see Section 4.4).

Cleaning inode logs is more complex. A log entry is dead in NOVA if it is not the last entry in the log (because the last entry records the inode's latest *ctime*) and any of the following conditions is met:

- A file write entry is dead, if it does not refer to valid data pages.

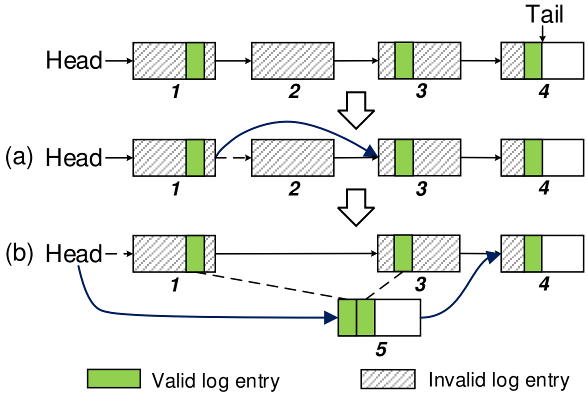


Figure 5: NOVA log cleaning. The linked list structure of log provides simple and efficient garbage collection. Fast GC reclaims invalid log pages by deleting them from the linked list (a), while thorough GC copies live log entries to a new version of the log (b).

- An inode update that modifies metadata (e.g., *mode* or *mtime*) is dead, if a later inode update modifies the same piece of metadata.
- A dentry update is dead, if it is marked invalid.

NOVA mark dentries invalid in certain cases. For instance, file creation adds a create dentry to the log. Deleting the file adds a delete dentry, and it also marks the create dentry as invalid. (If the NOVA garbage collector reclaimed the delete dentry but left the create dentry, the file would seem to reappear.)

These rules determine which log entries are alive and dead, and NOVA uses two different garbage collection (GC) techniques to reclaim dead entries.

Fast GC Fast GC emphasizes speed over thoroughness and it does not require any copying. NOVA uses it to quickly reclaim space when it extends an inode's log. If all the entries in a log page are dead, fast GC reclaims it by deleting the page from the log's linked list. Figure 5(a) shows an example of fast log garbage collection. Originally the log has four pages and page 2 contains only dead log entries. NOVA atomically updates the next page pointer of page 1 to point to page 3 and frees page 2.

Thorough GC During the fast GC log scan, NOVA tallies the space that live log entries occupy. If the live entries account for less than 50% of the log space, NOVA applies thorough GC after fast GC finishes, copies live entries into a new, compacted version of the log, updates the DRAM data structure to point to the new log, then atomically replaces the old log with the new one, and finally reclaims the old log.

Figure 5(b) illustrates thorough GC after fast GC is complete. NOVA allocates a new log page 5, and copies valid log entries in page 1 and 3 into it. Then, NOVA links page 5 to page 4 to create a new log and replace the old one. NOVA does not copy the live entries in page 4 to avoid updating the log tail, so that NOVA can atomically replace the old log by updating the log head pointer.

4.7. Shutdown and Recovery

When NOVA mounts the file system, it reconstructs the in-DRAM data structures it needs. Since applications may access only a portion of the inodes while the file system is running, NOVA adopts a policy called *lazy rebuild* to reduce the recovery time: It postpones rebuilding the radix tree and the inode until the system accesses the inode for the first time. This policy accelerates the recovery process and reduces DRAM consumption. As a result, during remount NOVA only needs to reconstruct the NVMM free page lists. The algorithm NOVA uses to recover the free lists is different for "clean" shutdowns than for system failures.

Recovery after a normal shutdown On a clean unmount, NOVA stores the NVMM page allocator state in the recovery inode's log and restores the allocator during the subsequent remount. Since NOVA does not scan any inode logs in this case, the recovery process is very fast: Our measurement shows that NOVA can remount a 50 GB file system in 1.2 milliseconds.

Recovery after a failure In case of a unclean dismount (e.g., system crash), NOVA must rebuild the NVMM allocator information by scanning the inode logs. NOVA log scanning is fast because of two design decisions. First, per-CPU inode tables and per-inode logs allow for vast parallelism in log recovery. Second, since the logs do not contain data pages, they tend to be short. The number of live log entries in an inode log is roughly the number of extents in the file. As a result, NOVA only needs to scan a small fraction of the NVMM during recovery. The NOVA failure recovery consists of two steps:

First, NOVA checks each journal and rolls back any uncommitted transactions to restore the file system to a consistent state.

Second, NOVA starts a recovery thread on each CPU and scans the inode tables in parallel, performing log scanning for every valid inode in the inode table. NOVA use different recovery mechanisms for directory inodes and file inodes: For a directory inode, NOVA scans the log's linked list to enumerate the pages it occupies, but it does not inspect the log's contents. For a file inode, NOVA reads the write entries in the log to enumerate the data pages.

During the recovery scan NOVA builds a bitmap of occupied pages, and rebuilds the allocator based on the result. After this process completes, the file system is ready to accept new requests.

4.8. NVMM Protection

Since the kernel maps NVMM into its address space during NOVA mount, the NVMM is susceptible to corruption by errant stores from the kernel. To protect the file system and prevent permanent corruption of the NVMM from stray writes, NOVA must make sure it is the only system software that accesses the NVMM.

NOVA uses the same protection mechanism that PMFS does. Upon mount, the whole NVMM region is mapped as read-only. Whenever NOVA needs to write to the NVMM pages, it opens a write window by disabling the processor's write protect control (CR0.WP). When CR0.WP is clear, kernel software running on ring 0 can write to pages marked read-only in the kernel address space. After the NVMM write completes, NOVA resets CR0.WP to close the write window. CR0.WP is not saved across interrupts so NOVA disables local interrupts during the write window. Opening and closing the write window does not require modifying the page tables or the TLB, so it is inexpensive.

5. Evaluation

In this section we evaluate the performance of NOVA and answer the following questions:

- How does NOVA perform against state-of-the-art file systems built for disks, SSDs, and NVMM?
- What kind of operations benefit most from NOVA?
- How do underlying NVMM characteristics affect NOVA performance?
- How efficient is NOVA garbage collection compared to other approaches?
- How expensive is NOVA recovery?

We first describe the experimental setup and then evaluate NOVA with micro- and macro-benchmarks.

5.1. Experimental setup

To emulate different types of NVMM and study their effects on NVMM file systems, we use the Intel Persistent Memory Emulation Platform (PMEP) [21]. PMEP is a dual-socket Intel Xeon processor-based platform with special CPU microcode and firmware. The processors on PMEP run at 2.6 GHz with 8 cores and 4 DDR3 channels. The BIOS marks the DRAM memory on channels 2 and 3 as emulated NVMM. PMEP supports configurable latencies and bandwidth for the emulated NVMM, allowing us to explore NOVA's performance on a variety of future memory technologies. PMEP emulates `clflushopt`, `clwb`, and `PCOMMIT` instructions with processor microcode.

In our tests we configure the PMEP with 32 GB of DRAM and 64 GB of NVMM. To emulate different NVMM technologies, we choose two configurations for PMEP's mem-

NVMM	Read latency	Write bandwidth	clwb latency	PCOMMIT latency
STT-RAM	100 ns	Full DRAM	40 ns	200 ns
PCM	300 ns	1/8 DRAM	40 ns	500 ns

Table 1: NVMM emulation characteristics. STT-RAM emulates fast NVMs that have access latency and bandwidth close to DRAM, and PCM emulates NVMs that are slower than DRAM.

ory emulation system (Table 1): For STT-RAM we use the same read latency and bandwidth as DRAM, and configure PCOMMIT to take 200 ns; For PCM we use 300 ns for the read latency and reduce the write bandwidth to 1/8th of DRAM, and PCOMMIT takes 500 ns.

We evaluate NOVA on Linux kernel 4.0 against seven file systems: Two of these, PMFS and Ext4-DAX are the only available open source NVMM file systems that we know of. Both of them journal metadata and perform in-place updates for file data. Two others, NILFS2 and F2FS are log-structured file systems designed for HDD and flash-based storage, respectively. We also compare to Ext4 in default mode (Ext4) and in data journal mode (Ext4-data) which provides data atomicity. Finally, we compare to Btrfs [54], a state-of-the-art copy-on-write Linux file system. Except for Ext4-DAX and Ext4-data, all the file systems are mounted with default options. Btrfs and Ext4-data are the only two file systems in the group that provide the same, strong consistency guarantees as NOVA.

PMFS and NOVA manage NVMM directly and do not require a block device interface. For the other file systems, we use the Intel persistent memory driver [48] to emulate NVMM-based ramdisk-like device. The driver does not provide any protection from stray kernel stores, so we disable the CR0.WP protection in PMFS and NOVA in the tests to make the comparison fair. We add clwb and PCOMMIT instructions to flush data where necessary in each file system.

5.2. Microbenchmarks

We use a single-thread micro-benchmark to evaluate the latency of basic file system operations. The benchmark creates 10,000 files, makes sixteen 4 KB appends to each file, calls fsync to persist the files, and finally deletes them.

Figures 6(a) and 6(b) show the results on STT-RAM and PCM, respectively. The latency of fsync is amortized across the append operations. NOVA provides the lowest latency for each operation, outperforms other file systems by between 35% and 17×, and improves the append performance by 7.3× and 6.7× compared to Ext4-data and Btrfs respectively. PMFS is closest to NOVA in terms of append and delete performance. NILFS2 performs poorly on create operations, suggesting that naively using log-structured, disk-oriented file systems on NVMM is unwise.

Workload	Average file size	I/O size (r/w)	Threads	R/W ratio	# of files Small/Large
Fileserver	128 KB	16 KB/16 KB	50	1:2	100K/400K
Webproxy	32 KB	1 MB/16 KB	50	5:1	100K/1M
Webserver	64 KB	1 MB/8 KB	50	10:1	100K/500K
Varmail	32 KB	1 MB/16 KB	50	1:1	100K/1M

Table 2: Filebench workload characteristics. The selected four workloads have different read/write ratios and access patterns.

NOVA is more sensitive to NVMM performance than the other file systems because NOVA's software overheads are lower, and so overall performance more directly reflects the underlying memory performance. Figure 6(c) shows the latency breakdown of NOVA file operations on STT-RAM and PCM. For create and append operations, NOVA only accounts for 21%–28% of the total latency. On PCM the NOVA delete latency increases by 76% because NOVA reads the inode log to free data and log blocks and PCM has higher read latency. For the create operation, the VFS layer accounts for 49% of the latency on average. The memory copy from the user buffer to NVMM consumes 51% of the append execution time on STT-RAM, suggesting that the POSIX interface may be the performance bottleneck on high speed memory devices.

5.3. Macrobenchmarks

We select four Filebench [23] workloads—fileserver, webproxy, webserver and varmail—to evaluate the application-level performance of NOVA. Table 2 summarizes the characteristics of the workloads. For each workload we test two dataset sizes by changing the number of files. The *small* dataset will fit entirely in DRAM, allowing file systems that use the DRAM page cache to cache the entire dataset. The *large* dataset is too large to fit in DRAM, so the page cache is less useful. We run each test five times and report the average. Figure 7 shows the Filebench throughput with different NVMM technologies and data set sizes.

In the fileserver workload, NOVA outperforms other file systems by between 1.8× and 16.6× on STT-RAM, and between 22% and 9.1× on PCM for the large dataset. NOVA outperforms Ext4-data by 11.4× and Btrfs by 13.5× on STT-RAM, while providing the same consistency guarantees. NOVA on STT-RAM delivers twice the throughput compared to PCM, because of PCM's lower write bandwidth. PMFS performance drops by 80% between the small and large datasets, indicating its poor scalability.

Webproxy is a read-intensive workload. For the small dataset, NOVA performs similarly to Ext4 and Ext4-DAX, and 2.1× faster than Ext4-data. For the large workload, NOVA performs between 36% and 53% better than F2FS and Ext4-DAX. PMFS performs directory lookup by linearly searching the directory entries, and NILFS2's directory lock

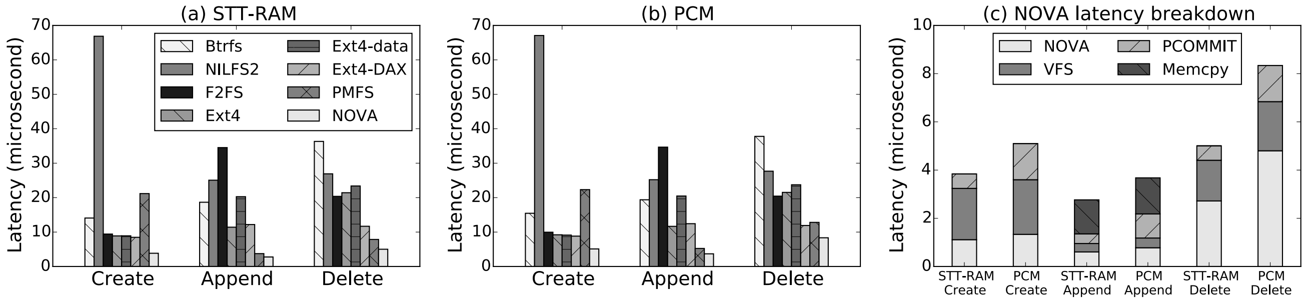


Figure 6: File system operation latency on different NVMM configurations. The single-thread benchmark performs create, append and delete operations on a large number of files.

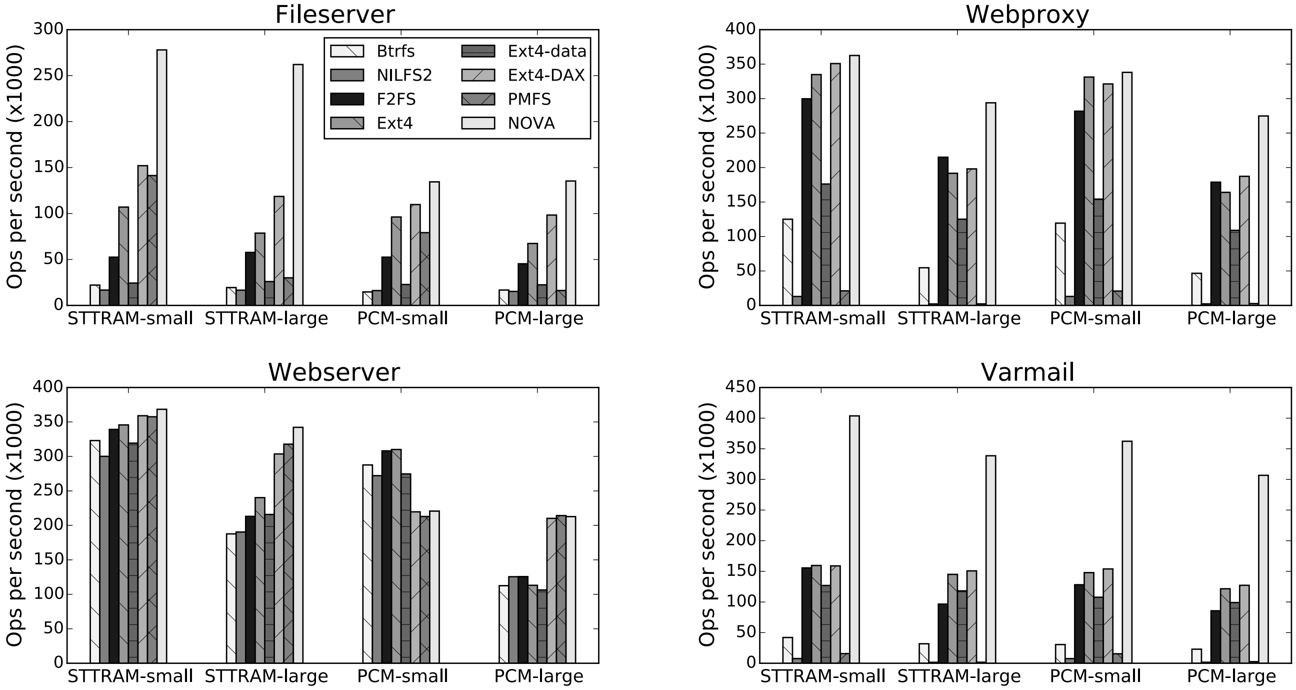


Figure 7: Filebench throughput with different file system patterns and dataset sizes on STT-RAM and PCM. Each workload has two dataset sizes so that the small one can fit in DRAM entirely while the large one cannot. The standard deviation is less than 5% of the value.

design is not scalable [57], so their performance suffers since webproxy puts all the test files in one large directory.

Webserver is a read-dominated workload and does not involve any directory operations. As a result, non-DAX file systems benefit significantly from the DRAM page cache and the workload size has a large impact on performance. Since STT-RAM has the same latency as DRAM, small workload performance is roughly the same for all the file systems with NOVA enjoying a small advantage. On the large data set, NOVA performs 10% better on average than Ext4-DAX and PMFS, and 63% better on average than non-DAX file systems. On PCM, NOVA's performance is about the same as the other DAX file systems. For the small dataset, non-DAX file systems are 33% faster on average due to DRAM caching. However, for the large dataset, NOVA's performance remains stable while non-DAX performance drops by 60%.

Varmail emulates an email server with a large number of small files and involves both `read` and `write` operations. NOVA outperforms Btrfs by 11.1× and Ext4-data by 3.1× on average, and outperforms the other file systems by between 2.2× and 216×, demonstrating its capabilities in write-intensive workloads and its good scalability with large directories. NILFS2 and PMFS still suffer from poor directory operation performance.

Duration	10s	30s	120s	600s	3600s
NILFS2	Fail	Fail	Fail	Fail	Fail
F2FS	37,979	23,193	18,240	Fail	Fail
NOVA	222,337	222,229	220,158	209,454	205,347
# GC pages					
Fast	0	255	17,385	159,406	1,170,611
Thorough	102	2,120	9,633	27,292	72,727

Table 3: Performance of a full file system. The test runs a 30 GB fileserver workload under 95% NVMM utilization with different durations, and reports the results in operations per second. The bottom three rows show the number of pages that NOVA garbage collector reclaimed in the test.

Overall, NOVA achieves the best performance in almost all cases and provides data consistency guarantees that are as strong or stronger than the other file systems. The performance advantages of NOVA are largest on write-intensive workloads with large number of files.

5.4. Garbage collection efficiency

NOVA resolves the issue that many LFSs suffer from, i.e. they have performance problems under heavy write loads, especially when the file system is nearly full. NOVA reduces the log cleaning overhead by reclaiming stale data pages immediately, keeping log sizes small, and making garbage collection of those logs efficient.

To evaluate the efficiency of NOVA garbage collection when NVMM is scarce, we run a 30 GB write-intensive fileserver workload under 95% NVMM utilization for different durations, and compare with the other log-structured file systems, NILFS2 and F2FS. We run the test with PMEP configured to emulate STT-RAM.

Table 3 shows the result. NILFS2 could not finish the 10-second test due to garbage collection inefficiencies. F2FS fails after running for 158 seconds, and the throughput drops by 52% between the 10s and 120s tests due to log cleaning overhead. In contrast, NOVA outperforms F2FS by 5.8× and successfully runs for the full hour. NOVA's throughput also remains stable, dropping by less than 8% between the 10s and one-hour tests.

The bottom half of Table 3 shows the number of pages that NOVA garbage collector reclaimed. On the 30s test fast GC reclaims 11% of the stale log pages. With running time rises, fast GC becomes more efficient and is responsible for 94% of reclaimed pages in the one-hour test. The result shows that in long-term running, the simple and low-overhead fast GC is efficient enough to reclaim the majority of stale log pages.

5.5. Recovery overhead

NOVA uses DRAM to maintain the NVMM free page lists that it must rebuild when it mounts a file system. NOVA accelerates the recovery by rebuilding inode information lazily, keeping the logs short, and performing log scanning in parallel.

Dataset	File size	Number of files	Dataset size	I/O size
Videoserver	128 MB	400	50 GB	1 MB
Fileserver	1 MB	50,000	50 GB	64 KB
Mailserver	128 KB	400,000	50 GB	16 KB

Table 4: Recovery workload characteristics. The number of files and typical I/O size both affect NOVA's recovery performance.

Dataset	Videoserver	Fileserver	Mailserver
STTRAM-normal	156 μs	313 μs	918 μs
PCM-normal	311 μs	660 μs	1197 μs
STTRAM-failure	37 ms	39 ms	72 ms
PCM-failure	43 ms	50 ms	116 ms

Table 5: NOVA recovery time on different scenarios. NOVA is able to recover 50 GB data in 116ms in case of power failure.

To measure the recovery overhead, we use the three workloads in Table 4. Each workload represents a different use case for the file systems: Videoserver contains a few large files accessed with large-size requests, mailserver includes a large number of small files and the request size is small, fileserver is in between. For each workload, we measure the cost of mounting after a normal shutdown and after a power failure.

Table 5 summarizes the results. With a normal shutdown, NOVA recovers the file system in 1.2 ms, as NOVA does not need to scan the inode logs. After a power failure, NOVA recovery time increases with the number of inodes (because the number of logs increases) and as the I/O operations that created the files become smaller (because file logs become longer as files become fragmented). Recovery runs faster on STT-RAM than on PCM because NOVA reads the logs to reconstruct the NVMM free page lists, and PCM has higher read latency than STT-RAM. On both PCM and STT-RAM, NOVA is able to recover 50 GB data in 116ms, achieving failure recovery bandwidth higher than 400 GB/s.

6. Conclusion

We have implemented and described NOVA, a log-structured file system designed for hybrid volatile/non-volatile main memories. NOVA extends ideas of LFS to leverage NVMM, yielding a simpler, high-performance file system that supports fast and efficient garbage collection and quick recovery from system failures. Our measurements show that NOVA outperforms existing NVMM file systems by a wide margin on a wide range of applications while providing stronger consistency and atomicity guarantees.

Acknowledgments

This work was supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. We would like to thank John Ousterhout, Niraj Tolia, Isabella Furth, and the anonymous reviewers for their insightful comments and suggestions. We are also thankful to Subramanya R. Dulloor from Intel for his support and hardware access.

References

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

[2] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A protoype phase change memory storage array. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.80 edition, May 2014.

[4] J. Arulraj, A. Pavlo, and S. R. Dulloor. 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*, SIGMOD '15, pages 707–722, New York, NY, USA, 2015. ACM.

[5] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, New York, NY, USA, 2000. ACM.

[6] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Implications of CPU Caching on Byte-addressable Non-volatile Memory Programming. Technical report, HP Technical Report HPL-2012-236, 2012.

[7] M. S. Bhaskaran, J. Xu, and S. Swanson. Bankshot: Caching Slow Storage in Fast Non-volatile Memory. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, pages 1:1–1:9, New York, NY, USA, 2013. ACM.

[8] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems, 2007.

[9] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. *arXiv preprint arXiv:0909.1780*, 2009.

[10] M. J. Breitwisch. Phase change memory. *Interconnect Technology Conference, 2008. IITC 2008. International*, pages 219–221, June 2008.

[11] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories. In *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 43, pages 385–395, New York, NY, USA, 2010. ACM.

[12] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM.

[13] A. M. Caulfield and S. Swanson. QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks. In *ISCA '13: Proceedings of the 40th Annual International Symposium on Computer architecture*, 2013.

[14] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):181–192, 2009.

[15] S. Chen and Q. Jin. Persistent B$^+$-trees in Non-volatile Main Memory. *Proc. VLDB Endow.*, 8(7):786–797, Feb. 2015.

[16] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[17] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[18] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M. G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y.-J. Lee, Q. Wang, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y.-T. Lee, J. Yoo, and G. Jeong. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 46–48, Feb 2012.

[19] 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 Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 105–118, New York, NY, USA, 2011. ACM.

[20] 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 ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[21] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software

for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[22] 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.

[23] Filebench file system benchmark. http://sourceforge.net/projects/filebench.

[24] R. HAGMANN. Reimplementing the cedar file system using logging and group commit. In *Proc. 11th ACM Symposium on Operating System Principles Austin, TX*, pages 155–162, 1987.

[25] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter*, pages 235–246, 1994.

[26] Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[27] S. G. International. XFS: A High-performance Journaling Filesystem. http://oss.sgi.com/projects/xfs.

[28] T. Kawahara. Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing. *Design & Test of Computers, IEEE*, 28(1):52–63, Jan 2011.

[29] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 2–13, New York, NY, USA, 2009. ACM.

[30] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies*, FAST '15, pages 273–286, Santa Clara, CA, Feb. 2015. USENIX Association.

[31] E. Lee, H. Bahn, and S. H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies*, FAST '13, pages 73–80, San Jose, CA, 2013. USENIX.

[32] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. *SIGOPS Oper. Syst. Rev.*, 30(5):84–92, Sept. 1996.

[33] P. H. Lensing, T. Cortes, and A. Brinkmann. Direct Lookup and Hash-based Metadata Placement for Local File Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 5:1–5:11, New York, NY, USA, 2013. ACM.

[34] Persistent Memory Programming. http://pmem.io.

[35] Trees I: Radix trees. https://lwn.net/Articles/175432/.

[36] D. E. Lowell and P. M. Chen. Free Transactions with Rio Vista. In *SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 92–101, New York, NY, USA, 1997. ACM.

[37] Y. Lu, J. Shu, and W. Wang. ReconFS: A Reconstructable File System on Flash Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST' 14, pages 75–88, Berkeley, CA, USA, 2014. USENIX Association.

[38] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, page 12, 2012.

[39] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[40] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. 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.

[41] D. Narayanan and O. Hodson. Whole-system Persistence with Non-volatile Memories. In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. ACM, March 2012.

[42] 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.

[43] Berkeley DB. http://www.oracle.com/technology/products/berkeley-db/index.html.

[44] 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 Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[45] S. Park, T. Kelly, and K. Shen. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 225–238, New York, NY, USA, 2013. ACM.

[46] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[47] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Craft-

ing Crash-Consistent Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 433–448, Broomfield, CO, Oct. 2014. USENIX Association.

[48] PMEM: the persistent memory driver + ext4 direct access (DAX). https://github.com/01org/prd.

[49] Persistent Memory File System. https://github.com/linux-pmfs/pmfs.

[50] Linux POSIX file system test suite. https://lwn.net/Articles/276617/.

[51] 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*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[52] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. L. Lung, and C. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, July 2008.

[53] POSIX 1003.1 - man page for rename. http://www.unix.com/man-page/POSIX/3posix/rename/.

[54] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[55] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[56] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST '14, pages 1–16, Santa Clara, CA, 2014. USENIX.

[57] R. Santana, R. Rangaswami, V. Tarasov, and D. Hildebrand. A Fast and Slippery Slope for File Systems. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pages 5:1–5:8, New York, NY, USA, 2015. ACM.

[58] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In *SOSP '93: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 146–160, New York, NY, USA, 1993. ACM.

[59] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti. An Empirical Study of File Systems on NVM. In *Proceedings of the 2015 IEEE Symposium on Mass Storage Systems and Technologies (MSST'15)*, 2015.

[60] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 3–3. USENIX Association, 1993.

[61] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File System Logging Versus Clustering: A Performance Comparison. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 21–21, Berkeley, CA, USA, 1995. USENIX Association.

[62] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[63] K. Suzuki and S. Swanson. The Non-Volatile Memory Technology Database (NVMDB). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, May 2015. http://nvmdb.ucsd.edu.

[64] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. E. Porter. How to Get More Value from Your File System Directory Cache. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 441–456, New York, NY, USA, 2015. ACM.

[65] S. Venkataraman, N. Tolia, P. Ranganathan, and R. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST '11, San Jose, CA, USA, February 2011.

[66] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[67] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS '11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2011. ACM.

[68] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. L. Moal, T. Bunker, J. Xu, S. Swanson, and Z. Bandić. DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST '14, pages 309–315, Santa Clara, CA, 2014. USENIX.

[69] J. Wang and Y. Hu. WOLF-A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, pages 47–60, Monterey, CA, 2002. USENIX.

[70] W. Wang, Y. Zhao, and R. Bunt. HyLog: A High Performance Approach to Managing Disk Layout. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, volume 4 of *FAST '04*, pages 145–158, San Francisco, CA, 2004. USENIX.

[71] M. Wilcox. Add support for NV-DIMMs to ext4. https://lwn.net/Articles/613384/.

[72] M. Wu and W. Zwaenepoel. eNVy: A Non-volatile, Main

Memory Storage System. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-VI, pages 86–97, New York, NY, USA, 1994. ACM.

[73] X. Wu and A. L. N. 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, New York, NY, USA, 2011. ACM.

[74] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, pages 3–3, Berkeley, CA, USA, 2012. USENIX.

[75] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies*, FAST '15, pages 167–181, Santa Clara, CA, Feb. 2015. USENIX Association.

[76] Y. Zhang and S. Swanson. A Study of Application Performance with Non-Volatile Main Memory. In *Proceedings of the 2015 IEEE Symposium on Mass Storage Systems and Technologies (MSST'15)*, 2015.

[77] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. 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-46, pages 421–432, New York, NY, USA, 2013. ACM.

[78] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[79] R. Zwisler. Add support for new persistent memory instructions. https://lwn.net/Articles/619851/.

Application-Managed Flash

Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim* and Arvind

Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology
Department of Computer Science and Engineering, Seoul National University

Abstract

In flash storage, an FTL is a complex piece of code that resides completely inside the storage device and is provided by the manufacturer. Its principal virtue is providing interoperability with conventional HDDs. However, this virtue is also its biggest impediment in reaching the full performance of the underlying flash storage. We propose to refactor the flash storage architecture so that it relies on a new block I/O interface which does not permit overwriting of data without intervening erasures.We demonstrate how high-level applications, in particular file systems, can deal with this restriction efficiently by employing *append-only* segments. This refactoring dramatically reduces flash management overhead and improves performance of applications, such as file systems and databases, by permitting them to directly manage flash storage. Our experiments on a machine with the new block I/O interface show that DRAM in the flash controller is reduced by 128X and the performance of the file system improves by 80% over conventional SSDs.

1 Introduction

NAND flash SSDs have become the preferred storage device in both consumer electronics and datacenters. Flash has superior random access characteristics to speed up many applications and consumes less power than HDDs. Thanks to Moore's law and advanced manufacturing technologies like 3D NAND [27], the use of flash-based devices is likely to keep rising over the next decade.

SSDs employ a flash translation layer (FTL) to provide an I/O abstraction of a generic block device so that HDDs can be replaced by SSDs without the software being aware of flash characteristics. An FTL is a complex piece of software because it must manage the overwriting restrictions, wear- leveling and bad-block management.

Implementing these management tasks requires significant hardware resources in the flash controller. In particular, tasks like address remapping and garbage collection require large amounts of DRAM and powerful CPUs (e.g., a 1 GHz quad-core CPU with 1 GB DRAM [19, 48, 45]). The FTL makes important decisions affecting storage performance and lifetime, without any awareness of the high-level application, and consequently the resulting performance is often suboptimal [15, 28, 9, 17]. Moreover, the FTL works as a black box – its inner-workings are hidden behind a block I/O layer, which makes the behavior of flash storage unpredictable for higher-level applications, for example, unexpected invocation of garbage collection [14] and swap-in/out of mapping entries [44, 23].

Another serious drawback of the FTL approach is the duplication of functionality between the FTL and the host. Many host applications already manage underlying storage to avoid in-place updates for several reasons such as better performance, efficient versioning, data consistency and so on. Log-structured or copy-on-write file systems always append new data to the device, mostly avoiding in-place updates [47, 20, 46, 31, 33, 50]. Similar log-structured systems are used in the database community [49, 1, 5]. The LSM-Tree is also a well known data structure based on a log-structured approach [42, 12, 2, 6]. Since the FTL is not aware of this avoidance of overwriting, it employs its own log-structured technique to manage flash. Running log-structured applications on top of a log-structured FTL wastes hardware resource and incurs extra I/Os. This double logging problem is also reported by empirical studies conducted by industry [53].

In this paper, we present a different approach to managing flash storage, which is called an **A**pplication-**M**anaged **F**lash (AMF). As its name implies, AMF moves the intelligence of flash management from the device to applications, which can be file systems, databases and user applications, leaving only essential management parts on the device side. For various applications to easily use AMF, we define a new block I/O interface which does not support overwrites of data unless they were explicitly deallocated (i.e., an attempt to overwrite data without a proper deallocation generates an error). This dramatically simplifies the management burden inside the device because fine-grained remapping and garbage collection do not have to be done in the device. The application using the flash device is completely responsible for avoiding in-place updates and issuing trim commands to indicate that the data has been deallocated and the device can erase and reuse the associated flash blocks. This direct flash management by applications has several advantages. For example, it can (i) efficiently schedule both regular and cleaning I/Os (e.g., copying and compacting) based on the states of the processes; (ii) accurately separate hot and cold data according to its properties (e.g., metadata versus data); and (iii) directly map objects (e.g., files) to physical locations without maintaining a huge mapping table.

In AMF, the device responsibility is reduced to providing error-free storage accesses and efficient parallelism support to exploit multiple storage chips on buses or channels. The device also keeps track of bad blocks and performs wear-leveling. It is preferable to do these operations in the device because they depend upon the specific circuit designs and manufacturing process of the device. Understandably, device manufacturers are reluctant to disclose such details. In our system, management tasks in the device are performed at the block granularity as opposed to the page granularity, therefore mapping tables required are considerably smaller.

One may think that avoiding in-place updates places too much burden on applications or users. However, this is not the case because many applications that use HDDs already employ append-only strategies as we pointed out earlier, and these are our target applications. For such applications, the only additional burden in using our interface is to avoid rare in-place updates. Moreover, forcing host applications to write data sequentially is not an extreme constraint either. For example, shingled magnetic recording (SMR) HDDs have already adopted a similar approach for the management of overlapped sectors [11].

To demonstrate the advantages of AMF, we used an open FPGA-based flash platform, BlueDBM [25, 36], as our testbed which provides error-free accesses to raw flash chips. We implemented a new lightweight FTL called an Application-managed FTL (AFTL) to support our block I/O interface. For our case study with applications, we have selected a file system because it is the most common application to access flash storage. We have implemented a new Application-managed Log-structured File-System (ALFS). The architecture of ALFS is exactly the same as the conventional LFS, except that it appends the metadata as opposed to updating it in-place. It should be noted that applying AMF is not limited to a file system only. Many real world applications can benefit from AMF. For example, key-value stores based on LSM-Trees (e.g., LevelDB [12] and RocksDB [2]), logical volume managers combined with CoW file systems (e.g., WAFL [20] and Btrfs [46]) and log-structured databases (e.g., RethinkDB [1]) are candidate applications for AMF.

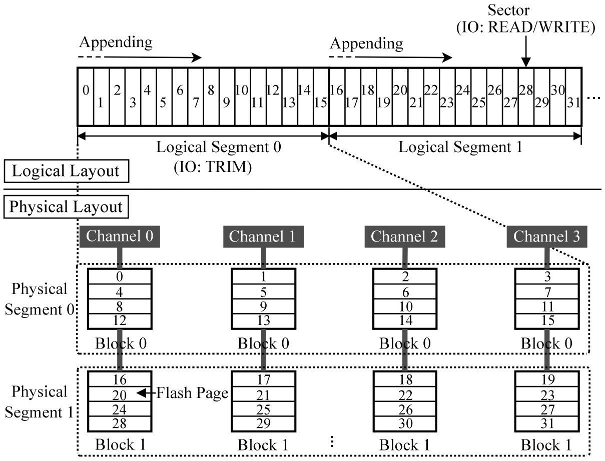


Figure 1: An AMF block I/O abstraction: It shows two logical segments (logical segments 0 and 1) and two corresponding physical ones on the device side (physical segments 0 and 1). A logical segment is composed of 16 sectors which are *statically* mapped to flash pages. A physical segment is organized with four flash blocks belonging to four channels and one way.

Our experiments show that AMF improves I/O performance and storage lifetime by up to 80% and 38%, respectively, over file systems implemented on conventional FTLs. The DRAM requirement for the FTL was reduced by a factor of 128 because of the new interface while the additional host-side resources (DRAM and CPU cycles) required by AMF were minor.

This paper is organized as follows: Section 2 explains our new block I/O interface. Sections 3 and 4 describe AMF. Section 5 evaluates AMF with various benchmarks. Section 6 reviews related work. Section 7 concludes with a summary and future directions.

2 AMF Block I/O Interface

Figure 1 depicts the block I/O abstraction of AMF, showing both logical and physical layouts. The block I/O interface of AMF is based on conventional block I/O – it exposes a linear array of fixed size blocks or sectors (e.g., 4 KB), which are accessed by three I/O primitives, `READ`, `WRITE` and `TRIM`. To distinguish a logical block from a flash block, we call it a *sector* in the remainder of this paper. Continuous sectors are grouped into a larger extent (e.g., several MB), called a *segment*.

A segment is allocated when the first write is performed and its size grows implicitly as more writes are performed. A segment is deallocated by issuing a TRIM command. Therefore, TRIM is always conducted in the unit of a segment. A sector of a segment can be read once it has been written. However, a sector can be written only once; an overwrite generates an error. To avoid this overwrite problem, the host software should write the sectors of a segment in an append-only manner, starting from the lowest sector address. This sector can be reused after the segment it belongs to has been deallocated by TRIM.

A segment exposed to upper layers is called a *logical segment*, while its corresponding physical form is called a *physical segment*. Segmentation is a well known concept in many systems. In particular, with log-structured systems, a logical segment is used as the unit of free space allocation, where new data is *sequentially* appended and free space is reclaimed later. A physical segment on the storage device side is optimized for such a sequential access by software. In Figure 1, a physical segment is composed of a group of blocks spread among different channels and ways, and sectors within a logical segment are statically mapped to flash pages within a physical one. This mapping ensures the maximum bandwidth of the device when data is read or written sequentially. It also provides predictable performance unaffected by the firmware's behavior.

Our block I/O interface expects the device controller to take the responsibility for managing bad-blocks and wear-leveling so that all the segments seen by upper layers are error-free. There are two main reasons for our decision. First, it makes the development of systems and applications easier since bad-block management and wear-leveling are relatively simple to implement at the device level and do not require significant resources. Second, the lifetime of NAND devices can be managed more effectively at lower levels where device-level information is available. Besides P/E cycles, the lifetime of NAND devices are affected by factors such as recovery effects [13] and bit error rates [43]; SSD vendors take these factors into consideration in wear-leveling and bad-block management. This information is proprietary and confidential – for example, some vendors do not reveal even P/E cycles on datasheets. Hiding these vendor and device-specific issues inside the flash controller also makes the host software vendor independent.

Compatibility Issue: Our block I/O interface maintains good compatibility with existing block I/O subsystems – the same set of I/O primitives with fixed size sectors (i.e., READ, WRITE and TRIM). The only new restrictions introduced by the AMF block I/O interface are (i) non-rewritable sectors before being trimmed, (ii) a linear array of sectors grouped to form a segment and (iii) the unit of a TRIM operation. Note that a segment size is easily shared by both applications and devices through interfaces like S.M.A.R.T and procfs. In our Linux implementation, for example, the existing block I/O layer is not changed at all. This allows us to convert existing software to run on AMF in an easy manner.

The architecture of AFTL is similar to block or segment-level FTLs and requires minimal functions for flash management, thus SSD vendors can easily build AMF devices by removing useless functions from their devices. For better compatibility with conventional systems, SSD vendors can enhance their SSD products to support two different modes: device-managed flash and application-managed flash. This allows us to choose the proper mode according to requirements. The addition of AFTL to existing SSDs may not require much efforts and hardware resources because of its simplicity.

3 AMF Log-structured File System

In this section, we explain our experience with the design and implementation of ALFS. We implement ALFS in the Linux 3.13 kernel based on an F2FS file system [33]. Optimizing or enhancing the fundamental LFS design is not a goal of this study. For that reason, most of the data structures and modules of F2FS are left unchanged. Instead, we focus on two design aspects: (i) where in-place updates occur in F2FS and (ii) how to modify F2FS for the AMF block I/O interface. The detailed implementation of F2FS is different from other LFSs [38, 31], but its fundamental design concept is the same as its ancestor, Sprite LFS [47]. For the sake of simplicity and generality, we explain the high-level design of ALFS using general terms found in Sprite LFS.

3.1 File Layout and Operations

Figure 2 shows the logical segments of ALFS, along with the corresponding physical segments in AFTL. All user files, directories and inodes, including any modifications/updates, are appended to free space in logical segments, called *data segments*. ALFS maintains an inode map to keep track of inodes scattered across the storage space. The inode map is stored in reserved logical segments, called *inode-map segments*. ALFS also maintains check-points that point to the inode map and keep the consistent state of the file system. A check-point is written periodically or when a flush command (e.g., fsync) is issued. Logical segments reserved for check-points are called *check-point segments*.

ALFS always performs out-of-place updates even for the check-point and the inode-map because of the requirements of the AMF block I/O interface. Hence, their locations are not fixed. This makes it difficult to find the

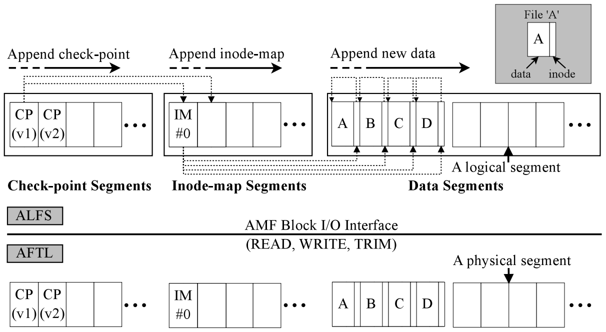


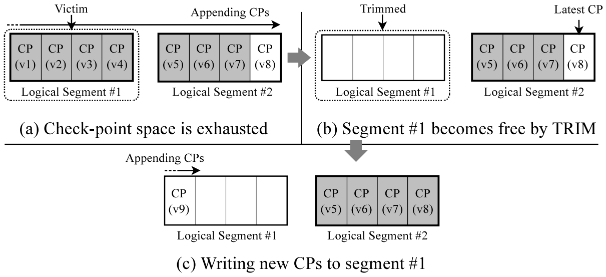
Figure 2: The upper figure illustrates the logical layout of ALFS. There is an initial check-point CP(v1). Four files are appended to data segments along with their inodes in the following order: A, B, C and D. Then, an inode map IM#0 is written which points to the locations of the inodes of the files. Finally, the check-point CP(v2) is written to check-point segments. The bottom figure shows the physical segments corresponding to the logical segments. The data layout of a logical segment perfectly aligns with its physical segment.

latest check-point and the locations of inodes in inode-map segments after mounting or power failure. Next, we explain how ALFS manages check-point segments for quick mount and recovery, and show how it handles inode-map segments for fast searches of inodes.

3.2 Check-Point Segment

The management of check-point segments is straightforward. ALFS reserves *two fixed* logical segments #1 and #2 for check-points. (Note: a logical segment #0 is reserved for a superblock). Figure 3 shows an example of check-point management. ALFS appends new check-points with incremental version numbers using the available free space. If free space in segments is exhausted, the segment containing only old check-point versions is selected as a victim for erasure (see Figure 3(a)). The latest check-point is still kept in the other segment. ALFS sends TRIM commands to invalidate and free the victim (see Figure 3(b)). Then, it switches to the freed segment and keeps writing new check-points (see Figure 3(c)). Even though ALFS uses the same logical segments repeatedly, it will not unevenly wear out flash because AFTL performs wear-leveling.

When ALFS is remounted, it reads all the check-point segments from AFTL. It finds the latest check-point by comparing version numbers. This brute force search is efficient because ALFS maintains only two segments for check-pointing, regardless of storage capacity. Since segments are organized to maximize I/O throughput, this search utilizes full bandwidth and mount time is short.



(a) Check-point space is exhausted

(b) Segment #1 becomes free by TRIM

(c) Writing new CPs to segment #1

Figure 3: Check-point segment handling

3.3 Inode-Map Segment

The management of inode-map segments is more complicated. The inode map size is decided by the maximum number of inodes (i.e., files) and is proportional to storage capacity. If the storage capacity is 1 TB and the minimum file size is 4 KB, 2^{28} files can be created. If each entry of the inode map is 8 B (4 B for an inode number and 4 B for its location in a data segment), then the inode map size is 2 GB (= 8 B$\times 2^{28}$). Because of its large size, ALFS divides the inode map into 4 KB blocks, called *inode-map blocks*. There are 524,288 4-KB inode-map blocks for the inode map of 2 GB, each of which contains the mapping of 512 inodes (see Table 1). For example, IM#0 in Figure 2 is an inode-map block.

ALFS always appends inode-map blocks to free space, so the latest inode-map blocks are scattered across inode-map segments. To identify the latest valid inode-map blocks and to quickly find the locations of inodes, we need to develop another scheme.

Inode-Map Block Management: Figure 4 illustrates how ALFS manages inode-map blocks. To quickly find the locations of inodes, ALFS maintains a table for inode-map blocks (TIMB) in main memory. TIMB consists of 4 B entries that point to inode-map blocks in inode-map segments. Given an inode number, ALFS finds its inode-map block by looking up TIMB. It then obtains the location of the inode from that inode-map block. The TIMB size is 2 MB for 524,288 inode-map blocks (= 4 B$\times 524,288$), so it is small enough to be kept in the host DRAM. The in-memory TIMB should be stored persistently; otherwise, ALFS has to scan all inode-map segments to construct the TIMB dur-

Data structure	Unit size	Count	Storage
Inode-map block	4 KB	524K	Flash (inode-map segs)
In-memory TIMB	2 MB	1	DRAM
TIMB block	4 KB	512	Flash (inode-map segs)
TIMB-blocks list	2 KB	1	Flash (a check-point)

Table 1: An example of data structures sizes and locations with a 1 TB SSD. Their actual sizes vary depending on ALFS implementation (e.g., an inode-map size) and storage capacity.

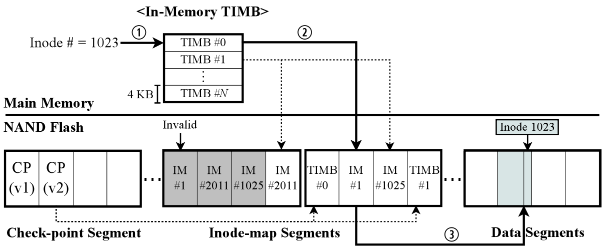


Figure 4: To find an inode, ALFS first looks up in-memory TIMB to find the location of inode-map blocks that points to the inode in flash. Each 4 KB TIMB block indicates 1,024 inode-map blocks in inode-map segments (e.g., TIMB#0 points to IM#0∼IM#1023 in flash). Since each inode-map block points to 512 inodes, TIMB#0 block indicates inodes ranging from 0-524,288 in flash. If ALFS searches for an inode whose number is 1023, it looks up TIMB#0 in the in-memory TIMB (①) and finds the location of IM#1 that points to 512∼1023 inodes (②). Finally, the inode 1023 can be read from a data segment (③). Note that the latest check-point points to all of the physical locations of TIMB blocks in flash.

ing mount. ALFS divides the TIMB into 4 KB blocks (TIMB blocks) and keeps track of dirty TIMB blocks that hold newly updated entries. ALFS appends dirty TIMB blocks to free space in inode-map segments just before a check-point is written.

TIMB blocks themselves are also stored in non-fixed locations. To build the in-memory TIMB and to safely keep it against power failures, a list of all the physical locations of TIMB blocks (TIMB-blocks list) is written to check-point segments together with the latest check-point. Since the size of the in-memory TIMB is 2 MB, the number of TIMB blocks is 512 (= 2 MB/4 KB). If 4 B is large enough to point to locations of TIMB blocks, the TIMB-blocks list is 2 KB (= 4 B×512). The actual size of check-point data is hundred bytes (e.g., 193 bytes in F2FS), so a check-point with a TIMB-block list can be written together to a 4 KB sector without extra writes.

Remount Process: The in-memory TIMB should be reloaded properly whenever ALFS is mounted again. ALFS first reads the latest check-point as we described in the previous subsection. Using a TIMB-blocks list in the check-point, ALFS reads all of the TIMB blocks from inode-map segments and builds the TIMB in the host DRAM. The time taken to build the TIMB is negligible because of its small size (e.g., 2 MB for 1 TB storage).

Up-to-date TIMB blocks and inode-map blocks are written to inode-map segments before a new check-point is written to NAND flash. If the check-point is successfully written, ALFS returns to the consistent state after power failures by reading the latest check-point. All the TIMB blocks and inode-map blocks belonging to an incomplete check-point are regarded as obsolete data. The recovery process of ALFS is the same as the remount process since it is based on LFS [47].

Garbage Collection: When free space in inode-map segments is almost used up, ALFS should perform garbage collection. In the current implementation, the least-recently-written inode-map segment is selected as a victim. All valid inode-map blocks in the victim are copied to a free inode-map segment that has already been reserved for garbage collection. Since some of inode-map blocks are moved to the new segment, the in-memory TIMB should also be updated to point to their new locations accordingly. Newly updated TIMB blocks are appended to the new segment, and the check-point listing TIMB blocks is written to the check-point segment. Finally, the victim segment is invalidated by a TRIM command and becomes a free inode-map segment.

To reduce live data copies, ALFS increases the number of inode-map segments such that their total size is larger than the actual inode-map size. This wastes file-system space but greatly improves garbage collection efficiency because it facilitates inode-map blocks to have more invalid data prior to being selected as a victim. ALFS further improves garbage collection efficiency by separating inode-map blocks (i.e., hot data) in inode-map segments from data segments (i.e., cold data). Currently, ALFS allocates inode-maps segments which are four times larger than its original size (e.g., 8 GB if the inode map size is 2 GB). The space wasted by extra segments is small (e.g., 0.68% = 7 GB / 1 TB).

All of the I/O operations required to manage inode-map blocks are extra overheads that are not present in the conventional LFS. Those extra I/Os account for a small portion, which is less than 0.2% of the total I/Os.

3.4 Data Segment

ALFS manages data segments exactly the same way as in the conventional LFS – it buffers file data, directories and inodes in DRAM and writes them all at once when their total size reaches a data segment size. This buffering is advantageous for ALFS to make use of the full bandwidth of AFTL. ALFS performs segment cleaning when free data segments are nearly exhausted. The procedure of segment cleaning is illustrated in Figure 5.

Besides issuing TRIM commands after segment cleaning, we have not changed anything in F2FS for management of data segments because F2FS already manages data segments in an append-only manner. This is a good example of how easily AMF can be used by other log-structured systems. It also allows us to automatically borrow advanced cleaning features from F2FS [33] without any significant effort.

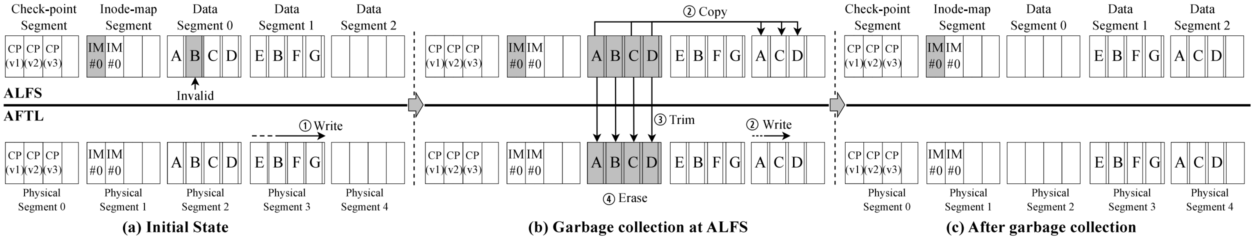


Figure 5: Writes and garbage collection of ALFS with AFTL: (a) Four new files E, B, F and G are written to data segment 1. The file B is a new version. ALFS appends IM#0 to the inode-map segment because it points to the locations of the files. A new check-point CP(v3) is appended. (b) Free space in ALFS is nearly exhausted, so ALFS triggers garbage collection. ALFS copies the files A, C and D to data segment 2. Since data segment 0 has only invalid data, ALFS sends TRIM commands to AFTL, making it free. Finally, AFTL erases physical segment 2. There are 3 page copies and 2 block erasures for garbage collection.

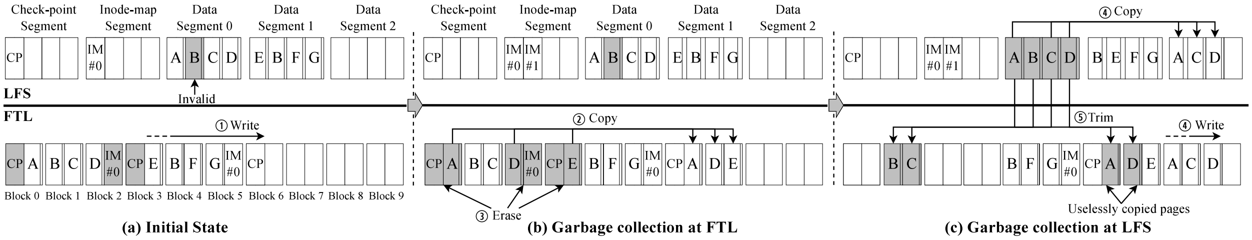


Figure 6: Writes and garbage collection of LFS with FTL: FTL sequentially writes all the sectors to NAND flash using a mapping table. (a) The files E, B, F, and G are appended to free pages. IM#0 and CP are overwritten in the same locations in LFS. FTL maps them to free pages, invaliding old versions. (b) FTL decides to perform garbage collection. It copies flash pages for A, D and E to free pages and gets 3 free blocks (Blocks 0, 2, 3). (c) LFS is unaware of FTL, so it also triggers garbage collection to create free space. It moves the files A, C and D to free space and sends TRIM commands. For garbage collection, there are 6 page copies and 3 block erasures. The files A and D are moved uselessly by FTL because they are discarded by LFS later.

3.5 Comparison with Conventional LFS

In this section, we compare the behavior of ALFS with a conventional LFS that runs on top of a traditional FTL. For the same set of operations, Figure 5 illustrates the behaviors of ALFS, while Figure 6 shows those of the conventional LFS with the FTL. For the sake of simplicity, we assume that ALFS and LFS have the same file-system layout. The sizes of a sector and a flash page are assumed to be the same. LFS keeps check-points and an inode-map in a fixed location and updates them by overwriting new data.[1] On the storage device side, LFS runs the page-level FTL that maps logical sectors to any physical pages in NAND flash. In AFTL, a physical segment is composed of two flash blocks. AFTL just erases flash blocks containing only obsolete pages.

Figures 5 and 6 demonstrate how efficiently ALFS manages NAND flash compared to LFS with the FTL. LFS incurs a larger number of page copies for garbage collection than ALFS. This inefficiency is caused by (i) in-place updates to check-point and inode-map regions by LFS. Whenever overwrites occur, the FTL has to map up-to-date data to new free space, invalidating its old version that must be reclaimed by the FTL later (see Blocks 0, 2, 3 in Figure 6(a)). Other versions of LFS that overwrite only check-points (e.g., Sprite LFS) also have the same problem. (ii) The second is unaligned logging by both LFS and the FTL which results in data from different segments being mixed up in the same flash blocks. To lessen FTL-level garbage collection costs, LFS discards the entire logical segment (i.e., data segment 0) after cleaning, but it unintentionally creates dirty blocks that potentially cause page copies in the future (see Blocks 6 and 7 in Figure 6(c)).

In ALFS, in-place updates are never issued to the device and the data layout of a logical segment perfectly aligns with a corresponding physical segment. Thus, the problems with LFS do not occur in ALFS.

4 AMF Flash Translation Layer (AFTL)

In this section, we explain the design and implementation of AFTL. We implement AFTL in a device driver because our SSD prototype does not have a processor, but it can be implemented in the device if a processor is avail-

[1] This is somewhat different depending on the design of LFS. Sprite LFS overwrites data in a check-point region only [47], while NILFS writes segment summary blocks in an in-place-update fashion [31]. F2FS overwrites both a check-point and an inode map [33]. Since ALFS is based on F2FS, we use the design of F2FS as an example.

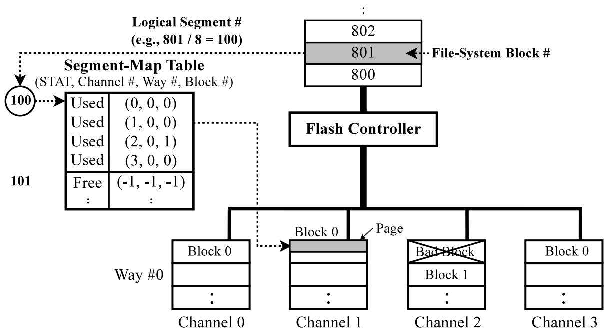


Figure 7: An example of how AFTL handles writes: There are four channels and one way in AFTL, and each block is composed of two pages. A physical segment has 8 pages. When a write request comes, AFTL gets a logical segment number (i.e., 100 = 801/8) using the logical sector number. It then looks up the segment-map table to find a flash block mapped to the logical segment. In this example, the logical block '801' is mapped to 'Block 0' in 'Channel #1'. Finally, AFTL writes the data to a corresponding page offset in the mapped block.

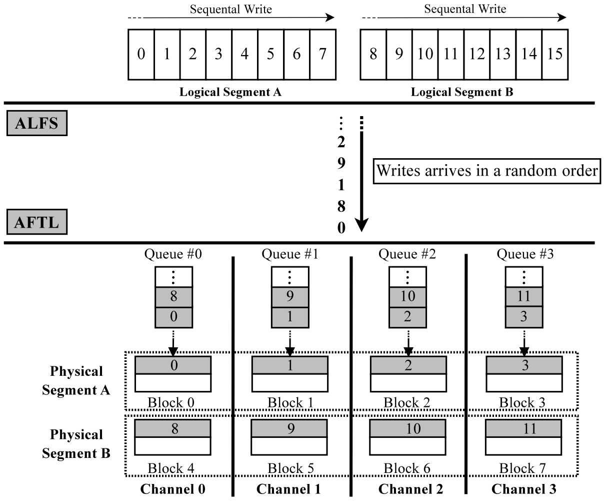


Figure 8: An example of how AFTL handles write requests when ALFS appends data to two segments A and B simultaneously: Numbers inside rectangles indicate a file-system sector address. ALFS sequentially writes data to segments A and B, but write requests arrive at AFTL in a random order (i.e., 0, 8, 1, ...). They are sorted in multiple I/O queues according to their destined channels and are written to physical segments in a way of fully utilizing four channels. If a single queue with FIFO scheduling is used, the sector '1' is delayed until '0' and '8' are sent to flash blocks '0' and '4' through the channel 0.

able. The architecture of AFTL is similar to a simplified version of the block-level FTL [4], except that AFTL does not need to run address remapping to avoid in-place updates, nor does it need to perform garbage collection. For the sake of clarity, we focus on describing the minimum requirements for AFTL implementation rather than explaining how to improve existing FTLs to support the new block I/O interface.

Wear-Leveling and Bad-Block Management: As discussed in Section 2, sectors in a logical segment are statically mapped to flash pages. For wear-leveling and bad-block management, AFTL only needs a small segment-map table that maps a logical segment to a physical segment. Each table entry contains the physical locations of flash blocks mapped to a logical segment along with a status flag (STAT). Each entry in the table points to blocks of a logical segment that is striped across channels and ways. STAT indicates Free, Used or Invalid.

Figure 7 shows how AFTL handles write requests. If any physical blocks are not mapped yet (i.e., STAT is Free or Invalid), AFTL builds the physical segment by allocating new flash blocks. A bad block is not selected. AFTL picks up the least worn-out free blocks in the corresponding channel/way. To preserve flash lifetime and reliability, AFTL can perform static wear-leveling that exchanges the most worn-out segments with the least worn-out ones [7]. If there are previously allocated flash blocks (i.e., STAT is Invalid), they are erased. If a logical segment is already mapped (i.e., STAT is Used), AFTL writes the data to the fixed location in the physical segment. ALFS informs AFTL via TRIM commands that the physical segments have only obsolete data. Then, AFTL can figure out which blocks are out-of-date. Upon receiving the TRIM command, AFTL invalidates that segment by changing its STAT to Invalid. Invalid segments are erased on demand or in background later.

I/O Queueing: AFTL employs per-channel/way I/O queues combined with a FIFO I/O scheduler. This multiple I/O queueing is effective in handling multiple write streams. ALFS allocates several segments and writes multiple data streams to different segments at the same time. For example, a check-point is often written to check-point segments while user files are being written to data segments. Even if individual write streams are sent to segments sequentially, multiple write streams arriving at AFTL could be mixed together and be random I/Os, which degrades I/O parallelism. Figure 8 shows how AFTL handles random writes using multiple queues.

This management of multiple write streams in AMF is more efficient than conventional approaches like multi-streamed SSDs [28]. In multi-streamed SSDs, the number of segments that can be opened for an individual write stream is specified at the configuration time. There is no such a limitation in AMF; ALFS opens as many logical segments as needed to write multiple streams. All the data are automatically separated to different physical segments according to the segment-level mapping. This enables applications to more efficiently separate data according to their properties.

Write skews do not occur for any channel or way in

Capacity	Block-level FTL	Hybrid FTL	Page-level FTL	AMF	
				AFTL	ALFS
512 GB	4 MB	96 MB	512 MB	4 MB	5.3 MB
1 TB	8 MB	186 MB	1 GB	8 MB	10.8 MB

Table 2: A summary of memory requirements

AFTL. This is because ALFS allocates and writes data in the unit of a segment, distributing all the write requests to channels and ways uniformly. Moreover, since FTL garbage collection is never invoked in AFTL, I/O scheduling between normal I/Os and GC I/Os is not required. Consequently, simple multiple I/O queueing is efficient enough to offer good performance, and complex firmware algorithms like load-balancing [8] and out-of-ordering [39, 16] are not required in AFTL.

5 Experimental Results

We begin our analysis by looking at memory requirements for AFTL and ALFS, comparing against commonly used FTL schemes. We then evaluate the performance of AMF using micro-benchmarks to understand its behavior under various I/O access patterns. We benchmark AMF using realistic applications that have more complex I/O access patterns. Finally, we measure lifetime, I/O latency and CPU utilization of the system.

5.1 Memory Requirements

We compare the mapping table sizes of AFTL with three FTL schemes: block-level, hybrid and page-level FTLs. Block-level FTL uses a flash block (512 KB) as the unit of mapping. Because of its low performance, it is rarely used in production SSDs. Page-level FTL performs mapping on flash pages (4-16KB). Hybrid FTL is a combination of block-level and page-level FTLs – while the block-level mapping is used to manage the storage space offered to end-users, the page-level mapping is used for an over-provisioning area. For the hybrid FTL, 15% of the total capacity is used as the over-provisioning area. AFTL maintains the segment-map table pointing to flash blocks for wear-leveling and bad-block management.

Table 2 lists the mapping table sizes of 512 GB and 1 TB SSDs. For the 512 GB SSD, the mapping table sizes are 4 MB, 96 MB, 512 MB and 4 MB for block-level, hybrid, page-level FTLs and AFTL, respectively. The mapping table sizes increase in proportional to the storage capacity – when the capacity is 1 TB, block-level, hybrid, page-level FTLs and AFTL require 8 MB, 62 MB, 1 GB and 8 MB memory, respectively. AFTL maintains a smaller mapping table than the page-level FTL, enabling us to keep all mapping entries in DRAM even for the 1 TB SSD. Table 2 shows the host DRAM requirement for ALFS, including tables for inode-map blocks (TIMB) as well as other data structures. As listed in the table, ALFS requires a tiny amount of host DRAM.

Category	Workload	Description
File System	FIO	A synthetic I/O workload generator
	Postmark	A small and metadata intensive workload
Database	Non-Trans	A non-transactional DB workload
	OLTP	An OLTP workload
	TPC-C	A TPC-C workload
Hadoop	DFSIO	A HDFS I/O throughput test application
	TeraSort	A data sorting application
	WordCount	A word count application

Table 3: A summary of benchmarks

5.2 Benchmark Setup

To understand the effectiveness of AMF, we compared it with two file systems, EXT4 and F2FS [33], running on top of two different FTL schemes, page-level FTL (PFTL) and DFTL [14]. They are denoted by EXT4+PFTL, EXT4+DFTL, F2FS+PFTL, and F2FS+DFTL, respectively.

PFTL was based on pure page-level mapping that maintained all the mapping entries in DRAM. In practice, the mapping table was too large to be kept in DRAM. To address this, DFTL stored all the mapping entries in flash, keeping only popular ones in DRAM. While DFTL reduced the DRAM requirement, it incurred extra I/Os to read/write mapping entries from/to NAND flash. We set the DRAM size so that the mapping table size of DFTL was 20% of PFTL. Since DFTL is based on the LRU-based replacement policy, 20% hot entries of the mapping table were kept in DRAM. For both PFTL and DFTL, greedy garbage collection was used, and an over-provisioning area was set to 15% of the storage capacity. The over-provisioning area was not necessary for AFTL because it did not perform garbage collection. For all the FTLs, the same dynamic wear-leveling algorithm was used, which allocated youngest blocks for writing incoming data.

For EXT4, a default journaling mode was used and the discard option was enabled to use TRIM commands. For F2FS, the segment size was always set to 2 MB which was the default size. For ALFS, the segment size was set to 16 MB which was equal to the physical segment size. ALFS allocated 4x larger inode-map segments than its original size. For both F2FS and ALFS, 5% of file-system space was used as an over-provisioning area which was the default value.

5.3 Performance Analysis

We evaluated AMF using 8 different workloads (see Table 3), spanning 3 categories: file-system, DBMS and

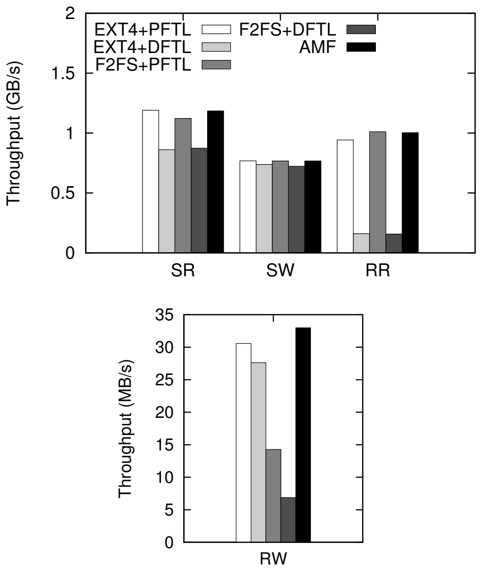


Figure 9: Experimental results with FIO

	EXT4+ PFTL	EXT4+ DFTL	F2FS+ PFTL		F2FS+ DFTL		AMF
	FTL	FTL	FS	FTL	FS	FTL	FS
FIO(SW)	1.00	1.00	1.00	1.00	1.00	1.00	1.00
FIO(RW)	1.41	1.45	1.35	1.82	1.34	2.18	1.38
Postmark(L)	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Postmark(H)	1.12	1.35	1.17	2.23	1.18	2.89	1.16
Non-Trans	1.97	2.00	1.58	2.90	1.59	2.97	1.59
OLTP	1.45	1.46	1.23	1.78	1.23	1.79	1.24
TPC-C	2.33	2.21	1.81	2.80	1.82	5.45	1.87
DFSIO	1.0	1.0	1.0	1.0	1.0	1.0	1.0
TeraSort	1.0	1.0	1.0	1.0	1.0	1.0	1.0
WordCount	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 4: Write amplification factors (WAF). For F2FS, we display WAF values for both the file system (FS) and the FTL. In FIO, the WAF values for the read-only workloads FIO (RR) and FIO (SR) are not included.

Hadoop. To understand the behaviors of AMF under various file-system operations, we conducted a series of experiments using two well known file system benchmarks, FIO [3] and Postmark [30]. We also evaluated AMF using response time sensitive database workloads: Non-Trans, OLTP and TPC-C. Finally, we assessed AMF with Hadoop applications from HiBench [21], HFSIO, TeraSort and WordCount, which required high I/O throughput for batch processing.

For performance measurements, we focused on analyzing the effect of extra I/Os by the FTL on performance specifically caused by garbage collection and swap-in/out of mapping entries. There were no extra I/Os from wear-leveling since dynamic wear-leveling was used. EXT4, F2FS and AMF all performed differently from the perspective of garbage collection. Since EXT4 is a journaling file system, only the FTL in the storage device performed garbage collection. In F2FS, both F2FS and the FTL did garbage collection. In AMF, only ALFS performed garbage collection. There were no extra swapping I/Os in PFTL and AFTL for fetching/evicting mapping entries from/to flash because their tables were always kept in DRAM. Only DFTL incurred extra I/Os to manage in-flash mapping entries. Note that our implementation of PFTL and DFTL might be different from that of commercial FTLs. Two technical issues related to PFTL and DFTL (i.e., cleaning and swapping costs), however, are well known and common problems. For this reason, our results are reasonable enough to understand the benefits of AMF on resolving such problems.

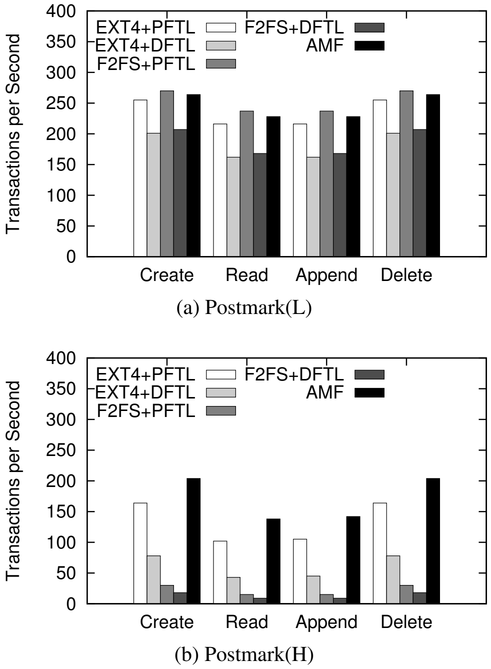
Our experiments were conducted under the same host and flash device setups. The host system was Intel's Xeon server with 24 1.6 GHz cores and 24 GB DRAM. The SSD prototype had 8 channels and 4 ways with 512 GB of NAND flash, composed of 128 4 KB pages per block. The raw performance of our SSD was 240K IOPS (930 MB/s) and 67K IOPS (260 MB/s) for reads and writes, respectively. To quickly emulate aged SSDs where garbage collection occurs, we set the storage capacity to 16 GB. This was a feasible setup because SSD performance was mostly decided by I/O characteristics (e.g., data locality and I/O patterns), not by storage capacity. The host DRAM was set to 1.5 GB to ensure that requests were not entirely served from the page cache.

5.3.1 File System Benchmarks

FIO: We evaluate sequential and random read/write performance using the FIO benchmark. FIO first writes a 10 GB file and performs sequential-reads (SR), random-reads (RR), sequential-writes (SW) and random-writes (RW) on it separately. We use a libaio I/O engine, 128 io-depth, and a 4 KB block, and 8 jobs run simultaneously. Except for them, default parameters are used.

Figure 9 shows our experimental results. For SR and SW, EXT4+PFTL, F2FS+PFTL and AMF show excellent performance. For sequential I/O patterns, extra live page copies for garbage collection do not occur (see Table 4). Moreover, since all the mapping entries are always kept in DRAM, there are no overheads to manage in-flash mapping entries. Note that these performance numbers are higher than the maximum performance of our SSD prototype due to the buffering effect of FIO.

EXT4+DFTL and F2FS+DFTL show slower performance than the others for SR and SW. This is caused by extra I/Os required to read/write mapping entries from/to NAND flash. In our measurements, only about 10% of them are missing in the in-memory mapping table, but its effect on performance is not trivial. When a mapping entry is missing, the FTL has to read it from flash and to



(a) Postmark(L)

(b) Postmark(H)

Figure 10: Experimental results with Postmark

evict an in-memory entry if it is dirty. While the FTL is doing this task, an incoming request has to be suspended. Moreover, it is difficult to fully utilize I/O parallelism when reading in-flash mapping entries because their locations were previously decided when they were evicted.

The performance degradation due to missing entries becomes worse with random-reads (RR) patterns because of their low hit ratio in the in-memory mapping table – about 67% of mapping entries are missing. For this reason, EXT4+DFTL and F2FS+DFTL show slow performance for RR. On the other hand, EXT4+PFTL, F2FS+PFTL and AMF exhibit good performance.

RW incurs many extra copies for garbage collection because of its random-writes patterns. AMF outperforms all the other schemes, exhibiting the highest I/O throughput and the lowest write amplification factor (WAF) (see Table 4). EXT4+PFTL shows slightly lower performance than AMF, but its performance is similar to that of AMF. In particular, F2FS+PFTL shows lower performance than AMF and EXT4+PFTL. This is because of duplicate storage management by F2FS and the FTL. F2FS has a similar WAF value as AMF, performing segment cleaning efficiently. However, extra writes for segment cleaning are sent to the FTL and trigger additional garbage collection at the FTL level, which results in extra page copies.[2] EXT4 and F2FS with DFTL show worse performance than those with PFTL because of extra I/Os for in-flash mapping-entries management.

Postmark: After the assessments of AMF with various I/O patterns, we evaluate AMF with Postmark, which is a small I/O and metadata intensive workload. To understand how garbage collection affects overall performance, we perform our evaluations with two different scenarios, light and heavy, denoted by Postmark(L) and Postmark(H). They each simulate situations where a storage space utilization is low (40%) and high (80%) – for Postmark(L), 15K files are created; for Postmark(H), 30K files are created. For both cases, file sizes are 5K-512KB and 60K transactions run.

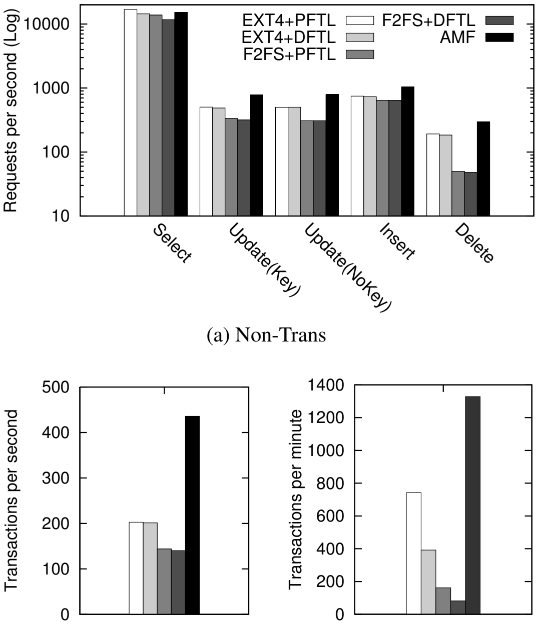
Figure 10 shows experimental results. F2FS+PFTL shows the best performance with the light workload, where few live page copies occur for garbage collection (except for block erasures) because of the low utilization of the storage space. EXT4+PFTL and AMF show fairly good performance as well. For the heavy workload where many live page copies are observed, AMF achieves the best performance. On the other hand, the performance of F2FS+PFTL deteriorates significantly because of the duplicate management problem. F2FS and EXT4 with DFTL perform worse because of overheads caused by in-flash mapping-entries management.

From the experimental results with Postmark, we also confirm that extra I/Os required to manage inode-map segments do not badly affect overall performance. Postmark generates many metadata updates, which requires lots of inode changes. Compared with other benchmarks, Postmark issues more I/O traffic to inode-map segments, but it accounts for only about 1% of the total I/Os. Therefore, its effect on performance is negligible. We will analyze it in detail Section 5.5.

5.3.2 Application Benchmarks

Database Application: We compare the performance of AMF using DBMS benchmarks. MySQL 5.5 with an Innodb storage engine is selected. Default parameters are used for both MySQL and Innodb. Non-Trans is used to evaluate performance with different types of queries: Select, Update (Key), Update (NoKey), Insert and Delete. The non-transactional mode of a SysBench benchmark is used to generate individual queries [32]. OLTP is an I/O intensive online transaction processing (OLTP) workload generated by the SysBench tool. For both Non-Trans and OLTP, 40 million table entries are created and 6 threads run simultaneously. TPC-C is a well-known OLTP workload. We run TPC-C on 14 warehouses with 16 clients each for 1,200 seconds.

[2] The segment size could affect performance of F2FS – F2FS shows better performance when its segment size is equal to the physical segment (16 MB). However, F2FS still suffers from the duplicate management problem, so it exhibits worse performance than AMF, regardless of the segment size. For this reason, we exclude results with various segment sizes and use the default segment size (2 MB).



(a) Non-Trans

(b) OLTP

(c) TPC-C

Figure 11: Experimental results with database apps.

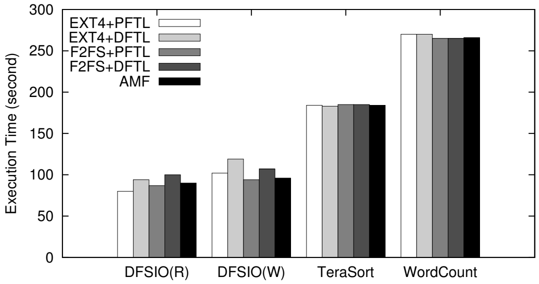


Figure 12: Experimental results with Hadoop apps.

Figure 11 shows the number of transactions performed under the different configurations. AMF outperforms all other schemes. Compared with the micro-benchmarks, database applications incur higher garbage collection overheads because of complicated I/O patterns. As listed in Table 4, AMF shows lower WAFs than EXT4+PFTL and EXT4+DFTL thanks to more advanced cleaning features borrowed from F2FS. F2FS+PFTL and F2FS+DFTL show similar file-system-level WAFs as AMF, but because of high garbage collection costs at the FTL level, they exhibit lower performance than AMF. The state-of-the-art FTLs used by SSD vendors maybe work better with more advanced features, but it comes at the price of more hardware resources and design complexity. In that sense, this result shows how efficiently and cost-effectively flash can be managed by the application.

Hadoop Application: We show measured execution times of Hadoop applications in Figure 12. Hadoop applications run on top of the Hadoop Distributed File System (HDFS) which manages distributed files in large clusters. HDFS does not directly manage physical storage devices. Instead, it runs on top of regular local disk file systems, such as EXT4, which deal with local files. HDFS always creates/deletes large files (e.g., 128 MB) on the disk file system to efficiently handle large data sets and to leverage maximum I/O throughput from sequentially accessing these files.

This file management of HDFS is well-suited for NAND flash. A large file is sequentially written across multiple flash blocks, and these flash blocks are invalidated together when the file is removed from HDFS. Therefore, FTL garbage collection is done by simply erasing flash blocks without any live page copies. Moreover, because of its sequential access patterns, the effect of missing mapping entries on performance is not significant. This is the reason why all five storage configurations show similar performance for Hadoop applications. The results also indicate that existing flash storage is excessively over-designed. With the exception of error management and coarse-grain mapping, almost all storage management modules currently implemented in the storage device are not strictly necessary for Hadoop.

5.4 Lifetime Analysis

We analyze the lifetime of the flash storage for 10 different write workloads. We estimate expected flash lifetime using the number of block erasures performed by the workloads since NAND chips are rated for a limited number of program/erase cycles. As shown in Figure 13, AMF incurs 38% fewer erase operations overall compared to F2FS+DFTL.

5.5 Detailed Analysis

We also analyze the inode-map management overheads, CPU utilizations and I/O latencies.

Inode-map Management Overheads: I/O operations

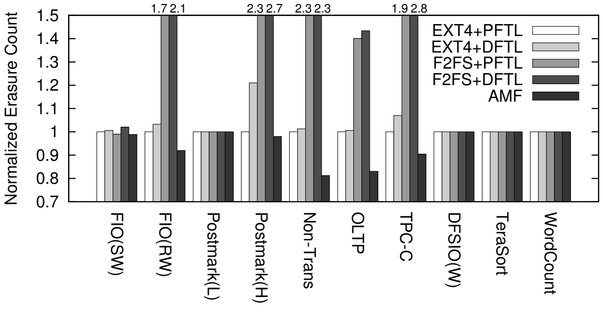
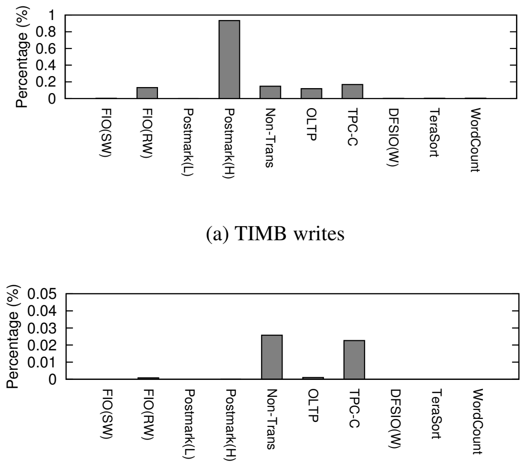


Figure 13: Erasure operations normalized to EXT4+PFTL



(a) TIMB writes

(b) Inode-map garbage collection

Figure 14: Inode-map management overheads analysis

required to manage inode-map segments in ALFS are extra overheads. Figure 14(a) shows the percentage of TIMB writes to flash storage. We exclude read-only workloads. TIMB writes account for a small proportion of the total writes. Moreover, the number of dirty TIMB blocks written together with a new check-point is small – 2.6 TIMB blocks are written, on average, when a check-point is written. Figure 14(b) illustrates how many extra copies occur for garbage collection in inode-map segments. Even though there are minor differences among the benchmarks, overall extra data copies for inode-map segments are insignificant compared to the total number of copies performed in the file system.

Host CPU/DRAM Utilization: We measure the CPU utilization of `AMF` while running `Postmark(H)`, and compare it with those of `EXT4+PFTL` and `F2FS+PFTL`. As depicted in Figure 15, the CPU utilization of `AMF` is similar to the others. `AMF` does not employ any additional layers or complicated algorithms to manage NAND flash. Only existing file-system modules (F2FS) are slightly modified to support our block I/O interface. As a result, extra CPU cycles required for `AMF` are negligible.

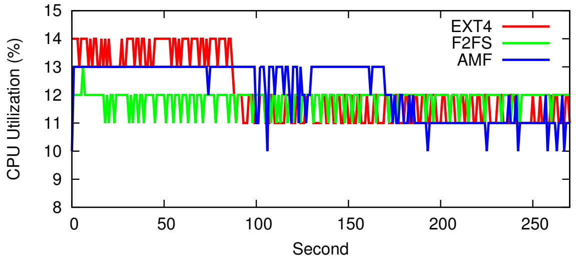


Figure 15: CPU utilization (%)

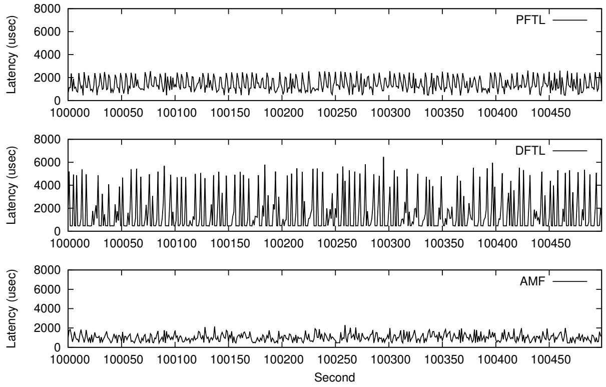


Figure 16: Write latency (μsec)

Host DRAM used by `AMF` is trivial. `AMF` with 16 GB flash requires 180 KB more DRAM than F2FS. Almost all of host DRAM is used to keep AMF-specific data structures (e.g., in-memory TIMB). The host DRAM requirement increases in proportion to the storage capacity, but, as shown in Table 1, it is small enough even for a large SSD (e.g., 10.8 MB for a 1 TB SSD).

I/O Latency: We measure I/O response times of three different FTLs, AMF, PFTL and DFTL, while running `Postmark(H)`. We particularly measure write latencies that are badly affected by both garbage collection and missing mapping entries. As shown in Figure 16, AMF has the shortest I/O response times with small fluctuations since only block erasures are conducted inside the FTL. On the other hand, PFTL and DFTL incur large fluctuations on response times because of FTL garbage collection and in-flash mapping-entries management.

6 Related Work

FTL Improvement with Enhanced Interfaces: Delivering system-level information to the FTL with extended I/O interfaces has received attention because of its advantage in device-level optimization [15, 28, 9, 17]. For example, file access patterns of applications [15] and multi-streaming information [28] are useful in separating data to reduce cleaning costs. Some techniques go one step further by offloading part or all of the file-system functions onto the device (e.g., file creations or the file-system itself) [29, 35, 54]. The FTL can exploit rich file-system information and/or effectively combine its internal operations with the file system for better flash management. The common problem with those approaches is that they require more hardware resources and greater design complexity. In AMF, host software directly manages flash devices, so the exploitation of system-level information can be easily made without additional interfaces or offloading host functions to the device.

Direct Flash Management without FTL: Flash file systems (FFS) [52, 37] and NoFTL [18] are designed to directly handle raw NAND chips through NAND-specific interfaces [51, 22]. Since there is no extra layer, it works efficiently with NAND flash with smaller memory and less CPUs power. Designing/optimizing systems for various vendor-specific storage architectures, however, is in fact difficult. The internal storage architectures and NAND properties are both complex to manage and specific for each vendor and semiconductor-process technology. Vendors are also reluctant to divulge the internal architecture of their devices. The decrease in reliability of NAND flash is another problem – this unreliable NAND can be more effectively managed inside the storage device where detailed physical information is available [13, 43]. For this reason, FFS is rarely used these days except in small embedded systems. AMF has the same advantages as FFS and NoFTL, however, by hiding internal storage architectures and unreliable NAND behind the block I/O interface, AMF eliminates all the concerns about architectural differences and reliability.

Host-Managed Flash: Host-based FTLs like DFS [24, 26, 41] are different from this study in that they just move the FTL to a device driver layer from storage firmware. If log-structured systems like LFS run on top of the device driver with the FTL, two different software layers (i.e., LFS and the FTL in the device driver) run their own garbage collection. As a result, host-based FTLs still have the same problems that the conventional FTL-based storage has.

A software defined flash (SDF) [40] exposes each flash channel to upper layers as individual devices with NAND I/O primitives (e.g., block erasure). Host applications are connected to channels each through a custom interface. In spite of the limited performance of a single channel, it achieves high aggregate throughput by running multiple applications in parallel. SDF is similar to our study in that it minimizes the functionality of the device and allows applications to directly manage the device. This approach, however, is suitable for special environments like the datacenter where aggregate I/O throughput is important and applications can easily access specialized hardware through custom interfaces. AMF is more general – because of compatibility with the existing I/O stacks, if modules that cause overwrites are modified to avoid it, any application can run on AMF.

REDO [34] shows that the efficient integration of a file system and a flash device offers great performance improvement. However, it does not consider important technical issues, such as metadata management affecting performance and data integrity, efficient exploitation of multiple channels, and I/O queueing. REDO is based on a simulation study, so it is difficult to know its feasibility and impact in real world systems and applications.

7 Conclusion

In this paper, we proposed the Application-Managed Flash (AMF) architecture. AMF was based on a new block I/O interface exposing flash storage as append-only segments, while hiding unreliable NAND devices and vendor-specific details. Using our new block I/O interface, we developed a file system (ALFS) and a storage device with a new FTL (AFTL). Our evaluation showed that AMF outperformed conventional file systems with the page-level FTL, both in term of performance and lifetime, while using significantly less resources.

The idea of AMF can be extended to various systems, in particular, log-structured systems. Many DBMS engines manage storage devices in an LFS-like manner [49, 1, 6], so we expect that AMF can be easily adapted to them. A storage virtualization platform could be a good target application where log-structured or CoW file systems [20] coupled with a volume manager [10] manage storage devices with its own indirection layer. A key-value store based on log-structured merge-trees is also a good target application [42, 12, 2]. According to the concept of AMF, we are currently developing a new key-value store to build cost-effective and high-performance distributed object storage.

Acknowledgments

We would like to thank Keith A. Smith, our shepherd, and anonymous referees for valuable suggestions. Samsung (Res. Agmt. Eff. 01/01/12), Quanta (Agmt. Dtd. 04/01/05), and SK hynix memory solutions supported our research. We also thank Jamey Hicks, John Ankcorn and Myron King from Quanta Research Cambridge for their help in developing device drivers. Sungjin Lee was supported by the National Research Foundation of Korea (NRF) grant (NRF-2013R1A6A3A03063762). The work of Jihong Kim was supported by the NRF grant funded by the Ministry of Science, ICT and Future Planning (MSIP) (NRF-2013R1A2A2A01068260) and the Next-Generation Information Computing Development Program through the NRF funded by the MSIP (NRF-2015M3C4A7065645).

Source code

All of the source code including both software and hardware are available to the public under the MIT license. Please refer to the following git repositories: `https://github.com/chamdoo/bdbm_drv.git` and `https://github.com/sangwoojun/bluedbm.git`.

References

[1] RethinkDB. `http://rethinkdb.com`, 2015.

[2] RocksDB: A persistent key-value store for fast storage environments. `http://rocksdb.org`, 2015.

[3] Axboe, J. FIO benchmark. `http://freecode.com/projects/fio`, 2013.

[4] Ban, A. Flash file system, 1995. US Patent 5,404,485.

[5] Bernstein, P. A., Reid, C. W., and Das, S. Hyder – A transactional record manager for shared flash. In *Proceedings of the biennial Conference on Innovative Data Systems Research* (2011).

[6] 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. *ACM Transactions on Computer Systems 26*, 2 (2008), 4.

[7] Chang, L.-P. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the Symposium on Applied Computing* (2007), pp. 1126–1130.

[8] Chang, Y.-B., and Chang, L.-P. A self-balancing striping scheme for NAND-flash storage systems. In *Proceedings of the Symposium on Applied Computing* (2008), pp. 1715–1719.

[9] Choi, H. J., Lim, S.-H., and Park, K. H. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage 4*, 4 (2009), 14:1–14:22.

[10] 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* (2008), pp. 129–142.

[11] Feldman, T., and Gibson, G. Shingled magnetic recording areal density increase requires new data management. *USENIX issue 38*, 3 (2013).

[12] Ghemawat, S., and Dean, J. LevelDB. `http://leveldb.org`, 2015.

[13] Grupp, L. M., Davis, J. D., and Swanson, S. The bleak future of NAND flash memory. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2012).

[14] Gupta, A., Kim, Y., and Urgaonkar, B. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 229–240.

[15] Ha, K., and Kim, J. A program context-aware data separation technique for reducing garbage collection overhead in NAND flash memory. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/O* (2011).

[16] Hahn, S., Lee, S., and Kim, J. SOS: Software-based out-of-order scheduling for high-performance NAND flash-based SSDs. In *Proceedings of the International Symposium on Mass Storage Systems and Technologies* (2013), pp. 1–5.

[17] Hahn, S. S., Jeong, J., and Kim, J. To collect or not to collect: Just-in-time garbage collection for high-performance SSDs with long lifetimes. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implemenation (Poster)* (2014).

[18] Hardock, S., Petrov, I., Gottstein, R., and Buchmann, A. NoFTL: Database systems on FTL-less flash storage. In *Proceedings of the VLDB Endowment* (2013), pp. 1278–1281.

[19] Hitachi. Hitachi accelerated flash, 2015.

[20] Hitz, D., Lau, J., and Malcolm, M. A. File system design for an NFS file server appliance. In *Proceedings of the Winter USENIX Conference* (1994).

[21] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. The HiBench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proceedings of the International Workshop on Data Engineering* (2010), pp. 41–51.

[22] Hunter, A. A brief introduction to the design of UBIFS, 2008.

[23] Jiang, S., Zhang, L., Yuan, X., Hu, H., and Chen, Y. S-FTL: An efficient address translation for flash memory by exploiting spatial locality. In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies* (2011), pp. 1–12.

[24] Josephson, W. K., Bongo, L. A., Li, K., and Flynn, D. DFS: A file system for virtualized flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2010).

[25] Jun, S.-W., Liu, M., Lee, S., Hicks, J., Ankcorn, J., King, M., Xu, S., and Arvind. BlueDBM: An appliance for big data analytics. In *Proceedings of the Annual International Symposium on Computer Architecture* (2015), pp. 1–13.

[26] Jung, M., Wilson, III, E. H., Choi, W., Shalf, J., Aktulga, H. M., Yang, C., Saule, E., Catalyurek, U. V., and Kandemir, M. Exploring the future of out-of-core computing with compute-local non-volatile memory. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013), pp. 75:1–75:11.

[27] Jung, S.-M., Jang, J., Cho, W., Cho, H., Jeong, J., Chang, Y., Kim, J., Rah, Y., Son, Y., Park, J., Song, M.-S., Kim, K.-H., Lim, J.-S., and Kim, K. Three dimensionally stacked NAND flash memory technology using stacking single crystal Si layers on ILD and TANOS structure for beyond 30nm node. In *Proceedings of the International Electron Devices Meeting* (2006), pp. 1–4.

[28] Kang, J.-U., Hyun, J., Maeng, H., and Cho, S. The multi-streamed solid-state drive. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems* (2014).

[29] Kang, Y., Yang, J., and Miller, E. L. Efficient storage management for object-based flash memory. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (2010).

[30] Katcher, J. PostMark: A new filesystem benchmark. *NetApp Technical Report TR3022* (1997).

[31] Konishi, R., Amagai, Y., Sato, K., Hifumi, H., Kihara, S., and Moriai, S. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review 40*, 3 (2006), 102–107.

[32] Kopytov, A. SysBench: A system performance benchmark. `http://sysbench.sourceforge.net`, 2004.

[33] Lee, C., Sim, D., Hwang, J.-Y., and Cho, S. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2015).

[34] Lee, S., Kim, J., and Arvind. Refactored design of I/O architecture for flash storage. *Computer Architecture Letters 14*, 1 (2015), 70–74.

[35] Lee, Y.-S., Kim, S.-H., Kim, J.-S., Lee, J., Park, C., and Maeng, S. OSSD: A case for object-based solid state drives. In *Proceedings of the International Symposium on Mass Storage Systems and Technologies* (2013), pp. 1–13.

[36] Liu, M., Jun, S.-W., Lee, S., Hicks, J., and Arvind. minFlash: A minimalistic clustered flash array. In *Proceedings of the Design, Automation and Test in Europe Conference* (2016).

[37] Manning, C. YAFFS: Yet another flash file system, 2004.

[38] MIN, C., KIM, K., CHO, H., LEE, S.-W., AND EOM, Y. I. SFS: random write considered harmful in solid state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2012).

[39] NAM, E. H., KIM, B., EOM, H., AND MIN, S. L. Ozone (O3): An out-of-order flash memory controller architecture. *IEEE Transactions on Computers 60*, 5 (2011), 653–666.

[40] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), pp. 471–484.

[41] OUYANG, X., NELLANS, D., WIPFEL, R., FLYNN, D., AND PANDA, D. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the International Symposium on High Performance Computer Architecture* (2011), pp. 301–311.

[42] ONEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[43] PAN, Y., DONG, G., AND ZHANG, T. Error rate-based wear-leveling for NAND flash memory at highly scaled technology nodes. *IEEE Transactions on Very Large Scale Integration Systems 21*, 7 (2013), 1350–1354.

[44] PARK, D., DEBNATH, B., AND DU, D. CFTL: A convertible flash translation layer adaptive to data access patterns. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2010), pp. 365–366.

[45] PHISON. PS3110 controller, 2014.

[46] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage 9*, 3 (2013), 9.

[47] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10* (1991), 1–15.

[48] SAMSUNG. Samsung SSD 840 EVO data sheet, rev. 1.1, 2013.

[49] VO, H. T., WANG, S., AGRAWAL, D., CHEN, G., AND OOI, B. C. LogBase: a scalable log-structured database system in the cloud. *Proceedings of the VLDB Endowment* (2012).

[50] VRABLE, M., SAVAGE, S., AND VOELKER, G. M. Bluesky: A cloud-backed file system for the enterprise. In *Proceedings of the USENIX conference on File and Storage Technologies* (2012).

[51] WOODHOUSE, D. Memory technology device (MTD) subsystem for Linux, 2005.

[52] WOODHOUSE, D. JFFS2: The journalling flash file system, version 2, 2008.

[53] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., AND SUNDARARAMAN, S. Don't stack your log on my log. In *Proceedings of the Workshop on Interactions of NVM/Flash with Operating Systems and Workloads* (2014).

[54] ZHANG, Y., ARULRAJ, L. P., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2012).

CloudCache: On-demand Flash Cache Management for Cloud Computing

Dulcardo Arteaga Jorge Cabrera
Florida International University

Jing Xu
VMware Inc.

Swaminathan Sundararaman
Parallel Machines

Ming Zhao
Arizona State University

Abstract

Host-side flash caching has emerged as a promising solution to the scalability problem of virtual machine (VM) storage in cloud computing systems, but it still faces serious limitations in capacity and endurance. This paper presents CloudCache, an on-demand cache management solution to meet VM cache demands and minimize cache wear-out. First, to support on-demand cache allocation, the paper proposes a new cache demand model, Reuse Working Set (RWS), to capture only the data with good temporal locality, and uses the RWS size (RWSS) to model a workload's cache demand. By predicting the RWSS online and admitting only RWS into the cache, CloudCache satisfies the workload's actual cache demand and minimizes the induced wear-out. Second, to handle situations where a cache is insufficient for the VMs' demands, the paper proposes a dynamic cache migration approach to balance cache load across hosts by live migrating cached data along with the VMs. It includes both on-demand migration of dirty data and background migration of RWS to optimize the performance of the migrating VM. It also supports rate limiting on the cache data transfer to limit the impact to the co-hosted VMs. Finally, the paper presents comprehensive experimental evaluations using real-world traces to demonstrate the effectiveness of CloudCache.

1 Introduction

Host-side flash caching employs flash-memory-based storage on a virtual machine (VM) host as the cache for its remote storage to exploit the data access locality and improve the VM performance. It has received much attention in recent years [10, 1, 14, 7], which can be attributed to two important reasons. First, as the level of consolidation continues to grow in cloud computing systems, the scalability of shared VM storage servers becomes a serious issue. Second, the emergence of flash-memory-based storage has made flash caching a promising option to address this IO scalability issue, because accessing a local flash cache is substantially faster than accessing the remote storage across the network.

However, due to the capacity and cost constraints of flash devices, the amount of flash cache that can be employed on a host is much limited compared to the dataset sizes of the VMs, particularly considering the increasing data intensity of the workloads and increasing number of workloads consolidated to the host via virtualization. Therefore, to fulfill the potential of flash caching, it is important to allocate the shared cache capacity among the competing VMs according to their actual demands. Moreover, flash devices wear out by writes and face serious endurance issues, which are in fact aggravated by the use for caching because both the writes inherent in the workload and the reads that miss the cache induce wear-out [33, 15]. Therefore, the cache management also needs to be careful not to admit data that are not useful to workload performance and only damage the endurance.

We propose CloudCache to address the above issues in flash caching through *on-demand cache management*. Specifically, it answers two challenging questions. First, *how to allocate a flash cache to VMs according to their cache demands?* Flash cache workloads depend heavily on the dynamics in the upper layers of the IO stack and are often unfeasible to profile offline. The classic working set model studied for processor and memory cache management can be applied online, but it does not consider the reuse behavior of accesses and may admit data that are detrimental to performance and endurance. To address this challenge, we propose a new cache demand model, *Reuse Working Set (RWS)*, to capture the data that have good temporal locality and are essential to the workload's cache hit ratio, and use the RWS size (*RWSS*), to represent the workload's cache demand. Based on this model, we further use prediction methods to estimate a workload's cache demand online and use new cache admission policies to admit only the RWS into cache, thereby delivering a good performance to the workload while minimizing the wear-out. Cloud-

Cache is then able to allocate the shared cache capacity to the VMs according to their actual cache demands.

The second question is *how to handle situations where the VMs' cache demands exceed the flash cache's capacity.* Due to the dynamic nature of cloud workloads, such cache overload situations are bound to happen in practice and VMs will not be able to get their desired cache capacity. To solve this problem, we propose a *dynamic cache migration* approach to balance cache load across hosts by live migrating the cached data along with the VMs. It uses both on-demand migration of dirty data to provide zero downtime to the migrating VM, and background migration of RWS to quickly warmup the cache for the VM, thereby minimizing its performance impact. Meanwhile, it can also limit the data transfer rate for cache migration to limit the impact to other co-hosted VMs.

We provide a practical implementation of CloudCache based on block-level virtualization [13]. It can be seamlessly deployed onto existing cloud systems as a drop-in solution and transparently provide caching and on-demand cache management. We evaluate it using a set of long-term traces collected from real-world cloud systems [7]. The results show that RWSS-based cache allocation can substantially reduce cache usage and wear-out at the cost of only small performance loss in the worst case. Compared to the WSS-based cache allocation, the RWSS-based method reduces a workload's cache usage by up to 76%, lowers the amount of writes sent to cache device by up to 37%, while delivering the same IO latency performance. Compared to the case where the VM can use the entire cache, the RWSS-based method saves even more cache usage while delivering an IO latency that is only 1% slower at most. The results also show that the proposed dynamic cache migration reduces the VM's IO latency by 93% compared to no cache migration, and causes at most 21% slowdown to the co-hosted VMs during the migration. Combining these two proposed approaches, CloudCache is able to improve the average hit ratio of 12 concurrent VMs by 28% and reduce their average 90th percentile IO latency by 27%, compared to the case without cache allocation.

To the best of our knowledge, CloudCache is the first to propose the RWSS model for capturing a workload's cache demand from the data with good locality and for guiding the flash cache allocation to achieve both good performance and endurance. It is also the first to propose dynamic cache migration for balancing the load across distributed flash caches and with optimizations to minimize the impact of cache data transfer. While the discussion in the paper focuses on flash-memory-based caches, we believe that the general CloudCache approach is also applicable to new nonvolatile memory (NVM) technologies (e.g., PCM, 3D Xpoint) which will likely be used as a cache layer, instead of replacing DRAM, in the storage hierarchy and will still need on-demand cache allocation to address its limited capacity (similarly to or less than flash) and endurance (maybe less severe than flash).

The rest of the paper is organized as follows: Section 2 and Section 3 present the motivations and architecture of CloudCache, Section 4 and Section 5 describe the on-demand cache allocation and dynamic cache migration approaches, Section 6 discusses the integration of these two approaches, Section 7 examines the related work, and Section 8 concludes the paper.

2 Motivations

The emergence of flash-memory-based storage has greatly catalyzed the adoption of a new flash-based caching layer between DRAM-based main memory and HDD-based primary storage [10, 1, 7, 24]. It has the potential to solve the severe scalability issue that highly consolidated systems such as public and private cloud computing systems are facing. These systems often use shared network storage [20, 5] to store VM images for the distributed VM hosts, in order to improve resource utilization and facilitate VM management (including live VM migration [11, 25]). The availability of a flash cache on a VM host can accelerate the VM data accesses using data cached on the local flash device, which are much faster than accessing the hard-disk-based storage across network. Even with the increasing adoption of flash devices as primary storage, the diversity of flash technologies allows the use of a faster and smaller flash device (e.g., single-level cell flash) as the cache for a slower but larger flash device (e.g., multi-level cell flash) used as primary storage.

To fulfill the potential of flash caching, it is crucial to employ *on-demand cache management*, i.e., allocating shared cache capacity among competing workloads based on their demands. The capacity of a commodity flash device is typically much smaller than the dataset size of the VMs on a single host. How the VMs share the limited cache capacity is critical to not only their performance but also the flash device endurance. On one hand, if a workload's necessary data cannot be effectively cached, it experiences orders of magnitude higher latency to fetch the missed data from the storage server and at the same time slows down the server from servicing the other workloads. On the other hand, if a workload occupies the cache with unnecessary data, it wastes the valuable cache capacity and compromises other workloads that need the space. Unlike in CPU allocation where a workload cannot use more than it needs, an active cache workload can occupy all the allocated space beyond its actual demand, thereby hurting both the performance of other workloads and the endurance of flash device.

S-CAVE [22] and vCacheShare [24] studied how to

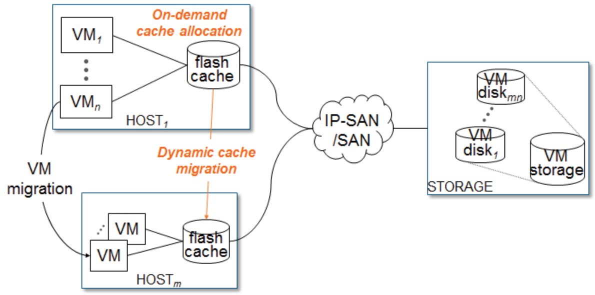


Figure 1: Architecture of CloudCache

optimize flash cache allocation according to a certain criteria (e.g., a utility function), but they cannot estimate the workloads' actual cache demands and thus cannot meet such demands for meeting their desired performance. HEC [33] and LARC [15] studied cache admission policies to reduce the wear-out damage caused by data with weak temporal locality, but they did not address the cache allocation problem. Bhagwat *et al.* studied how to allow a migrated VM to access the cache on its previous host [9], but they did not consider the performance impact to the VMs. There are related works studying other orthogonal aspects of flash caching, including write policies [17], deduplication/compression [21], and other design issues [14, 7]. A detailed examination of related work is presented in Section 7.

3 Architecture

CloudCache supports *on-demand cache management* based on a typical flash caching architecture illustrated in Figure 1. The VM hosts share a network storage for storing the VM disks, accessed through SAN or IP SAN [20, 5]. Every host employs a flash cache, shared by the local VMs, and every VM's access to its remote disk goes through this cache. CloudCache provides on-demand allocation of a flash cache to its local VMs and dynamic VM and cache migration across hosts to meet the cache demands of the VMs. Although our discussions in this paper focus on block-level VM storage and caching, our approaches also work for network file system based VM storage, where CloudCache will manage the allocation and migration for caching a VM disk file in the same fashion as caching a VM's block device. A VM disk is rarely write-shared by multiple hosts, but if it does happen, CloudCache needs to employ a cache consistency protocol [26], which is beyond the scope of this paper.

CloudCache supports different write caching policies: (1) *Write-invalidate*: The write invalidates the cached block and is submitted to the storage server; (2) *Write-through*: The write updates both the cache and the storage server; (3) *Write-back*: The write is stored in the cache immediately but is submitted to the storage server later when it is evicted or when the total amount of dirty data in the cache exceeds a predefined threshold. The write-invalidate policy performs poorly for write-intensive workloads. The write-through policy's performance is close to write-back when the write is submitted to the storage server asynchronously and the server's load is light [14]; otherwise, it can be substantially worse than the write-back policy [7]. Our proposed approaches work for all these policies, but our discussions focus on the write-back policy due to limited space for our presentation. The reliability and consistency of delayed writes in write-back caching are orthogonal issues to this paper's focus, and CloudCache can leverage the existing solutions (e.g., [17]) to address them.

Trace	Time (days)	Total IO (GB)	WSS (GB)	Write (%)
Webserver	281	2,247	110	51
Moodle	161	17,364	223	13
Fileserver	152	57,887	1037	22

Table 1: Trace statistics

In the next few sections, we introduce the two components of CloudCache, *on-demand cache allocation* and *dynamic cache migration*. As we describe the designs, we will also present experimental results as supporting evidence. We consider a set of block-level IO traces [7] collected from a departmental private cloud as representative workloads. The characteristics of the traces are summarized in Table 1. These traces allow us to study long-term cache behavior, in addition to the commonly used traces [4] which are only week-long.

4 On-demand Cache Allocation

CloudCache addresses two key questions about on-demand cache allocation. First, *how to model the cache demand of a workload?* A cloud workload includes IOs with different levels of temporal locality which affect the cache hit ratio differently. A good cache demand model should be able to capture the IOs that are truly important to the workload's performance in order to maximize the performance while minimizing cache utilization and flash wear-out. Second, *how to use the cache demand model to allocate cache and admit data into cache?* We need to predict the workload's cache demand accurately online in order to guide cache allocation, and admit only the useful data into cache so that the allocation does not get overflown. In this section, we present the CloudCache's solutions to these two questions.

4.1 RWS-based Cache Demand Model

Working Set (WS) is a classic model often used to estimate the cache demand of a workload. The working set $WS(t,T)$ at time t is defined as the set of distinct (address-wise) data blocks referenced by the workload during a time interval $[t-T,t]$ [12]. This definition uses

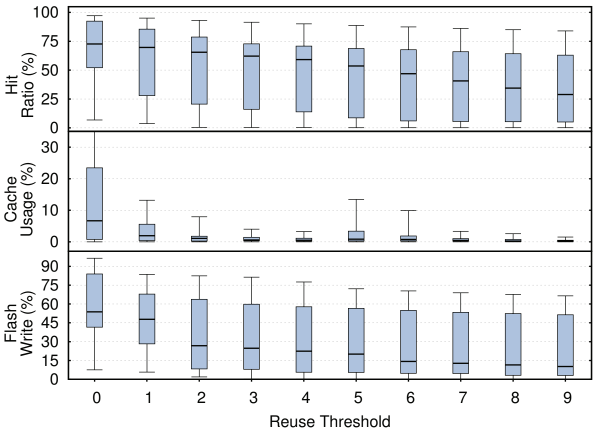


Figure 2: RWS analysis using different values of N

the principle of locality to form an estimate of the set of blocks that the workload will access next and should be kept in the cache. The *Working Set Size* (WSS) can be used to estimate the cache demand of the workload.

Although it is straightforward to use WSS to estimate a VM's flash cache demand, a serious limitation of this approach is that it does not differentiate the level of temporal locality of the data in the WS. Unfortunately, data with weak temporal locality, e.g., long bursts of sequential accesses, are abundant at the flash cache layer, as they can be found in many types of cloud workloads, e.g., when the guest system in a VM performs a weekly backup operation. Caching these data is of little benefit to the application's performance, since their next reuses are too far into the future. Allowing these data to be cached is in fact detrimental to the cache performance, as they evict data blocks that have better temporal locality and are more important to the workload performance. Moreover, they cause unnecessary wear-out to the flash device with little performance gain in return.

To address the limitation of the WS model, we propose a new cache-demand model, *Reuse Working Set*, $RWS_N(t,T)$, which is defined as the set of distinct (address-wise) data blocks that a workload has *reused* at least N times during a time interval $[t-T,t]$. Compared to the WS model, RWS captures only the data blocks with a temporal locality that will benefit the workload's cache hit ratio. When $N=0$ RWS reduces to WS. We then propose to use *Reuse Working Set Size* (*RWSS*) as the estimate of the workload's cache demand. Because RWSS disregards low-locality data, it has the potential to more accurately capture the workload's actual cache demand, and reduce the cache pollution and unnecessary wear-out caused by such data references.

To confirm the effectiveness of the RWS model, we analyze the MSR Cambridge traces [4] with different values of N and evaluate the impact on cache hit ratio, cache usage—the number of cached blocks vs. the number of IOs received by cache, and flash write ratio—the number of writes sent to cache device vs. the number of IOs received by cache. We assume that a data block is admitted into the cache only after it has been accessed N times, i.e., we cache only the workload's RWS_N. Figure 2 shows the distribution of these metrics from the 36 MSR traces using box plots with whiskers showing the quartiles. Increasing N from 0, when we cache the WS, to 1, when we cache the RWS_1, the median hit ratio is reduced by 8%, but the median cache usage is reduced by 82%, and the amount of flash writes is reduced by 19%. This trend continues as we further increase N.

These results confirm the effectiveness of using RWSS to estimate cache demand—it is able to substantially reduce a workload's cache usage and its induced wear-out at a small cost of hit ratio. A system administrator can balance performance against cache usage and endurance by choosing the appropriate N for the RWS model. In general, N = 1 or 2 gives the best tradeoff between these objectives. (Similar observations can be made for the traces listed in Table 1.) In the rest of this paper, we use N = 1 for RWSS-based cache allocation. Moreover, when considering a cloud usage scenario where a shared cache cannot fit the working-sets of all the workloads, using the RWS model to allocate cache capacity can achieve better performance because it prevents the low-locality data from flushing the useful data out of the cache.

In order to measure the RWSS of a workload, we need to determine the appropriate time window to observe the workload. There are two relevant questions here. First, how to track the window? In the original definition of process WS [12], the window is set with respect to the process time, i.e., the number of accesses made by the process, instead of real time. However, it is difficult to use the number of accesses as the window to measure a VM's WS or RWSS at the flash cache layer, because the VM can go idle for a long period of time and never fill up its window, causing the previously allocated cache space to be underutilized. Therefore, we use real-time-based window to observe a workload's RWSS.

The second question is how to decide the size of the time window. If the window is set too small, the observed RWS cannot capture the workload's current locality, and the measured RWSS underestimates the workload's cache demand. If the window is set too large, it may include the past localities that are not part of the workload's current behavior, and the overestimated RWSS will waste cache space and cause unnecessary wear-out. Our solution to this problem is to profile the workload for a period of time, and simulate the cache hit ratio when we allocate space to the workload based on its RWSS measured using different sizes of windows. We then choose the window at the "knee point" of this

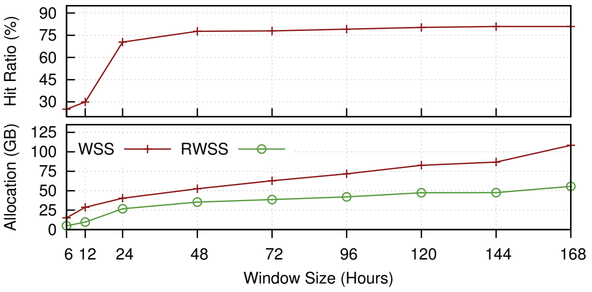


Figure 3: Time window analysis for the Moodle trace

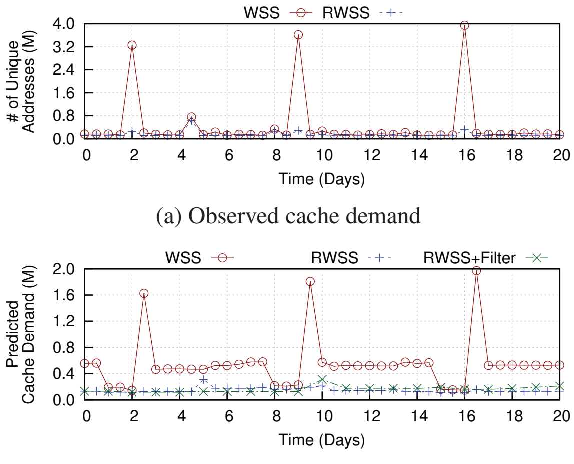
hit ratio vs. window size model, i.e., the point where the hit ratio starts to flatten out. This profiling can be performed periodically, e.g., bi-weekly or monthly, to adjust the choice of window size *online*.

We present an example of estimating the window size using two weeks of the Moodle trace. Figure 3 shows that the hit ratio increases rapidly as the window size increases initially. After the 24-hour window size, it starts to flatten out, while the observed RWSS continues to increase. Therefore, we choose between 24 to 48 hours as the window size for measuring the RWSS of this workload, because a larger window size will not get enough gain in hit ratio to justify the further increase in the workload's cache usage, if we allocate the cache based on the observed RWSS. In case of workloads for which the hit ratio keeps growing slowly with increasing window size but without showing an obvious knee point, the window size should be set to a small value because it will not affect the hit ratio much but can save cache space for other workloads with clear knee points.

4.2 Online Cache Demand Prediction

The success of RWSS-based cache allocation also depends on whether we can accurately predict the cache demand of the next time window based on the RWSS values observed from the previous windows. To address this problem, we consider the classic exponential smoothing and double exponential smoothing methods. The former requires a smoothing parameter α, and the latter requires an additional trending parameter β. The values of these parameters can have a significant impact on the prediction accuracy. We address this issue by using the self-tuning versions of these prediction models, which estimate these parameters based on the error between the predicted and observed RWSS values.

To further improve the robustness of the RWSS prediction, we devise filtering techniques which can dampen the impact of outliers in the observed RWSS values when predicting RWSS. If the currently observed RWSS is λ times greater than the average of the previous n observed values (including the current one), this value is replaced with the average. For example, n is set to 20 and λ is set to 5 in our experiments. In this way, an outlier's impact in the prediction is mitigated.



(a) Observed cache demand

(b) Predicted demand

Figure 4: RWSS-based cache demand prediction

Figure 4 shows an example of the RWSS prediction for three weeks of the Webserver trace. The recurring peaks in the observed WSS in Figure 4a are produced by a weekly backup task performed by the VM, which cause the predicted WSS in Figure 4b to be substantially inflated. In comparison, the RWSS model automatically filters out these backup IOs and the predicted RWSS is only 26% of the WSS on average for the whole trace. The filtering technique further smooths out several outliers (e.g., between Day 4 and 5) which are caused by occasional bursts of IOs that do not reflect the general trend of the workload.

4.3 Cache Allocation and Admission

Based on the cache demands estimated using the RWSS model and prediction methods, the cache allocation to the concurrent VMs is adjusted accordingly at the start of every new time window—the smallest window used to estimate the RWSS of all the VMs. The allocation of cache capacity should not incur costly data copying or flushing. Hence, we consider *replacement-time enforcement* of cache allocation, which does not physically partition the cache across VMs. Instead, it enforces logical partitioning at replacement time: a VM that has not used up its allocated share takes its space back by replacing a block from VMs that have exceeded their shares. Moreover, if the cache is not full, the spare capacity can be allocated to the VMs proportionally to their predicted RWSSes or left idle to reduce wear-out.

The RWSS-based cache allocation approach also requires an *RWSS-based cache admission policy* that admits only reused data blocks into the cache; otherwise, the entire WS will be admitted into the cache space allocated based on RWSS and evict useful data. To enforce this cache admission policy, CloudCache uses a small portion of the main memory as the staging area for referenced addresses, a common strategy for implementing cache admission [33, 15]. A block is admitted into the cache only after it has been accessed *N* times, no matter whether they are reads or writes. The size of the staging area is bounded and when it gets full the staged addresses are evicted using LRU. We refer to this approach of staging only addresses in main memory as *address staging*.

CloudCache also considers a *data staging* strategy for cache admission, which stores both the addresses and data of candidate blocks in the staging area and manages them using LRU. Because main memory is not persistent, so more precisely, only the data returned by read requests are staged in memory, but for writes only their addresses are staged. This strategy can reduce the misses for read accesses by serving them from the staging area before they are admitted into the cache. The tradeoff is that because a data block is much larger than an address (8B address per 4KB data), for the same staging area, data staging can track much less references than address staging and may miss data with good temporal locality.

To address the limitations of address staging and data staging and combine their advantages, CloudCache considers a third *hybrid staging* strategy in which the staging area is divided to store addresses and data, and the address and data partitions are managed using LRU separately. This strategy has the potential to reduce the read misses for blocks with small reuse distances by using data staging and admitting the blocks with relative larger reuse distances by using address staging.

4.4 Evaluation

The rest of this section presents an evaluation of the RWSS-based on-demand cache allocation approach. CloudCache is created upon block-level virtualization by providing virtual block devices to VMs and transparently caching their data accesses to remote block devices accessed across the network (Figure 1). It includes a kernel module that implements the virtual block devices, monitors VM IOs, and enforces cache allocation and admission, and a user-space component that measures and predicts RWSS and determines the cache shares for the VMs. The kernel module stores the recently observed IOs in a small circular buffer for the user-space component to use, while the latter informs the former about the cache allocation decisions. The current implementation of CloudCache is based on Linux and it can be seamlessly deployed as a drop-in solution on Linux-based environments including VM systems that use Linux-based IO stack [8, 2]. We have also created a user-level cache simulator of CloudCache to facilitate the cache hit ratio and flash write ratio analysis, but we use only the real implementation for measuring real-time performance.

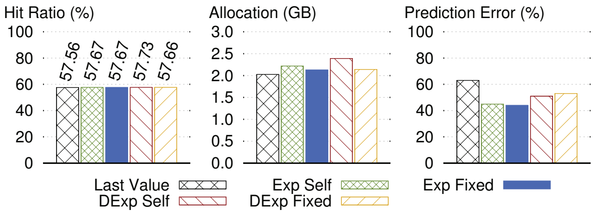


Figure 5: Prediction accuracy

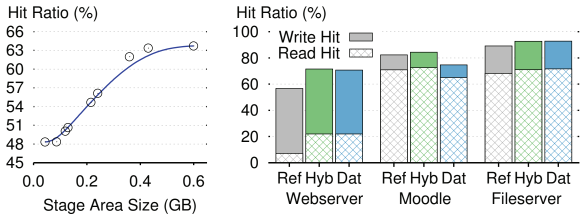
The traces described in Section 3 are replayed on a real iSCSI-based storage system. One node from a compute cluster is set up as the storage server and the others as the clients. Each node has two six-core 2.4GHz Xeon CPUs and 24GB of RAM. Each client node is equipped with the CloudCache modules, as part of the Dom0 kernel, and flash devices (Intel 120GB MLC SATA-interface) to provide caching to the hosted Xen VMs. The server node runs the IET iSCSI server to export the logical volumes stored on a 1TB 7.2K RPM hard disk to the clients via a Gigabit Ethernet. The clients run Xen 4.1 to host VMs, and each VM is configured with 1 vCPU and 2GB RAM and runs kernel 2.6.32. The RWSS window size for the Webserver, Moodle, and Fileserver traces are 48, 24, and 12 hours, respectively. Each VM's cache share is managed using LRU internally, although other replacement policies are also possible.

4.4.1 Prediction Accuracy

In the first set of experiments we evaluate the different RWSS prediction methods considered in Section 4.2: (1) *Exp fixed*, exponential smoothing with $\alpha = 0.3$, (2) *Exp self*, a self-tuning version of exponential smoothing, (3) *DExp fixed*, double-exponential smoothing with $\alpha = 0.3$ and $\beta = 0.3$, (4) *DExp self*, a self-tuning version of double-exponential smoothing, and (5) *Last value*, a simple method that uses the last observed RWSS value as predicted value for the new window.

Figure 5 compares the different prediction methods using three metrics: (1) *hit ratio*, (2) *cache allocation*, and (3) *prediction error*—the absolute value of the difference between the predicted RWSS and observed RWSS divided by the observed RWSS. Prediction error affects both of the other two metrics—under-prediction increases cache misses and over-prediction uses more cache. The figure shows the average values of these metrics across all the time windows of the entire 9-month Webserver trace.

The results show that the difference in hit ratio is small among the different prediction methods but is considerable in cache allocation. The last value method has the highest prediction error, which confirms the need of prediction techniques. The exponential smoothing methods have the lowest prediction errors, and *Exp Self* is more preferable because it automatically trains its parameter. We believe that more advanced prediction methods are possible to further improve the prediction accuracy and



(a) Staging area size (b) Staging strategies

Figure 6: Staging strategy analysis

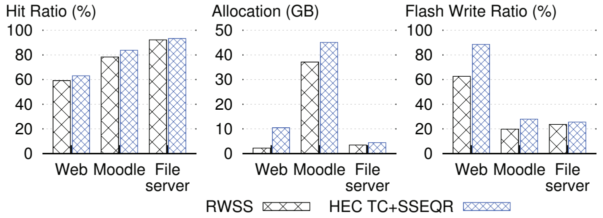


Figure 7: Comparison to HEC

our solution can be extended to run multiple prediction methods at the same time and choose the best one at runtime. But this simple smoothing-based method can already produce good results, as shown in the following experiments which all use *Exp Self* to predict cache demand.

4.4.2 Staging Strategies

In the second set of experiments, we evaluate CloudCache's staging strategies. First, we study the impact of the staging area size. In general, it should be decided according to the number of VMs consolidated to the same cache and the IO intensity of their workloads. Therefore, our approach is to set the total staging area size as a percentage, e.g., between 0.1% and 1%, of the flash cache size, and allocate the staging area to the workloads proportionally to their flash cache allocation. Figure 6a gives an example of how the staging area allocation affects the Webserver workload's hit ratio when using address staging. The results from data staging are similar. In the rest of the paper, we always use 256MB as the total staging area size for RWSS-based cache allocation. Note that we need 24B of the staging space for tracking each address, and an additional 4KB if its corresponding data is also staged.

Next we compare the address, data, and hybrid staging (with a 1:7 ratio between address and data staging space) strategies with the same staging area size in Figure 6b. Data staging achieves a better read hit ratio than address staging by 67% for the Webserver trace but it loses to address staging by 9% for Moodle. These results confirm our discussion in Section 4.3 about the tradeoff between these strategies. In comparison, the hybrid staging combines the benefits of these two and is consistently the best for all traces. We have tested different ratios for hybrid staging, and our results show that the hit ratio difference is small (<1%). But a larger address staging area tracks a longer history and admits more data into the cache, which results in more cache usage and flash writes. Therefore, in the rest of this paper, we always use hybrid staging with 1:7 ratio between address and data staging space for RWSS-based allocation.

We also compare to the related work High Endurance Cache (HEC) [33] which used two cache admission techniques to address flash cache wear-out and are closely related to our staging strategies. HEC's Touch Count (TC) technique uses an in-memory bitmap to track all the cache blocks (by default 4MB) and admit only reused blocks into cache. In comparison, CloudCache tracks only a small number of recently accessed addresses to limit the memory usage and prevent blocks accessed too long ago from being admitted into cache. HEC's Selective Sequential Rejection (SSEQR) technique tracks the sequentiality of accesses and rejects long sequences (by default any longer-than-4MB sequence). In comparison, CloudCache uses the staging area to automatically filter out long scan sequences.

Because HEC did not consider on-demand cache allocation, we implemented it by using TC to predict cache demand and using both TC and SSEQR to enforce cache admission. Figures 7 shows the comparison using the different traces, which reveals that on average HEC allocates up to 3.7x more cache than our RWSS-based method and causes up to 29.2% higher flash write ratio—the number of writes sent to cache device vs. the number of IOs received by cache. In return, it achieves only up to 6.4% higher hit ratio. The larger cache allocation in HEC is because it considers all the historical accesses when counting reuses, whereas the RWSS method considers only the reuses occurred in the recent history—the previous window. (If we were able to apply the same cache allocation given by the RWSS method while using HEC's cache admission method, we would achieve a much lower hit ratio, e.g., 68% lower for Moodle, and still a higher flash write ratio, e.g., 69% higher for Moodle.) The result also confirms that the RWSS method is able to automatically reject scan sequences (e.g., it rejects on average 90% of the IOs during the backup periods), whereas HEC needs to explicitly detect scan sequences using a fixed threshold.

4.4.3 WSS vs. RWSS-based Cache Allocation

In the third set of experiments, we compare RWSS-based to WSS-based cache allocation using the same prediction method, exponential smoothing with self-tuning. In both cases, the cache allocation is strictly enforced, and at the start of each window, the workload's extra cache usage beyond its new allocation is immediately dropped. This setting produces the *worst-case* result for on-demand cache allocation, because in practice Cloud-

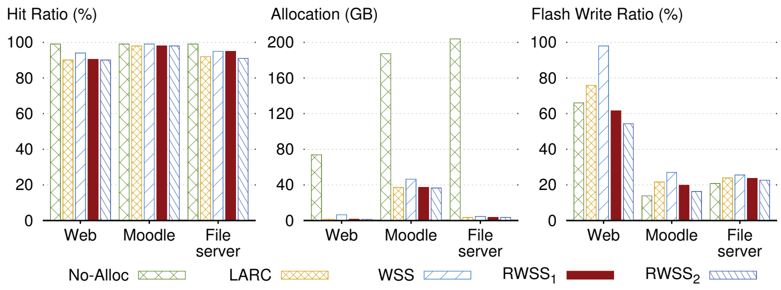
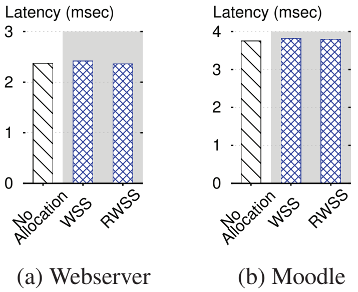


Figure 8: Allocation methods



(a) Webserver

(b) Moodle

Figure 9: VM IO latency comparison

Cache allows a workload to use spare capacity beyond its allocation and its extra cache usage is gradually reclaimed via replacement-time enforcement. We also include the case where the workload can use up the entire cache as a baseline (*No Allocation*), where the cache is large enough to hold the entire working set and does not require any replacement.

Figure 8 shows the comparison among these different approaches. For RWSS, we consider two different values for the N in RWS_N, as described in Section 4.1. In addition, we also compare to the related cache admission method, LARC [15], which dynamically changes the size of the staging area according to the current hit ratio—a higher hit ratio reduces the staging area size. Like HEC, LARC also does not provide on-demand allocation, so we implemented it by using the number of reused addresses to predict cache demand and using LARC for cache admission.

$RWSS_1$ achieves a hit ratio that is only 9.1% lower than *No Allocation* and 4% lower than *WSS*, but reduces the workload's cache usage substantially by up to 98% compared to *No Allocation* and 76% compared to *WSS*, and reduces the flash write ratio by up to 6% compared to *No Allocation* and 37% compared to *WSS*. (The cache allocation of RWSS and LARC is less than 4GB for Webserver and Fileserver and thus barely visible in the figure). *No Allocation* has slightly lower flash write ratio than $RWSS_1$ for Moodle and Fileserver only because it does not incur cache replacement, as it is allowed to occupy as much cache space as possible, which is not a realistic scenario for cloud environments. Compared to *LARC*, $RWSS_1$ achieves up to 3% higher hit ratio and still reduces cache usage by up to 3% and the flash writes by up to 18%, while using 580MB less staging area on average. Comparing the two different configurations of RWSS, $RWSS_2$ reduces cache usage by up to 9% and flash writes by up to 18%, at the cost of 4% lower hit ratio, which confirms the tradeoff of choosing different values of N in our proposed RWS model.

To evaluate how much performance loss the hit ratio reduction will cause, we replay the traces and measure their IO latencies. We consider a one-month portion of the Webserver and Moodle traces. They were replayed on the real VM storage and caching setup specified in Section 4.4. We compare the different cache management methods in terms of 95th percentile IO latency. Figure 9 shows that the *RWSS*-based method delivers the similar performance as the alternatives (only 1% slower than *No Allocation* for Moodle) while using much less cache and causing more writes to the cache device as shown in the previous results.

The results confirm that our proposed RWSS-based cache allocation can indeed substantially reduce a workload's cache usage and the corresponding wear-out at only a small performance cost. In real usage scenarios our performance overhead would be much smaller because a workload's extra cache allocation does not have to be dropped immediately when a new time window starts and can still provide hits while being gradually replaced by the other workloads. Moreover, because the WSS-based method requires much higher cache allocations for the same workloads, cloud providers have to either provision much larger caches, which incurs more monetary cost, or leave the caches oversubscribed, which leads to bad performance as the low-locality data are admitted into the cache and flush out the useful data.

5 Dynamic Cache Migration

The *on-demand cache allocation* approach discussed in the previous section allows CloudCache to estimate the cache demands of workloads online and dynamically allocate the shared capacity to them. To handle scenarios where the cache capacity is insufficient to meet all the demands, this section presents the *dynamic cache migration* approach to balance the cache load across different hosts by dynamically migrating a workload's cached data along with its VM. It also considers techniques to optimize the performance for the migrating VM as well as minimize the impact to the others during the migration.

5.1 Live Cache Migration

Live VM migration allows a workload to be transparently migrated among physical hosts while running in its VM [11, 25]. In CloudCache, we propose to use live

VM migration to balance the load on the flash caches of VM hosts—when a host's cache capacity becomes insufficient to meet the local VMs' total cache demands (as estimated by their predicted RWSSes), some VMs can be migrated to other hosts that have spare cache capacity to meet their cache demands.

VM-migration-based cache load balancing presents two challenges. First, the migrating VM's dirty cache data on the migration *source* host must be synchronized to the *destination* host before they can be accessed again by the VM. A naive way is to flush all the dirty data to the remote storage server for the migrating VM. Depending on the amount of dirty data and the available IO bandwidth, the flushing can be time consuming, and the VM cannot resume its activity until the flushing finishes. The flushing will also cause a surge in the storage server's IO load and affect the performance of the other VMs sharing the server. Second, the migrating VM needs to warm up the cache on the destination host, which may also take a long time, and it will experience substantial performance degradation till the cache is warmed up [14, 7].

To address these challenges, CloudCache's dynamic cache migration approach uses a combination of reactive and proactive migration techniques:

On-Demand Migration: When the migrated VM accesses a block that is dirty in the source host's cache, its local cache forwards the request to the source host and fetches the data from there, instead of the remote storage server. The metadata of the dirty blocks, i.e., their logical block addresses, on the source host are transferred along with VM migration, so the destination host's local cache is aware of which blocks are dirty on the source host. Because the size of these metadata is small (e.g., 8B per 4KB data), the metadata transfer time is often negligible. It is done before the VM is activated on the destination, so the VM can immediately use the cache on the destination host.

Background Migration: In addition to reactively servicing requests from the migrated VM, the source host's cache also proactively transfers the VM's cached data—its RWS—to the destination host. The transfer is done in background to mitigate the impact to the other VMs on the source host. This background migration allows the destination host to quickly warm up its local cache and improve the performance of the migrated VM. It also allows the source host to quickly reduce its cache load and improve the performance of its remaining VMs. Benefiting from the RWSS-based cache allocation and admission, the data that need to be transferred in background contain only the VM's RWS which is much smaller than the WS, as shown in the previous section's results. Moreover, when transferring the RWS, the blocks are sent in the decreasing order of their recency so the data that are most likely to used next are transferred earliest.

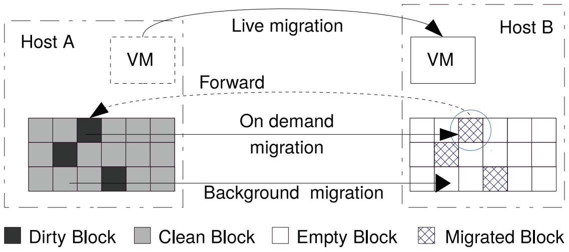
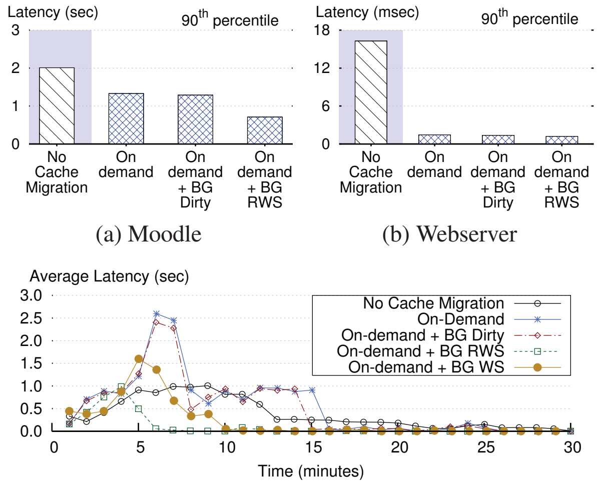


Figure 10: Architecture of dynamic cache migration

On-demand migration allows the migrated VM to access its dirty blocks quickly, but it is inefficient for transferring many blocks. Background migration can transfer bulk data efficiently but it may not be able to serve the current requests that the migrated VM is waiting for. Therefore, the combination of these two migration strategies can optimize the performance of the VM. Figure 10 illustrates how CloudCache performs cache migration. When a VM is live-migrated from Host *A* to Host *B*, to keep data consistent while avoiding the need to flush dirty data, the cached metadata of dirty blocks are transferred to Host *B*. Once the VM live migration completes, the VM is activated on Host *B* and its local flash cache can immediately service its requests. By using the transferred metadata, the cache on Host *B* can determine whether a block is dirty or not and where it is currently located. If a dirty block is still on Host *A*, a request is sent to fetch it on demand. At the same time, Host *A* also sends the RWS of the migrated VM in background. As the cached blocks are moved from Host *A* to Host *B*, either on-demand or in background, Host A vacates their cache space and makes it available to the other VMs.

The CloudCache module on each host handles both the operations of local cache and the operations of cache migration. It employs a multithreaded design to handle these different operations with good concurrency. Synchronization among the threads is needed to ensure consistency of data. In particular, when the destination host requests a block on demand, it is possible that the source host also transfers this block in background, at the same time. The destination host will discard the second copy that it receives, because it already has a copy in the local cache and it may have already overwritten it. As an optimization, a write that aligns to the cache block boundaries can be stored directly in the destination host's cache, without fetching its previous copy from the source host. In this case, the later migrated copy of this block is also discarded. The migrating VM needs to keep the same device name for its disk, which is the virtual block device presented by CloudCache's block-level virtualization. CloudCache assigns unique names to the virtual block devices based on the unique IDs of the VMs in the cloud system. Before migration, the mapping from the virtual block device to physical device (e.g., the iSCSI device) is created on the destination host, and after migration, the counterpart on the source host is removed.



(a) Moodle

(b) Webserver

(c) The migrating VM's performance (average IO latency per minute) for Moodle. The migration starts at the 5th minute.

Figure 11: Migration strategies

5.2 Migration Rate Limiting

While the combination of on-demand and background migrations can optimize the performance of a migrating VM, the impact to the other VMs on the source and destination hosts also needs to be considered. Cache migration requires reads on the source host's cache and writes to the destination host's cache, which can slow down the cache IOs from the other co-hosted VMs. It also requires network bandwidth, in addition to the bandwidth already consumed by VM memory migration (part of the live VM migration [11, 25]), and affects the network IO performance of the other VMs.

In order to control the level of performance interference to co-hosted VMs, CloudCache is able to limit the transfer rate for cache migration. Given the rate limit, it enforces the maximum number of data blocks that can be transferred from the source host to the destination host every period of time (e.g., 100ms), including both on-demand migration and background migration. Once the limit is hit, the migration thread will sleep and wait till the next period to continue the data transfer. If on-demand requests arrive during the sleep time, they will be delayed and served immediately after the thread wakes up. The rate can be set based on factors including the priority of the VMs and the RWSS of the migrating VM. CloudCache allows a system administrator to tune the rate in order to minimize the cache migration impact to the co-hosted VMs and still migrate the RWS fast enough to satisfy the cache demands.

5.3 Evaluation

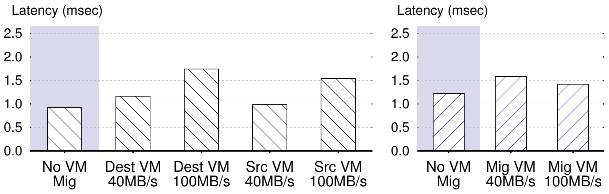
We evaluate the performance of CloudCache's dynamic cache migration using the same testbed described in Section 4.4. Dynamic cache migration is implemented in the CloudCache kernel module described in Section 4.4. It exposes a command-line interface which is integrated with virt-manager [3] for coordinating VM migration with cache migration. We focus on a day-long portion of the Moodle and Webserver traces. The Moodle one-day trace is read-intensive which makes 15% of its cached data dirty (about 5GB), and the Webserver one-day trace is write-intensive which makes 85% of its cached data dirty (about 1GB).

We consider four different approaches: (1) *No Cache Migration:* the cached data on the source host are not migrated with the VM; (2) *On-demand:* only the on-demand cache migration is used to transfer dirty blocks requested by the migrated VM; (3) *On-demand + BG Dirty:* in addition to on-demand cache migration, background migration is used to transfer only the dirty blocks of the migrated VM; (4) *On-demand + BG RWS:* both on-demand migration of dirty blocks and background migration of RWS are used. In this experiment, we assume that the cache migration can use the entire 1Gbps network bandwidth, and we study the impact of rate limiting in the next experiment. For on-demand cache migration, it takes 0.3s to transfer the metadata for the Moodle workload and 0.05s for the Webserver workload.

Figure 11a shows that for the Moodle workload, on-demand cache migration decreases the 90th percentile latency by 33% and the addition of background migration of dirty data decreases it by 35%, compared to *No Cache Migration*. However, the most significant improvement comes from the use of both on-demand migration of dirty data and background migration of the entire RWS, which reduces the latency by 64%. The reason is that this workload is read-intensive and reuses a large amount of clean data; background migration of RWS allows the workload to access these data from the fast, local flash cache, instead of paying the long network latency for accessing the remote storage.

For the Webserver workload, because its RWS is mostly dirty, the difference among the three cache migration strategies is smaller than the Moodle workload (Figure 11b). Compared to the *No Cache Migration* case, they reduce the 90th percentile latency by 91.1% with on-demand migration of dirty data, and by 92.6% with the addition of background migration of RWS.

Note that the above results for the *No Cache Migration* case do not include the time that the migrated VM has to wait for its dirty data to be flushed from the source host to the remote storage before it can resume running again, which is about 54 seconds for the Moodle workload and 12 seconds for the Webserver workload, assuming it can use all the bandwidths of the network and storage server.



(a) Co-hosted VMs (b) Migrating VM

Figure 12: Impact of different cache migration rate

In comparison, the VM has zero downtime when using our dynamic cache migration.

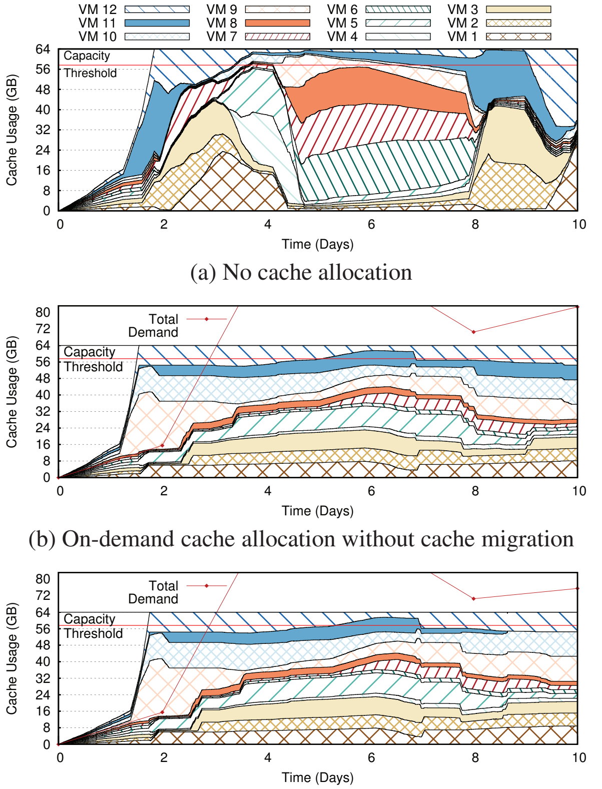
Figure 11c shows how the migrating VM's performance varies over time in this Moodle experiment so we can observe the real-time performance of the different migration strategies. The peaks in *On-demand* and *On-demand + BG Dirty* are caused by bursts of on-demand transfer of clean data blocks requested by the migrated VM. We believe that we can further optimize our prototype implementation to avoid such spikes in latency.

In Figure 11c, we also compare our approach to an alternative cache migration implementation (*On-demand + BG WS*) which migrates the VM's entire working set without the benefit of our proposed RWS model. Using the same Moodle trace, at the time of migration, its RWSS is 32GB and WSS is 42GB. As a result, migrating the WS takes twice the time of migrating only the RWS (6mins vs. 3mins) and causes a higher IO latency overhead too (71% higher in 90th percentile latency).

In the next experiment, we evaluate the performance impact of rate limiting the cache migration. In addition to the migrating VM, we run another IO-intensive VM on both the source and destination hosts, which replays a different day-long portion of the Webserver trace. We measure the performance of all the VMs when the cache migration rate is set at 40MB/s and 100MB/s and compare to their normal performance when there is no VM or cache migration. Figure 12 shows that the impact to the co-hosted VMs' 90th percentile IO latency is below 16% and 21% for the 40MB/s and 100MB/s rate respectively. Note that this is assuming that the co-hosted VMs already have enough cache space, so in reality, their performance would actually be much improved by using the cache space vacated from the migrating VM. Meanwhile, the faster migration rate reduces the migrating VM's 90th percentile IO latency by 6%. Therefore, the lower rate is good enough for the migrating VM because the most recently used data are migrated first, and it is more preferable for its lower impact to the co-hosted VMs.

6 Putting Everything Together

The previous two sections described and evaluated the RWSS-based on-demand cache allocation and dynamic cache migration approaches separately. In this section, we present how to use them together to realize on-demand cache management for multiple VM hosts. Consider the flash cache on a single host. If its capacity is sufficient to satisfy the predicted cache demands for all the local VMs, it is simply allocated to the VMs according to their demands. The spare capacity is distributed to the VMs proportionally to their demands, or left idle to minimize wear-out. If the cache capacity is not sufficient, then cache migration needs to be considered in order to satisfy the demands of all the VMs.



(a) No cache allocation

(b) On-demand cache allocation without cache migration

(c) On-demand cache allocation with dynamic cache migration

Figure 13: Cache usages of 12 concurrent VMs

When considering the use of cache migration, there are three key questions that need to be answered, *when to migrate*, *which VM to migrate*, and *which host to migrate it to?* To answer the first question, CloudCache reserves a certain percentage (e.g., 10%) of the cache capacity as a buffer to absorb the occasional surges in cache demands, and it starts a migration when the total cache demand exceeds the 90% threshold for several consecutive RWSS windows (e.g., three times). This approach prevents the fluctuations in cache workloads from triggering unnecessary cache migrations which affect the VMs' performance and the system's stability.

To answer the second and third questions, CloudCache's current strategy is to minimize the imbalance of cache load among the hosts in the system. The host that requires cache migration queries every other host's current cache load. It then evaluates all the possible migration plans of moving one of its local VMs to a host that can accommodate the VM's RWS under the 90% threshold. It then chooses the plan that minimizes the variance of the hosts' cache load distribution.

We use a real experiment to illustrate the use of our approaches for meeting the cache demands of dynamically changing workloads. We consider two VM hosts each with 64GB of flash cache. Host *A* ran 12 VMs, and Host *B* ran three VMs, concurrently. Each VM replayed a different 10-day portion of the Webserver trace. The cache allocation was adjusted every 2 days on both hosts. The first time window is the warm-up phase during which the VMs were given equal allocation of the cache capacity. Afterwards, the cache was allocated to the VMs proportionally to their estimated RWSSes. Moreover, a VM could take more than its share if there was idle capacity from the other VMs' shares because our approach is work-conserving. The experiment was done on the real VM storage and caching setup specified in Section 4.4.

Figure 13a shows how the cache space is distributed among the VMs on Host A when *(a)* there is no cache allocation, *(b)* on-demand cache allocation but without cache migration, and *(c)* on-demand cache allocation with dynamic cache migration. Comparing (a) and (b), we can see how our RWSS-based on-demand allocation improves the fairness among the competing VMs. For example, between Days 4 and 8, VMs 6, 7, 8 dominated the cache space in (a), but in (b), every VM got a fair share of the cache space proportionally to their estimated RWSSes. Notice that VMs 7 and 8 were allocated much less in (b) than what they got in (a), which is an evidence of how the RWS-based cache demand model filtered out the VMs' low-locality data and kept only those that are useful to their performance. As a result, comparing the average performance of all 12 VMs across the entire experiment, (b) is better than (a) by 17% in terms of hit ratio and 13% in terms of 90th percentile IO latency.

In (c) dynamic cache migration was enabled in addition to on-demand cache allocation. After the total demand—the sum of the 12 VMs' RWSSes—exceeded the threshold for three consecutive windows, CloudCache initiated cache migration on Day 8 and chose to move VM 11, the one with the largest predicted RWSS at that time, and its cached data to Host *B*. As VM 11's RWS was moved to Host *B*, the remaining 11 VMs took over the whole cache on Host *A*, proportionally to their estimated RWSSes. As a result, comparing the average performance of all 12 VMs after Day 8, (c) is better than (b) by 49% in terms of hit ratio and 24% in terms of 90th percentile IO latency. Across the entire experiment, it outperforms (a) by 28% in hit ratio and 27% in 90th percentile IO latency, and outperforms (b) by 10% in hit ratio and 16% in 90th percentile IO latency.

Although this experiment involved only two VM hosts and the migration of only one VM, the above results are still representative for the migration of any VM and its cache data between two hosts in a large cloud computing environment. But we understand in such a large environment, more intelligence is required to make the optimal VM migration decisions. There is a good amount of related work (e.g., [31, 32]) on using VM migration to balance load on CPUs and main memory and to optimize performance, energy consumption, etc. CloudCache is the first to consider on-demand flash cache management across multiple hosts, and it can be well integrated into these related solutions to support the holistic management of different resources and optimization for various objectives. We leave this to our future work because the focus of this paper is on the key mechanisms for on-demand cache management, i.e., on-demand cache allocation and dynamic cache migration, which are missing in existing flash cache management solutions and are non-trivial to accomplish.

7 Related Work

There are several related flash cache management solutions. S-CAVE [22] considers the number of reused blocks when estimating a VM's cache demand, and allocates cache using several heuristics. vCacheShare [24] allocates a read-only cache by maximizing a unity function that captures a VM's disk latency, read-to-write ratio, estimated cache hit ratio, and reuse rate of the allocated cache capacity. Centaur [18] uses MRC and latency curves to allocate cache to VMs according to their QoS targets. However, these solutions admit all referenced data into cache, including those with weak temporal locality, and can cause unnecessary cache usage and wear-out. Moreover, none of them considers dynamic cache migration for meeting the demands when a cache is overloaded. These problems are addressed by CloudCache's on-demand cache allocation and dynamic cache migration approaches.

HEC and LARC studied cache admission policies to filter out data with weak temporal locality and reduce the flash wear-out [33, 15]. However, they do not consider the problem of how to allocate shared cache capacity to concurrent workloads, which is addressed by CloudCache. Moreover, our RWSS-based approach is able to more effectively filter out data with no reuses and achieve good reduction in cache footprint and wear-out, as shown in Section 4.4.

Bhagwat *et al.* studied how to allow a migrated VM to request data from the cache on its previous host [9], in the same fashion as the on-demand cache migration proposed in this paper. However, as shown in Section 5.3, without our proposed background cache migration, on-demand migration alone cannot ensure good performance for the migrated VM. It also has a long-lasting, negative impact on the source host in terms of both performance interference and cache utilization. When the migrated VM's data are evicted on the source host, the performance becomes even worse because a request has to be forwarded by the source host to the primary storage. VMware's vSphere flash read cache [6] also supports background cache migration. Although its details are unknown, without our proposed RWS model, a similar solution would have to migrate the VM's entire cache footprint. As shown in Section 5.3, this requires longer migration time and causes higher impact to performance. In comparison, CloudCache considers the combination of on-demand migration and background migration and is able to minimize the performance impact to both the migrated VM and the other co-hosted VMs.

In the context of processor and memory cache management, ARC addresses the cache pollution from scan sequences by keeping such data in a separate list (*T1*) and preventing them from flooding the list (*T2*) of data with reuses [23]. However, data in *T1* still occupy cache space and cause wear-out. Moreover, it does not provide answers to how to allocate shared cache space to concurrent workloads. Related work [19] proposed the model of effective reuse set size to capture the necessary cache capacity for preventing non-reusable data from evicting reusable data, but it assumes that all data have to be admitted into the cache. There are also related works on processor and memory cache allocations. For example, miss-rate curve (MRC) can be built to capture the relationship between a workload's cache hit ratio and its cache sizes, and used to guide cache allocation [27, 34, 28, 29, 30]. Process migration was also considered for balancing processor cache load on a multi-core system [16].

Compared to these processor and memory caching works, flash cache management presents a different set of challenges. Low-locality data are detrimental to not only a flash cache's performance but also its lifetime, which unfortunately are abundant at the flash cache layer. While VM migration can be used to migrate workloads across hosts, the large amount of cached data cannot be simply flushed or easily shipped over. CloudCache is designed to address these unique challenges by using RWSS to allocate cache to only data with good locality and by providing dynamic cache migration with techniques to minimize its impact to VM performance.

8 Conclusions and Future Work

Flash caching has great potential to address the storage bottleneck and improve VM performance for cloud computing systems. Allocating the limited cache capacity to concurrent VMs according to their demands is key to making efficient use of flash cache and optimizing VM performance. Moreover, flash devices have serious endurance issues, whereas weak-temporal-locality data are abundant at the flash cache layer, which hurt not only the cache performance but also its lifetime. Therefore, on-demand management of flash caches requires fundamental rethinking on how to estimate VMs' cache demands and how to provision space to meet their demands.

This paper presents CloudCache, an on-demand cache management solution to these problems. First, it employs a new cache demand model, Reuse Working Set (RWS), to capture the data with good temporal locality, allocate cache space according to the predicted Reuse Working Set Size (RWSS), and admit only the RWS into the allocated space. Second, to handle cache overload situations, CloudCache takes a new cache migration approach which live-migrates a VM with its cached data to meet the cache demands of the VMs. Extensive evaluations based on real-world traces confirm that the RWSS-based cache allocation approach can achieve good cache hit ratio and IO latency for a VM while substantially reducing its cache usage and flash wear-out. It also confirms that the dynamic cache migration approach can transparently balance cache load across hosts with small impact to the migrating VM and the other co-hosted VMs.

CloudCache provides a solid framework for our future work in several directions. First, we plan to use flash simulators and open-controller devices to monitor the actual Program/Erase cycles and provide more accurate measurement of our solution's impact on flash device wear-out. Second, when the aggregate cache capacity from all VM hosts is not sufficient, CloudCache has to allocate cache proportionally to the VMs' RWSSes. We plan to investigate a more advanced solution which maps each VM's cache allocation to its performance and optimizes the allocation by maximizing the overall performance of all VMs. Third, although our experiments confirm that flash cache allocation has a significant impact on application performance, the allocation of other resources, e.g., CPU cycles and memory capacity, is also important. We expect to integrate existing CPU and memory management techniques with CloudCache to provide a holistic cloud resource management solution. Finally, while the discussion in this paper focuses on flash-memory-based caching, CloudCache's general approach is also applicable to emerging NVM devices, which we plan to evaluate when they become available.

9 Acknowledgements

We thank the anonymous reviewers and our shepherd, Carl Waldspurger, for the thorough reviews and insightful suggestions. This research is sponsored by National Science Foundation CAREER award CNS-125394 and Department of Defense award W911NF-13-1-0157.

References

[1] Fusion-io ioCache. http://www.fusionio.com/products/iocache/.

[2] Kernel Based Virtual Machine. http://www.linux-kvm.org/page/Main_Page.

[3] Manage virtual machines with virt-manager. https://virt-manager.org.

[4] MSR cambridge traces. http://iotta.snia.org/traces/388.

[5] Network Block Device. http://nbd.sourceforge.net/.

[6] Performance of vSphere flash read cache in VMware vSphere 5.5. https://www.vmware.com/files/pdf/techpaper/vfrc-perf-vsphere55.pdf.

[7] D. Arteaga and M. Zhao. Client-side flash caching for cloud systems. In *Proceedings of International Conference on Systems and Storage (SYSTOR 14)*, pages 7:1–7:11. ACM, 2014.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP 03)*. ACM, 2003.

[9] D. Bhagwat, M. Patil, M. Ostrowski, M. Vilayannur, W. Jung, and C. Kumar. A practical implementation of clustered fault tolerant write acceleration in a virtualized environment. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 287–300, Santa Clara, CA, 2015. USENIX Association.

[10] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In *Proceedings of the 28th IEEE Conference on Massive Data Storage (MSST 12)*, Pacific Grove, CA, USA, 2012. IEEE.

[11] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation (NSDI 05)*, pages 273–286. USENIX Association, 2005.

[12] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[13] E. V. Hensbergen and M. Zhao. Dynamic policy disk caching for storage networking. Technical Report RC24123, IBM, November 2006.

[14] D. A. Holland, E. L. Angelino, G. Wald, and M. I. Seltzer. Flash caching on the storage client. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC 13)*. USENIX Association, 2013.

[15] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with lazy adaptive replacement. In *Proceedings of the 29th IEEE Symposium on Mass Storage Systems and Technologies (MSST 13)*, pages 1–10. IEEE, 2013.

[16] R. C. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.

[17] R. Koller, L. Marmol, R. Ranganswami, S. Sundararaman, N. Talagala, and M. Zhao. Write policies for host-side flash caches. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST 13)*, 2013.

[18] R. Koller, A. J. Mashtizadeh, and R. Rangaswami. Centaur: Host-side SSD caching for storage performance control. In *Proceedings of the 2015 IEEE International Conference on Autonomic Computing (ICAC 15), Grenoble, France, July 7-10, 2015*, pages 51–60, 2015.

[19] R. Koller, A. Verma, and R. Rangaswami. Generalized ERSS tree model: Revisiting working sets. *Performance Evaluation*, 67(11):1139–1154, Nov. 2010.

[20] M. Krueger, R. Haagens, C. Sapuntzakis, and M. Bakke. Small computer systems interface protocol over the internet (iSCSI): Requirements and design considerations. Internet RFC 3347, July 2002.

[21] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC 14)*, pages 501–512. USENIX Association, 2014.

[22] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou. S-CAVE: Effective SSD caching to improve virtual machine storage performance. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT 13)*, pages 103–112. IEEE Press, 2013.

[23] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies (FAST 03)*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[24] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu. vCacheShare: Automated server flash cache space management in a virtualization environment. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC 14)*, pages 133–144, Philadelphia, PA, June 2014. USENIX Association.

[25] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the USENIX Annual Technical Conference (ATC 05)*, pages 391–394. USENIX, 2005.

[26] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):134–154, 1988.

[27] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (ICPADS 01)*, pages 116–127, 2001.

[28] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 09)*, pages 121–132, New York, NY, USA, 2009. ACM.

[29] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, Feb. 2015. USENIX Association.

[30] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 335–349, Broomfield, CO, Oct. 2014. USENIX Association.

[31] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI 07)*, pages 17–17, Berkeley, CA, USA, 2007. USENIX Association.

[32] J. Xu and J. Fortes. A multi-objective approach to virtual machine management in datacenters. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC 11)*, pages 225–234, New York, NY, USA, 2011. ACM.

[33] J. Yang, N. Plasson, G. Gillis, and N. Talagala. HEC: improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR 13)*, page 10. ACM, 2013.

[34] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 04)*, pages 177–188, New York, NY, USA, 2004. ACM.