Proceedings of FAST '17: 15th USENIX Conference on File and Storage Technologies

February 27–March 2, 2017
Santa Clara, CA

Conference Organizers

Program Co-Chairs

Geoff Kuenning, *Harvey Mudd College*

Carl Waldspurger, *CloudPhysics*

Program Committee

Nitin Agrawal, *Samsung Research*

Irfan Ahmad, *CloudPhysics*

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

Mahesh Balakrishnan, *Yale University*

André Brinkmann, *Johannes Gutenberg University Mainz*

Haibo Chen, *Shanghai Jiao Tong University*

Peter Desnoyers, *Northeastern University*

Ashvin Goel, *University of Toronto*

Ajay Gulati, *ZeroStack*

Danny Harnik, *IBM Research—Haifa*

Kimberly Keeton, *Hewlett Packard Labs*

Ricardo Koller, *IBM Research*

Sungjin Lee, *Inha University*

Tudor Marian, *Google*

Marshall Kirk McKusick, *McKusick Consultancy*

Arif Merchant, *Google*

Brad Morrey, *Hewlett Packard Labs*

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

Raju Rangaswami, *Florida International University*

Erik Riedel, *Dell Technologies*

Jiri Schindler, *Simplivity*

Bianca Schroeder, *University of Toronto*

Philip Shilane, *Dell EMC*

Keith A. Smith, *NetApp*

Swaminathan Sundararaman, *Parallel Machines*

Vasily Tarasov, *IBM Research*

Eno Thereska, *Confluent and Imperial College London*

An-I Andy Wang, *Florida State University*

Hakim Weatherspoon, *Cornell University*

Sage Weil, *Red Hat*

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

Gala Yadgar, *Technion—Israel Institute of Technology*

Erez Zadok, *Stony Brook University*

Ming Zhao, *Arizona State University*

Work-in-Progress/Posters Co-Chairs

Irfan Ahmad, *CloudPhysics*

Theodore Wong, *Human Longevity, Inc.*

Steering Committee

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

William J. Bolosky, *Microsoft Research*

Angela Demke Brown, *University of Toronto*

Greg Ganger, *Carnegie Mellon University*

Garth Gibson, *Carnegie Mellon University*

Casey Henderson, *USENIX Association*

Kimberly Keeton, *HP Labs*

Florentina Popovici, *Google*

Erik Riedel, *Dell Technologies*

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*

Tutorial Coordinators

John Strunk, NetApp

Eno Thereska, *Confluent and Imperial College London*

External Reviewers

Dulcardo Arteaga

Saman Biookaghazadeh

Jorge Cabrera

Peter Corbett

Runyu Jin

Cheng Li

Wenji Li

Lanyue Lu

Thanu Pillai

Michel Roger

Kishore Udayashankar

Qirui Yang

FAST '17: 15th USENIX Conference on File and Storage Technologies
February 27–March 2, 2017
Santa Clara, CA

Message from the Program Co-Chairs vii

Tuesday, February 28

Garbage

Algorithms and Data Structures for Efficient Free Space Reclamation in WAFL 1
Ram Kesavan, Rohit Singh, and Travis Grusecki, *NetApp;* Yuvraj Patel, *University of Wisconsin—Madison*

Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs 15
Shiqin Yan, Huaicheng Li, Mingzhe Hao, and Michael Hao Tong, *University of Chicago;* Swaminathan Sundararaman, *Parallel Machines;* Andrew A. Chien and Haryadi S. Gunawi, *University of Chicago*

The Logic of Physical Garbage Collection in Deduplicating Storage 29
Fred Douglis, Abhinav Duggal, Philip Shilane, and Tony Wong, *Dell EMC;* Shiqin Yan, *Dell EMC and University of Chicago;* Fabiano Botelho, *Rubrik, Inc*.

The System

File Systems Fated for Senescence? Nonsense, Says Science! 45
Alex Conway and Ainesh Bakshi, *Rutgers University;* Yizheng Jiao and Yang Zhan, *The University of North Carolina at Chapel Hill;* Michael A. Bender, William Jannen, and Rob Johnson, *Stony Brook University;* Bradley C. Kuszmaul, *Oracle Corporation and Massachusetts Institute of Technology;* Donald E. Porter, *The University of North Carolina at Chapel Hill;* Jun Yuan, *Farmingdale State College of SUNY;* Martin Farach-Colton, *Rutgers University*

To FUSE or Not to FUSE: Performance of User-Space File Systems 59
Bharath Kumar Reddy Vangoor, *Stony Brook University;* Vasily Tarasov, *IBM Research-Almaden;* Erez Zadok, *Stony Brook University*

Knockoff: Cheap Versions in the Cloud 73
Xianzheng Dou, Peter M. Chen, and Jason Flinn, *University of Michigan*

HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases 89
Salman Niazi, Mahmoud Ismail, Seif Haridi, and Jim Dowling, *KTH Royal Institute of Technology;* Steffen Grohsschmiedt, *Spotify AB;* Mikael Ronström, *Oracle*

Edward Sharpe and the Magnetic Zeros

Evolving Ext4 for Shingled Disks 105
Abutalib Aghayev, *Carnegie Mellon University;* Theodore Ts'o, *Google, Inc.;* Garth Gibson, *Carnegie Mellon University;* Peter Desnoyers, *Northeastern University*

SMaRT: An Approach to Shingled Magnetic Recording Translation 121
Weiping He and David H.C. Du, *University of Minnesota*

Facilitating Magnetic Recording Technology Scaling for Data Center Hard Disk Drives through Filesystem-Level Transparent Local Erasure Coding 135
Yin Li and Hao Wang, *Rensselaer Polytechnic Institute;* Xuebin Zhang, *Dell EMC/DSSD;* Ning Zheng, *Rensselaer Polytechnic Institute;* Shafa Dahandeh, *Western Digital;* Tong Zhang, *Rensselaer Polytechnic Institute*

(Continues on next page)

Wednesday, March 1

Corruption

Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions 149
Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, *University of Wisconsin—Madison*

Omid, Reloaded: Scalable and Highly-Available Transaction Processing 167
Ohad Shacham, *Yahoo Research;* Francisco Perez-Sorrosal, *Yahoo;* Edward Bortnikov and Eshcar Hillel, *Yahoo Research;* Idit Keidar, *Technion—Israel Institute of Technology and Yahoo Research*; Ivan Kelly, *Midokura;* Matthieu Morel, *Skyscanner;* Sameer Paranjpye, *Arimo*

Application Crash Consistency and Performance with CCFS 181
Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, and Lanyue Lu, *University of Wisconsin—Madison;* Vijay Chidambaram, *The University of Texas at Austin;* Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, *University of Wisconsin—Madison*

High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System 197
Harendra Kumar; Yuvraj Patel, *University of Wisconsin—Madison;* Ram Kesavan and Sumith Makam, *NetApp*

Frameworks

Mirador: An Active Control Plane for Datacenter Storage 213
Jake Wires and Andrew Warfield, *Coho Data*

Chronix: Long Term Storage and Retrieval Technology for Anomaly Detection in Operational Data 229
Florian Lautenschlager, *QAware GmbH;* Michael Philippsen and Andreas Kumlehn, *Friedrich-Alexander-Universität Erlangen-Nürnberg;* Josef Adersberger, *QAware GmbH*

Crystal: Software-Defined Storage for Multi-Tenant Object Stores 243
Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, and Pedro García-López, *Universitat Rovira i Virgili;* Yosef Moatti and Eran Rom, *IBM Research—Haifa*

Solid State Records

WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems 257
Se Kwon Lee, *UNIST (Ulsan National Institute of Science and Technology);* K. Hyun Lim, *Hongik University;* Hyunsub Song, Beomseok Nam, and Sam H. Noh, *UNIST (Ulsan National Institute of Science and Technology)*

SHRD: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizing in Device 271
Hyukjoong Kim and Dongkun Shin, *Sungkyunkwan University;* Yun Ho Jeong and Kyung Ho Kim, *Samsung Electronics*

Graphene: Fine-Grained IO Management for Graph Computing 285
Hang Liu and H. Howie Huang, *The George Washington University*

Thursday, March 2

Faster Faster

vNFS: Maximizing NFS Performance with Compounds and Vectorized I/O 301
Ming Chen, *Stony Brook University;* Dean Hildebrand, *IBM Research-Almaden;* Henry Nelson, *Ward Melville High School;* Jasmit Saluja, Ashok Sankar Harihara Subramony, and Erez Zadok, *Stony Brook University*

On the Accuracy and Scalability of Intensive I/O Workload Replay 315
Alireza Haghdoost and Weiping He, *University of Minnesota;* Jerry Fredin, *NetApp;* David H.C. Du, *University of Minnesota*

On the Performance Variation in Modern Storage Stacks 329
Zhen Cao, *Stony Brook University;* Vasily Tarasov, *IBM Research-Almaden;* Hari Prasath Raman, *Stony Brook University;* Dean Hildebrand, *IBM Research-Almaden;* Erez Zadok, *Stony Brook University*

Enlightening the I/O Path: A Holistic Approach for Application Performance. 345
Sangwook Kim, *Apposha and Sungkyunkwan University;* Hwanju Kim, *Sungkyunkwan University and Dell EMC;* Joonwon Lee and Jinkyu Jeong, *Sungkyunkwan University*

Open Channel D

LightNVM: The Linux Open-Channel SSD Subsystem 359
Matias Bjørling, *CNEX Labs, Inc. and IT University of Copenhagen;* Javier Gonzalez, *CNEX Labs, Inc.;* Philippe Bonnet, *IT University of Copenhagen*

FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs 375
Jian Huang, *Georgia Institute of Technology;* Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, and Bikash Sharma, *Microsoft;* Moinuddin K. Qureshi, *Georgia Institute of Technology*

DIDACache: A Deep Integration of Device and Application for Flash Based Key-Value Caching 391
Zhaoyan Shen, *Hong Kong Polytechnic University;* Feng Chen and Yichen Jia, *Louisiana State University;* Zili Shao, *Hong Kong Polytechnic University*

Message from the FAST '17 Program Co-Chairs

Welcome to the 15th 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, SMR, reliability, transaction processing, cloud storage, measurement, and traditional file systems. Our authors hail from eight countries on three continents and represent academia, industry, and the open-source communities. Many of the submitted papers are the fruits of collaboration among all these communities.

FAST '17 received 116 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, 68 papers received at least two more reviews. The Program Committee discussed 56 papers in an all-day meeting on December 9, 2016, at Harvey Mudd College in Claremont, California. 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 five years, we included a category of short papers. We received 27 short paper submissions, but to our surprise, none were accepted. This year we included a new category of deployed-systems papers, which address experience with the practical design, implementation, analysis or deployment of large-scale, operational systems. Deployed systems papers need not present new ideas or results, but should offer useful guidance to practitioners. We received 10 deployed-systems submissions, of which we accepted 3. We were happy to see a number of submissions (and accepted papers) from adjacent areas such as anomaly detection and graph processing.

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 '17. We would also like to thank the attendees of FAST '17 and the future readers of these papers. Together with the authors, you form the FAST community and make storage research vibrant and exciting. We extend our thanks to the USENIX staff, especially Casey Henderson, Hilary Hartman, and Michele Nelson, 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 Keith Smith and 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 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 479 thoughtful and meticulous reviews. HotCRP recorded over 163,000 words in reviews and comments (excluding HotCRP boilerplate; 329K when included). The reviewers' evaluations, 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 Emily Dorsey, Emily First, and Elizabeth Krenkel; local arrangements and administrative support from Joyce Greene; and IT and A/V support from Taylor Calderone.

We look forward to an interesting and enjoyable conference!

Geoff Kuenning, *Harvey Mudd College*
Carl Waldspurger, *CloudPhysics*
FAST '17 Program Co-Chairs

Algorithms and Data Structures for Efficient Free Space Reclamation in WAFL

Ram Kesavan, Rohit Singh, Travis Grusecki
Netapp
ram.kesavan@gmail.com {rh0,travisg}@netapp.com

Yuvraj Patel
University of Wisconsin-Madison
yuvraj@cs.wisc.edu

Abstract

NetApp®WAFL®is a transactional file system that uses the copy-on-write mechanism to support fast write performance and efficient snapshot creation. However, copy-on-write increases the demand on the file system to find free blocks quickly; failure to do so may impede allocations for incoming writes. Efficiency is also important, because the task may consume CPU and other resources. In this paper, we describe the evolution (over more than a decade) of WAFL's algorithms and data structures for reclaiming space with minimal impact on the overall storage appliance performance.

1 Introduction

A file system controls the storage and retrieval of data and metadata. It typically carves up its persistent storage into addressable blocks and then allocates and frees these blocks to process client requests. Efficient free space management is a crucial element of file system performance. Enterprise-class file systems demand consistently high performance for reads and writes. NetApp, Inc. is a storage and data management company that offers software, systems, and services to manage and store data, including its proprietary Data ONTAP® operating system [12]. Data ONTAP implements a proprietary file system called Write Anywhere File Layout (WAFL) [11]. WAFL is a transaction-based file system that employs copy-on-write (COW) mechanisms to achieve fast write performance and efficient snapshot creation.

Like other modern file systems such as FFS [16], XFS [22], ZFS [17], ext3, ext4 [15], and brtfs [19], WAFL tracks allocated and free space for two basic reasons: to enable the file system to find and allocate free blocks to accommodate new writes; and to report space usage to the system administrator to guide purchasing and provisioning decisions. Unlike ext3 and ext4, which write data in place, WAFL, ZFS, and btrfs never overwrite a block containing active data or metadata in place; instead, a free block is allocated and the modified data or metadata is written to it. Because the old data is preserved during a transaction, the process of creating a snapshot is simplified. However, each overwrite also obsoletes the older block (if that block does not belong to any snapshot). Such file systems have to track and reclaim these unneeded blocks at pretty much the same rate as the rate of incoming writes to ensure sufficient availability of free space for new allocations.

The metadata that tracks free space can either be persistent or maintained in memory. Storing this metadata only in memory has the downside that it needs to be rebuilt whenever the machine reboots or the file system is brought online. Therefore, most file systems choose to persist this metadata. Traditional file systems keep their free space metadata up to date. Alternatively, a file system can choose to let its free space metadata go stale, and periodically scan the entire file system to recompute it. Although there is an up-front performance cost to keeping the metadata up to date, this choice provides a convenient pay-as-you-go model for free space reclamation work. An asynchronous recomputation of free space can avoid the up-front performance cost, but file system designs need to cope with potentially stale free space metadata, including how that stale information is reported to the administrator. Additionally, scanning the entire file system to recompute free space only gets more expensive as file system size and activity increases. WAFL has always chosen to keep its free space metadata up to date, except when it comes to snapshots. This requires a free space reclamation infrastructure that can keep up with performance requirements without taking significant CPU and storage I/O away from servicing client operations.

In this paper, we present that infrastructure and show how it satisfies high performance requirements. We describe how it has evolved to keep up with the increase in the file system size while accommodating storage effi-

ciency features like snapshots, clones, deduplication, and compression. We also explain the exception case; that is, how free space is reclaimed asynchronously and efficiently after snapshots are deleted. We do not discuss how free space is reported to the administrator.

We first present the challenges of free space reclamation for a modern enterprise file system with diverse requirements. We then explore the problems and their corresponding solutions as well as their evolution. We analyze these solutions by using performance experiments. Rather than writing a consolidated evaluation section, we chose to include the relevant evaluation in each section; we believe that this approach improves the readability of the paper. Along the way, we present some lessons learned from building and maintaining a feature-rich file system that runs on hundreds of thousands of systems that service a wide range of applications.

2 Background & Motivation

This section presents the challenges of tracking and reclaiming free space to enterprise-quality file systems, which are multi-TiB in size and can process gibibytes per second (GiB/s) worth of operations.

2.1 How Do Blocks Get Freed?

Copy-on-write (COW) file systems never modify data or metadata in-place. Instead, they write out the the new version to a free location, leaving the previous version intact. When combined with a transactional model for writing out a collection of modifications, the file system on the persistent media remains consistent even after a power loss. A COW model means that an overwrite almost always makes the previous version of the block unnecessary unless it belongs to a snapshot. Thus, a 1 GiB/s overwrite workload generates 1 GiB/s worth of potentially free blocks. This means that a COW file system has to (1) find and allocate 1 GiB/s worth of free space for new writes, and (2) modify metadata to record 1 GiB/s worth of freed blocks to keep the file system up to date. The updates to the metadata also generate new allocations and potential frees.

When a file is deleted the file system needs to eventually reclaim all of its blocks. Certain applications create and delete files in bursts. Reclaiming that space in a predictable fashion is a significant challenge.

The ability to create and retain snapshots of a file system is crucial to the life cycle of data management; snapshots are used for data protection, replication, recovery, and so on. When snapshots are deleted, all blocks that uniquely belong to them need to be reclaimed by the file system. Storage efficiency features like compression and deduplication often run in the background to reduce space consumption and free blocks.

A file system must choose a rate for reclaiming free blocks generated from the above activities that satisfies the customer's need for accurate reporting (for provisioning and purchasing decisions) while minimizing performance impact to client operations.

2.2 Lazy Reclamation of Free Space

Traditionally, file systems keep their free space metadata up to date. For instance, if objects in the file system reference block *b*, and if all of these references are dropped in a transaction to the file system, then the free space metadata that is written out by that transaction would indicate that *b* is free. In other words, the metadata persisted by a transaction accurately tracks all the deleted references to blocks. Thus, the update cost increases directly with the number of deletes. This provides a pay-as-you-go model for free space reclamation work.

Alternatively, file systems can choose a consistency model in which the free space metadata can become stale but in a conservative fashion. In this model, dropping a reference becomes very simple; at the end of the transaction from the previous example, the file system metadata indicates that *b* is still allocated even though no objects refer to it. In the background, a periodic scan walks every object in the file system to rebuild a map of blocks referenced by all its objects. This idea has been explored in a research setting [9] and commercially [24]. Using standard hard disks with a typical user workload, scanning these metadata blocks can take 2 to 4 seconds per GiB of user data [9]. Under these assumptions, it would take more than a day to scan a 50 TiB collection of user data before any free space could be reclaimed. Modern storage efficiency techniques like compression, deduplication, cloning, etc., pack more user data into usable storage, and allow user data blocks to be pointed to by different metadata blocks or files. This results in more metadata for the same amount of used storage. Thus, even when hard disks are replaced by faster media like SSDs, the reclamation scan of a multi-TiB file system would still take hours to complete. While this may be acceptable with relatively static datasets, it quickly becomes untenable under heavy write workloads and with increasing file system sizes. The file system would need to hold aside a large reserve of free space to hedge against unpredictable reclamation times. For example, a workload with an overwrite rate of 200 MiB/s would produce more than 17.2 TiB of writes over the course of a day. A file system with 50 TiB of user data would need to keep around 34% free space just to ensure that overwrites suc-

ceed, even if the background scanner is running as fast as possible.

Therefore, WAFL chooses to pay the continual cost of keeping its metadata up to date in all cases, except for snapshots. Section 5 describes how space is reclaimed after snapshot deletion.

2.3 Cost Of Freeing Blocks In WAFL

This section is a brief introduction to WAFL. The Write Anywhere File Layout (WAFL) is a UNIX style file system with a collection of inodes that represent its files [11, 8]. The file system is written out as a tree of blocks rooted at a superblock. Every file system object in WAFL, including metadata, is a file. A file in WAFL is a symmetric n-ary tree of 4 KiB blocks, where the level of the tree is $\lceil log_n \frac{filesize}{4KiB} \rceil$. The leaf node ($L_0$) holds file data, the next level node (L_1) is an indirect block that refers to L_0s, and so on. WAFL is a COW file system, where every modified block of a file is written to a new location on storage. Only the file system superblock is ever written in place. One advantage of this method is that new allocations can be collected together and written out efficiently.

As buffers and inodes are modified (or *dirtied*) by client operations, they are batched together for performance and crash consistency. Every mutable client operation is also recorded to a log in nonvolatile memory before it is acknowledged; the operations in the log are replayed to recover data in the event of a crash. WAFL collects the resultant dirty buffers and inodes from hundreds of thousands of logged operations, and uses a checkpoint mechanism called a *consistency point* (*CP*) to flush them to persistent media as one single and very large transaction. Each CP is an atomic transaction that succeeds only if all of its state is successfully written to persistent storage. Updates to in-memory data structures are isolated and targeted for a specific CP to ensure that each CP represents a consistent and complete state of the file system. Blocks that are allocated and freed for a CP are also captured as modifications to the allocation metadata files that are written out in that same CP. Once the entire set of new blocks that belong to a CP is persisted to storage, a new file system superblock is atomically written in-place that references this new file system tree [8, 11].

File systems use different data structures like linked-lists, bitmaps, B(+) trees, etc., to track free space information, and incorporate the cost of maintaining such structures in their designs. Their block allocators either use these structures directly or build secondary structures to help find space efficiently. WAFL uses bitmap files as the primary data structure to track the allocated or free state of a block. For example, the i^{th} block of the file system is free if the i^{th} bit in the *activemap* file is 0; the block is allocated if the bit is 1. WAFL uses a 4 KiB block size, and therefore, a 1 GiB WAFL file system needs a 32 KiB activemap file, a 1 TiB WAFL file system needs a 32 MiB one, and so on. Clearly, the metadata for multi-TiB sized file systems is too large to fit in memory - the WAFL buffer cache [7] - and needs to be read on-demand from persistent media. WAFL uses auxiliary structures based on the bitmaps to speed up block allocation, but that is outside the scope of this paper.

WAFL allocates blocks that are colocated in the block number space, which minimize updates to the activemap. However, frees may be distributed randomly over the number space, and all such updates to the activemap have to be written out in the same transaction. The more random the updates, the larger the number of dirty activemap buffers for that CP to process. This prolongs the transaction, which negatively affects the write throughput of the file system.

In the rest of this paper, we discuss the techniques that let WAFL handle the nondeterministic nature of block free processing in a way that ensures smooth and deterministic system performance.

2.4 Free Space Defragmentation

Reclaiming free space is not necessarily the same as generating contiguous free space. As a file system ages, the free space in that file system gets fragmented as random blocks are freed while others stay allocated or trapped in snapshots. Contiguous free space is important for write performance and subsequent sequential read performance.

Various ways exist to defragment free space. File systems like LFS [20] use techniques like *segment cleaning* to reclaim contiguous free space. Some file systems put aside a large reserve of free space or recommend holding the space usage below a certain percentage in order to provide acceptable levels of free space contiguity and performance. Some file systems provide tools to defragment free space. WAFL implements both background and inline free space defragmentation. However, techniques for free space defragmentation are outside the scope of this paper.

3 Free Space: The Basics

Let us first define the consistency model of the activemap. As explained in earlier sections, the superblock points to a self-consistent tree of blocks that define a WAFL file system. This means that the metadata in the tree accurately describes the allocated blocks of the tree.

Definition 1. Every tree of blocks contains an activemap of bits; its i^{th} bit (referred to as a_i) is set iff the i^{th} block b_i of the file system is in use by that tree.

3.1 Reuse of Freed Blocks

Lemma 1. *In a COW file system, a block freed during transaction CP_n can only be allocated for use in a subsequent transaction. In other words, it cannot be allocated to a new block written out by transaction CP_n.*

Proof. Transaction CP_n gets persisted only when its new superblock is written in-place; a disruption discards all intermediate state. Recovery is accomplished by starting with the last persistent state, CP_{n-1}, and reapplying changes computed by replaying the operations in the nonvolatile log. Therefore, CP_{n-1} must remain intact until the CP_n is complete. Any free and reuse of a block by CP_n violates this invariant. ■

Thus, when bit a_i is cleared in CP_n, the WAFL block allocator cannot use b_i until the superblock of CP_n has been written out. This is implemented quite simply by creating a second copy of the corresponding activemap 4 KiB buffer when the first bit in it is cleared. The two copies are called *safe* and *current*; the block allocator consults the former and bits are cleared only in the latter. The block allocator sets bits in both copies when recording allocations. The current copy is eventually written out by the CP and the safe copy is discarded when the CP completes. Therefore, any activemap buffer dirtied due to a block free needs twice the memory, and this gets expensive if a large number of random blocks get freed in one CP. Although this state could be maintained in a more memory-efficient manner, it would result in longer codepaths for consulting the activemap. Section 4 describes a more elegant solution to the random update problem.

3.2 A Cyclic Dependency

The activemap file is a flat self-referential file. Like all files, it is composed of blocks and those blocks are by definition also tracked by the activemap. An activemap block written to b_j *covers* activemap block b_i if bit a_i resides in b_j.

Definition 2. An *activemap chain* that starts with a specific activemap block is defined as the longest list of activemap blocks where the $(i+1)^{th}$ block in the list covers the i^{th} block.

By definition, each activemap block can belong to only one unique activemap chain. Thus, assuming that no snapshots are present, when an activemap buffer is dirtied and the subsequent CP (allocates a new block and) frees the old block corresponding to that activemap block, it must dirty the next element in the chain. If that buffer is not yet dirty, this step is repeated. This continues until a previously dirty activemap buffer is encountered. Thus, when any block is freed and its activemap bit is cleared, the entire chain for that activemap block gets updated. In theory, all blocks of the activemap could belong to a single chain; in the worst case, a CP might dirty and write out the entire activemap file! As mentioned earlier, the activemap of a multi-TiB file system might not fit into main memory.

It should be noted that long chains are not easily formed because WAFL often allocates blocks that are colocated in the activemap; they are only formed by very unlikely sequences of allocations and frees. Three solutions have been built in WAFL to solve this problem.

1. **Prefetch the entire chain**. WAFL prefetches the entire chain when an activemap buffer is dirtied. Ideally, the prefetches complete by the time the CP starts allocating new blocks for the activemap file of the file system; the CP processes metadata towards its end, so this works in the right circumstances.

2. **Preemptively break chains**. A background task preemptively finds each moderately long chain and dirties its first block; the subsequent CP breaks that chain.

3. **Postpone the free of the old block**. When a CP processes a dirty activemap buffer, it simply moves the reference to the old block to a new metafile called the *overflow file*. The L_1s of the overflow file serve as an appendlog for free activemap blocks, and the CP avoids dirtying the next element of the chain. Once the CP completes, the blocks in the overflow file are now really freed; their corresponding activemap bits are cleared. Thus, an activemap chain of n elements gets broken down completely in at most n consecutive CPs. This ensures consistent CP length without violating Definition 1.

Section 3.4 discusses the evolution of these solutions in WAFL.

3.3 Processing File Deletion

To process a file deletion operation, WAFL moves the file from the directory namespace into a hidden namespace. However, the space used by the hidden file is not reclaimed; this does not violate Definition 1. Then, potentially across several CPs, the file gradually shrinks as WAFL frees the blocks of this file incrementally. Eventually, the entire tree of blocks is freed and the inode for the hidden file reclaimed. The updates to the activemap are more random if the file has been randomly overwritten, because its tree will refer to random blocks; we call these *aged files*.

3.4 Evaluation and Some History

It should be clear by now that if a very large number of random blocks are freed in a single CP it will need to process a lot of dirty activemap buffers. Long activemap chains can make a bad situation worse.

WAFL was originally designed in 1994 for file sharing and user directory workloads over NFS [21] and CIFS [10], which is characterized by the sequential read and write of many small files. The maximum file system size was 28 GiB at the time; the activemaps for these file systems spanned approximately 200 blocks. The problems described in this section were not applicable for these particular workloads with such small quantities of metadata.

More than two decades later, WAFL hosts a very wide range of applications, from file-intensive applications to virtual machine images and databases over both file-based protocols like NFS/CIFS and block-based SCSI protocols [18], which do random I/O to fewer large files. During this timespan, the sizes of individual disks (hard drive and solid state) and file systems have exploded. The maximum file system size supported by WAFL has increased from 28 GiB in 1994 to 16 TiB in 2004, and to 400 TiB in 2014. Today, the activemaps for the largest WAFL file systems span millions of blocks.

As a result, the problem of random updates to metadata became acute. Larger and larger fractions of the available CPU cycles and I/O resources were being consumed for processing frees. Some applications, like the ones used for electronic design automation, needed to delete large numbers of files in bursts without compromising the performance of other workloads on the system. Clearly, this was not ideal. Section 4 describes the solutions to this problem.

In 2000, WAFL used two mechanisms in tandem – prefetching the activemap chain and preemptively breaking activemap chains – to avoid the cyclic dependency problem. By 2010, WAFL supported file systems that were 100 TiB in size. So, these mechanisms were replaced by the overflow file, which works well for file systems of any size. The overflow file mechanism provided an interesting technique for postponing deletion without changing the consistency semantics of the activemap. Section 4 presents designs that use this technique to convert many random updates to the activemap into fewer and predictable bunched updates.

4 Batching Updates to Metadata

As explained earlier, when a large number of random blocks are freed in a CP, they can generate many metadata updates. If these updates can be accumulated, they can be sorted into more efficient metadata updates. And, if this work can be done over the course of multiple CPs, the cost can be spread out over time in a predictable fashion.

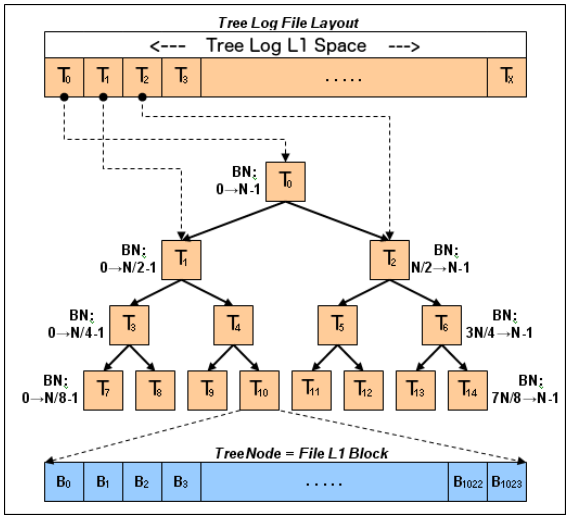


Figure 1: Structure of the TLog.

Much like the overflow file, the two structures described in this section postpone the freeing of blocks by taking ownership of them; the blocks are referenced by the L_1s of these files, which preserves Definition 1. We call this *delete logging*. File system consistency checking treats these structures like regular files. Sorting occurs by (partially) ordering the references by block number across the L_1s of these files. Once they have been sufficiently sorted, the references to the blocks are punched out from the L_1s, and the corresponding activemap bits cleared in a batched fashion.

4.1 The Tree Log

The first manifestation of a delete logging mechanism in WAFL was the *Tree Log* (or *TLog*). As Figure 1 shows, the L_1s (which reference the blocks that the TLog owns) of the TLog form the nodes of a binary tree, where each node contains block references lying in a fixed range of the block number space. A left child covers the left half of its parent's range, and a right child covers the right half.

A delete-logged block is inserted into the root node. If the root node fills up, its contents are pushed to the children nodes based on the range they cover. If either child fills up, the process cascades down the tree. If a leaf node of the tree fills up, all the blocks in it are punched out and the activemap is updated. The size of the TLog determines the range covered by a leaf node. For ex-

ample, if a file system has n blocks and each activemap block covers 32K blocks, a TLog sized to have $n/32K$ leaf nodes ensures that each leaf node covers a single activemap block. In typical deployments, a TLog sized to approximately 1.5% of the file system size provides significant performance improvement; each leaf node covers 4 to 6 activemap blocks.

The TLog structure had one significant problem: there was no easy way to tell how many nodes would be updated due to an insertion. That was a function of which part of the binary tree was affected by an insertion and the fullness of the nodes in that part of the tree. In extreme cases, a small number of insertions would suddenly cause a large number of nodes to be updated, which would result in the CP needing to process many more dirty TLog buffers than if the frees had not been delete-logged. To alleviate these worst-case events, the infrastructure would prefetch the activemap buffers (as it would if delete logging were off), and when it detected one of these bad situations it would stop delete-logging and free directly instead. In practice, this situation was very rare and not reliably created in our testing. However, after it was seen in the field we decided to gradually move away from the TLog structure to a new *Batched Free Log* (or *BFLog*) solution.

4.2 The Batched Free Log

Instead of embedding a sorting data structure in its L_1s, the BFLog adopts a different approach. It accumulates delete-logged blocks and, after some threshold, it sorts and frees them.

This can be achieved in several ways. In WAFL, it is accomplished by using three files: (a) the *active log*, (b) the *inactive log*, and (c) the *sorted log*. A block is delete-logged by appending to an L_1 of the active log. Once the active log fills to a certain threshold, it becomes the inactive log and it is replaced by an empty active log. The delete-logged blocks in the inactive log are sorted across all of its L_1s. This is accomplished in two steps: sorting the blocks within a fixed number of L_1s at a time, followed by a merge-sort, which also moves the blocks to the sorted log. Once the sorted log is ready, its blocks are punched out and the activemap is updated in a bunched fashion. It should be noted that all processing of BFLog data structures is localized or sequential in nature, which is important for its performance.

Once the BFLog reaches steady-state, it becomes possible to pace its operations such that, on the average, each append of a new block results in one block being sorted and/or one block being merge-sorted and/or one block being punched out of the sorted log. This is in stark contrast to the unpredictability of the TLog, where the sorting was strictly controlled by the contents of the binary tree. Rigorous experimentation showed that sizing the BFLog (all three files together) to 0.5% of the file system provided sufficient performance boost.

4.3 Logging in a Reference Counted File System

Several storage efficiency features of WAFL depend on the property that a block can be shared by multiple files or within the same file. The extra references to a block are tracked by a *refcount* file, which is a flat file of integers, one per block. When a multiply referenced block is freed, the refcount file is updated, and because the file is several times less dense than the activemap, the problem caused by random frees of blocks increases manifold. The now ubiquitous deduplication feature results in highly reference counted file systems, which makes delete-logging even more critical. The delete-logging of a multiply referenced block is no different from that of a regular block; when punching out the block the refcount file is updated for all but the last reference.

4.4 Evaluation

Although we have anecdotal data from a few customer systems that show the unpredictable behavior of the TLog, we do not have reproducible experiments to demonstrate it. Because the TLog has now been replaced by the BFLog, we present BFLog data only to show the merits of delete-logging.

We first studied the benefits to random overwrite workloads. A set of LUNs [23] was configured and several clients were used to generate heavy random 8 KiB overwrite traffic to the LUNs; this simulates database/OLTP writes. Although the benefit of the BFLog was a bit muted on high-end platforms (we observed a roughly 5% to 10% improvement in throughput), they were higher on mid-range platforms. Figure 2 shows results on a mid-range system with 12 cores (Intel Westmere) and 98 GiB of DRAM.

Without delete logging, our throughput plateaued at approximately 60k IOPs, whereas with delete logging we were able to continue to about 70k IOPs; this represents approximately a 17% improvement in throughput. In addition to a throughput improvement, we also observed anywhere from a 34% to 48% improvement in latency across most load points. These improvements were achieved via a 65% reduction in our metadata overhead, because the BFLog was able to batch and coalesce a large number of updates to metadata blocks.

The SPEC SFS [4] is not capable of generating the sort of file deletion load that some of our customers do. We

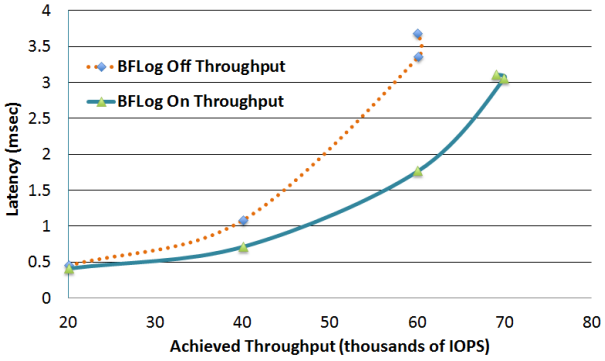


Figure 2: The benefits of delete logging for a random overwrite workload on a mid-range system.

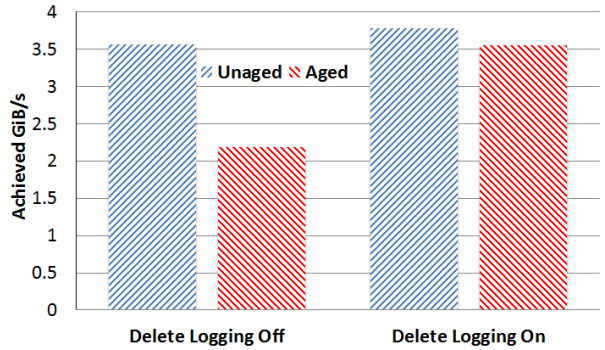


Figure 3: The benefits of delete logging on a high-end all-SSD system for the file deletion throughput workload.

fashioned a custom benchmark that measures file deletion throughput as the rate at which an otherwise idle system reclaims space after a number of large files totalling 1 TiB get deleted in short order.

Figure 3 shows the results of the delete throughput test with and without BFLog on an all-SSD high-end system with 20 cores (Intel Ivy Bridge) and 128 GiB of DRAM. This was repeated with files that had been significantly aged by randomly overwriting them for two hours. The blocks freed from an aged file are more randomly distributed over the number space. The baseline runs (without the BFLog) show that 2 hours of aging results in a 40% drop in delete throughput.

Although the BFLog improved delete throughput by a modest 6% in the unaged data set, it improved delete throughput by over 60% in the aged one. It shrunk the difference between the aged and unaged results from 40% to 6%, which shows that it offset the randomness in the blocks caused by aging. It should be noted that using solid state undersells the improvement. The benefits are much higher on a system with hard drives, because random IOs to load the activemap blocks result in a much bigger difference without delete-logging. Additionally, if this were a highly deduplicated dataset (which it wasn't) the improvement would have been much larger because the BFLog would batch the updates to the less dense refcount file. This result shows the primary benefit of delete logging, i.e., reordering random metadata found in aged file systems so that it appears to be sequential, as found in unaged file systems.

Without delete-logging, increasing delete throughput requires forcing deletes to run faster, which in turn results in more random reads of activemap blocks and more work for the CP to write them out. This takes valuable I/O and CPU resources away from client workloads. Delete-logging allows us to increase delete throughtput without hurting client workloads.

5 Snapshots and Free Space

Snapshots are a crucial building block for many data management features. A file system snapshot is a point-in-time version of the entire file system. In WAFL, it is created quite simply by storing an extra copy of a CP's superblock. None of the blocks in the snapshot can be freed until the snapshot is deleted. This section explains how free space reclamation protects and reclaims the blocks held in snapshots.

We introduce a new term, *volume*, which includes the current file system and all of its snapshots. The current file system is the active part of the volume that services all client operations whose modifications are being written out by the CP mechanism. Snapshots are denoted by S_i, and we use S_0 to denote the current file system.

Clearly, the activemap in S_0 cannot really tell us which blocks are trapped in a given snapshot. Initially the activemap of S_0 is literally the same tree of blocks as the activemap of a new snapshot, S_1, but it quickly diverges due to changes in S_0. The blocks of S_1 are already recorded in its activemap (called *snapmap*), which remains unchanged due to the COW nature of the file system. Therefore, a block used by S_1 can get freed in S_0, and then be marked as free in the activemap.

Lemma 2. *The set of blocks pointed to by a volume can be represented by the bit-OR of its activemap together with the snapmaps of all its snapshots.*

Proof. Since a snapshot is simply a tree of blocks pointed to by a CP's superblock, based on Definition 1, each snapmap accurately captures the set of blocks pointed to by that tree. Therefore, the bit-OR of the activemap with all the snapmaps accurately represents the set of blocks pointed to by the volume. ■

5.1 The Summary Map

A block in the volume is free iff it is free in the activemap and in all the snapmaps. The latter condition needs more IO and CPU to ascertain. There are several options for solving this problem: a file that stores a bitmask per block with one bit per snapshot, a B+tree that captures free/allocated blocks in each snapshot in some searchable but condensed fashion, etc. WAFL uses a bitmap file called the *summary map*, which is a bit-OR of the snapmaps of all snapshots in the volume. This simple choice allows for inexpensive space tracking.

Space reclamation after a snapshot deletion happens lazily but, as this section shows, efficiently. When a snapshot is deleted, some bits in the summary need to be cleared, and doing that atomically (as part of a single CP) is impractical. There are other reasons for the summary to become instantaneously stale, but in a conservative way. WAFL supports a feature called *volume snaprestore*, in which a volume can jump backwards in time to an older snapshot, say S_i. This is accomplished rather trivially by taking the superblock of S_i and writing it out as the volume's superblock in the next CP. The summary map at the time of S_i's creation had been frozen, and after the snaprestore, it becomes the summary of S_0. However, this summary can include blocks that belong to snapshots older than S_i that have since been deleted.

5.2 Snapshot Creation

When a snapshot is created, the summary map has not yet been updated with the new snapshot's blocks. A background *snap create scan* now walks and bit-ORs the snapmap into the summary. It should be noted that this scan is idempotent, and so it can safely restart from the beginning after a reboot.

Theorem 1. *Assuming no deleted snapshots, the set of blocks pointed to by a volume is always guaranteed to be equal to the bit-OR of the activemap and summary even while the snap create scan is in progress.*

Proof. Although the summary does not necessarily include the blocks of a newly created snapshot, S_1, this does not violate Lemma 2, because all blocks that belong to S_1 are also recorded in the activemap at the instant of S_1's creation. To maintain the invariant, if a block is to be freed in the activemap and if it is also shared by S_1, it must be recorded in the summary. In other words, if the snap create scan hasn't gotten to that summary bit yet, it is done on demand. This stops once the snap create scan has completed its task. ■

There are few more interesting cases here. The newly created snapshot S_1 could get deleted before the scan is done, but let us delay that discussion till Section 5.4. A new snapshot S_2 could get created before S_1's scan is complete. In that case, the scan restarts but switches to bit-OR'ing S_2's snapmap into the summary. There is no longer a need to process S_1's snapmap because all blocks that belong to S_1 either belong to S_2 or were freed before S_2 was created. Section 5.5 shows how this consultation as well as the entire snap create scan can be eliminated!

5.3 Another Cyclic Dependency

Another cyclic dependency comes into play while the snap create scan is in progress. Suppose that snapshot S_1 gets created and so the activemap is identical to S_1's snapmap. Now, b_i gets freed and so a_i is cleared and the i^{th} bit in the summary is set by the snap create on-demand work. This results in a dirty summary buffer, which means that the old version of that summary block b_j has to be freed. So a_j gets cleared, which in turn sets the j^{th} summary bit, and so on. Thus, although unlikely, long activemap-summary chains can get created, which is twice as bad as the activemap chain problem. One customer system hit this problem on an aged volume of size 80 TiB. The CP uses the overflow file to postpone the free of the old summary block and thereby avoids dirtying the next activemap in the chain.

5.4 Snapshot Deletion

When a snapshot is removed from the namespace its blocks may or may not be reflected in the summary map. A background *snap delete scan* walks the summary to remove blocks that belong exclusively to that snapshot. This scan is also idempotent and can be safely restarted at any time.

Theorem 2. *Independent of the status of the snap create and delete scans, the set of blocks pointed to by a volume is always equal to or a subset of the bit-OR of the activemap and summary map.*

Proof. Theorem 1 proved that the bit-OR is equal to the set of blocks pointed to by a volume independent of the status of the create scan. While the snap delete scan is in progress, the summary is guaranteed to include all the bits that are set in the deleting snapshot's snapmap unless the snapshot got deleted before its corresponding snap create scan was able to completely bit-OR its snapmap into the summary. Then, either the block still lives in S_0, in which case its activemap bit would be set, or it has since been freed, in which case, there's no need for the summary to protect it if it was uniquely owned by the deleted snapshot. ■

Lemma 3. *Let all snapshots in a volume (including S_0) be sorted in temporal order of creation. For any block b_i in the volume, the bit string created by concatenating the i^{th} bit of all the snapmaps – in the case of S_0, the activemap – in that temporal order will yield a $0^*1^*0^*$ pattern. This is called the 010 snapshot property.*

Proof. Once allocated in S_0, a block can belong to subsequent snapshots until it gets freed in S_0. Once freed in S_0, it is unavailable for allocation until the activemap and all snapmaps say it is free. So, it cannot belong to subsequent snapshots until all snapshots that it belongs to have been deleted. Hence, the bit pattern. ■

There are two ways for the delete scan to update the summary, one of which uses the 010 property.

1. **Deletion by Subtraction (or dbys)**. This scan uses the 010 property to process the deletion of S_i. If a block is used by S_i but not by either of its temporal neighbor snapshots, then it exclusively belongs to S_i. In other words, the scan clears a summary bit if the bit in S_i's snapmap is set but the bits in the neighbor snapshots' snapmaps are not. The youngest snapshot has no younger neighbor and the oldest snapshot has no older neighbor in this scheme. Suppose that a second snapshot S_j is deleted while the dbys scan is in progress. A separate scan can process that S_j, and because the result is idempotent, both scans can work independently as long as they never concurrently update the same bit. If S_j happens to be S_i's neighbor, then the same scan replaces S_j with the next neighbor and continues onwards. Now the scan is updating the summary on behalf of both deletions. When the scan hits the last bit it completes S_i's deletion, but it wraps around and continues processing S_j's deletion until it gets back to where it was when S_j was deleted.

2. **Deletion by Addition (or dbya)**. This scan rebuilds the summary by simply bit-OR'ing the snapmaps of all remaining snapshots. If a second snapshot gets deleted while the scan is in progress, the scan can be restarted for the remaining snapshots. In practice, it is better for the scan to continue down the bitmap space to leverage the readaheads of the snapmaps and summary that have already been issued. When the scan gets to the last bit in the summary, it then wraps around and continues until it gets back to where the second deletion occurred.

As mentioned earlier, both modes of the delete scan can restart after a reboot without affecting correctness. Section 5.2 mentioned one interesting case. What happens if the youngest snapshot S_1 is deleted before its snap create scan is complete? Suppose that S_2 is the next youngest snapshot. It is possible that S_1 got created while the create scan for S_2 was still in progress, in which case, the scan had switched to using S_1's snapmap. Therefore, after the deletion of S_1, the create scan needs to redo S_2.

5.5 Evaluation

The CPU and I/O requirement for all these scans is somewhat straightforward; they need to load the necessary snapmap and summary blocks, and bit manipulation is not too expensive. Since the scans walk the block number space linearly, the necessary blocks can be easily prefetched in time. In theory, a given scan can be parallelized at any granularity because each bit of the summary is independent of the other. In practice, our MP programming model [6] allows for ranges of the summary to be updated concurrently. In most cases, the delete scans can be throttled to run at slow speeds so they don't interfere with client traffic; even on loaded systems, a scan of a 1 TiB sized volume (with 8K bitmap blocks) usually completes in a few minutes. In rare cases, snapshots are deleted to create space in the volume, and the scans need to run fast. We do not present the infrastructure for pacing and throttling scans; that is a larger topic.

5.5.1 Snapshot Creation

This scan can be completely eliminated thanks to Theorem 1. As explained earlier, a snapmap block of the youngest snapshot S_1 is the same exact block as the activemap block until the activemap diverges from it; and that happens when it first gets dirtied after the CP in which S_1 was created. Section 5.2 describes how the scan's work is done on-demand if an activemap bit is cleared before the scan gets to it. If this work is done instead on the first modification of the activemap block after S_1 is created, then the scan is unnecessary. This requires a last-modify timestamp on each activemap block that can be compared with S_1's creation time. WAFL increments a CP count and stores it in the superblock, and also stores it as a last-modify timestamp in the header information of each metadata block. This CP count serves as the timestamp needed to eliminate the scan.

It should be noted that the on-demand create scan work cannot be paced or throttled. It is a function of when and how bits getting cleared in the activemap. Section 4 describes how the BFLog paces the clearing of bits in the activemap in a batched manner, which indirectly also paces the on-demand create scan work.

5.5.2 Snapshot Deletion

The performance trade-off between dbys and dbya modes of snapshot deletion is obvious. The dbys scan loads and processes blocks from three snapmaps and the

summary for a deletion. When multiple snapshots are deleted, their number and temporal pattern defines the number of neighbor snapmaps loaded by the scan. On the other hand, the dbya scan loads the snapmaps of all remaining snapshots. The more efficient mode can be chosen based on the number of snapmaps needed by the scan. If more snapshots are deleted while the dbys scan is in progress, it can be aborted and a dbya scan can be kicked off. This switch is safe because the scans are idempotent. It is not possible to switch from dbya to dbys mode for reasons explained below.

The dbys mode clearly needs the deleted snapshot to be preserved (in a hidden namespace) to the extent that its snapmap can be loaded and consulted until the scan is complete. In constrast, the dbya mode does not need the deleted snapshots. Usually, a block is considered free if it is marked free in the activemap and summary. However, this invariant is violated by the dbys mode. Suppose that S_1 is deleted and the dbys scan clears the i^{th} bit of the summary block. It is possible that the block b_i in the volume happens to be a snapmap block of S_1, or an ancestor of a snapmap block in S_1's tree of blocks. If the block allocator assumes b_i to be free and writes new content to it, then the dbys scan may be rendered infeasible, or worse, corrupt the summary map. Unless the file system has some convenient way to tell whether any given block of S_1 is needed for the dbys scan, it is easier to protect all of S_1's blocks until the scan is done. This leads to:

Theorem 3. *While a dbys scan is in progress to delete one or more snapshots, if the i^{th} bit of both the activemap and summary is 0, then block b_i is free iff the i^{th} bit in the snapmap of those snapshots is also 0.*

This also means that the computation for available space in the volume has to incorporate blocks that belong to deleting snapshots while the dbys scan is in progress. This additional complexity requires more persistent data structures to solve. Therefore, dbya is chosen sometimes even when the dbys mode looks attractive from the stand point of performance. One such situation is when the volume is running low on space. Another is when the 010 snapshot property is not true; for example, WAFL allows for a file (and its blocks) to be directly restored into S_0 from a not-so-recent snapshot. This is done by simply pointing the file directly to the blocks in the snapshot instead of copying them, which violates the 010 property. Such events are timestamped so that the system can deduce when the property is restored.

6 FlexVol Virtualization Layer

WAFL uses a virtualization technique that allows hosting hundreds or more volumes (*NetApp FlexVol*®) within the same collection of physical storage media called an *aggregate*. This has been key to providing several new capabilities such as FlexVol cloning, replication, thin provisioning, etc. Section 3 of [8] explains this layering: Each aggregate is a WAFL volume with blocks called *physical volume block numbers* (or *pvbns*) that map directly to blocks on persistent storage, and each FlexVol is a WAFL volume with blocks called *virtual volume block numbers* (or *vvbns*) that are actually just blocks in a file in the aggregate WAFL volume; the file is called the *container file* for the FlexVol. In other words, the vvbn of a block in a FlexVol is the same as its block offset in the corresponding container file. The interested reader is strongly encouraged to read [8] for diagrams and details.

Each layer maintains bitmaps to track its free blocks and each layer supports snapshots. In other words, a FlexVol uses its activemap, summary map, BFLog, etc., to track allocated and free vvbns, and an aggregate uses another set of this metadata to track pvbns. After a vvbn is completely freed in the FlexVol, it is necessary to effectively punch out the physical block at the appropriate L_1 offset of the container file before the corresponding pvbn can be marked free in the aggregate's activemap. It is crucial to do this soon after the vvbn gets freed, so that all FlexVols can share the available free space in the aggregate without restriction. However, this means that freeing a random vvbn requires loading and updating a random container file L_1; this operation is expensive because the L_1 is several times less dense than a bitmap. This motivated an important performance optimization in WAFL even before the structures in Section 4 were implemented. WAFL delays the free from the container in order to batch them. WAFL maintains a count of the number of *delayed-free* blocks in a container file per 32K consecutive vvbns, which is written to a per FlexVol file in the aggregate.

There are soft limits on the total number of delayed frees so that the file system can absorb large bursts of delayed frees getting created. Deletion of snapshots that exclusively own many blocks, a volume snaprestore (described in Section 5.1), or other events can generate such bursts of delayed-frees. When these limits are exceeded, the system tries to process that backlog with some urgency. Because the BFLog batches frees in the activemap, it can choose to do the "container-punch" work if there are sufficient delayed-frees in that range.

6.1 Evaluation

We first show the importance of the delayed-free optimization. A benchmark was built in house with the read/write mix that models the query and update operations of an OLTP/benchmark application. It was built

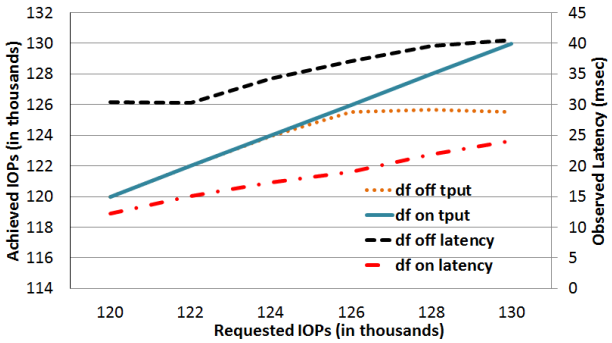


Figure 4: Benefit of delaying container frees: IOPs (left side y-axis) and latency (right side y-axis) versus input load with and without delayed frees (df) using an OLTP/database workload on a high-end all-SSD system with 20 cores and 128 GiB DRAM.

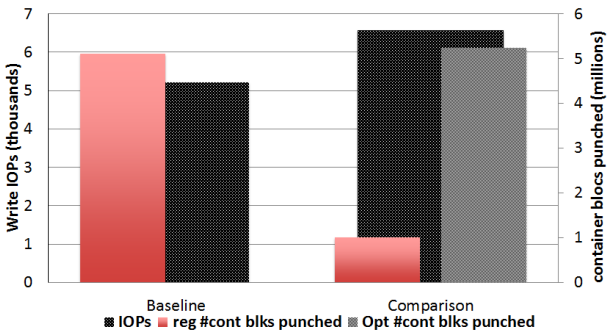


Figure 5: Benefit of the BFLog punching out the container blocks based on delayed-free counts on a low-end system.

to be very similar to the Storage Performance Council-1 (SPC-1) benchmark [5]. The benchmark was run on a high-end all-SSD system with 20 Intel Ivy Bridge cores and 128 GiB of DRAM, with and without the delayed-free optimization. Figure 4 shows the achieved IOPS (on the left y-axis) and latency (on the right y-axis). Only a small range of the input load is shown near the data points where the achieved load hits saturation and flattens out on the system without the optimization. The corresponding normalized latencies are 60% lower with delayed-frees. This shows that random container L_1 reads and updates hurt performance even on a high-end system with SSDs.

The primary source of benefit in the latencies we observed came from the greater than 60% reduction of metadata overhead with delayed-frees enabled. As a result, the CPs were able to give up more CPU cycles that could be used to service user requests at a higher rate.

Next, we study the aforementioned BFLog optimization. When the BFLog frees vvbns in the activemap, it consults the delayed-free count for that range. These delayed-frees could have accumulated from previously run BFLog or snap delete scan activity. If the total count is higher than an empirically derived threshold, the BFLog does the "container-punch" work, and appends the corresponding pvbns in the aggregate's BFLog active log. This is an incremental cost to the BFLog since it already had the FlexVol bitmaps loaded for freeing vvbns. Without the optimization, a separate background task searches for and processes these delayed-frees by loading all the FlexVol and container metadata, and that is more expensive and intrusive to client workloads.

To measure this, we ran test that randomly overwrote the data set with 32 KiB sized I/Os while FlexVol snapshots that exclusively owned a large number of blocks were deleted. This was done on a low-end system with 4 Intel Wolfdale cores and 20 GiB DRAM, which is more susceptible to this problem. Figure 5 shows the results. The client write load saw a 26% increase in throughput because, as the right-hand bar graphs show, the BFLog was able to perform the "container-punch" work much more efficiently; about 84% of it was done in the optimized mode.

7 Related Work

Originally FFS [16] started by using a linked list to maintain free blocks in the file system, which simplified the process of freeing blocks. The allocator picked up entries from this free list for allocation. However, once multiple files were created and removed, this unsorted free list grew in size, which led to random blocks being allocated by the allocator. This resulted in slower sequential reads. The designers then replaced the free list with a bitmap that identifies free blocks in a group, which resulted in better allocation but increased the cost of updating bitmaps for processing random frees. Similarly ext4 [3] also uses bitmaps for tracking free space. To make the allocation faster, it builds an in-memory buddy cache that maintains the status of clusters of 2^n blocks as free or allocated. Thus, ext4 also suffers from the problem of random frees like FFS.

The XFS [22] file system tracks free space by using two B+ trees. One B+ tree tracks space by block number and the second one by the size of the free space block. This scheme allows XFS to quickly find free space near a given block and therefore an extent of free blocks. When searching large regions of free space, a B+ tree performs better than a linear scan of a bitmap, and the second B+ tree can readily provide that information. Unfortunately, B+ trees are expensive to maintain in the face of random frees. Moreover, an insert or delete into a B+ tree can end up in a split or merge operation, which increases the cost

of a free operation. Btrfs [19] also maintains a B-tree in the form of an extent allocation tree to track allocated and free extents that serves as a persistent free space map for allocation. This looks similar to what XFS does for free space management and thus it also suffers when performing random sub-extent sized frees.

ZFS [2] tries to handle the problem of random frees by maintaining a log of allocations and frees per metaslab called a Space Map. As the random frees are appended to the log, it becomes extremely efficient because it is not necessary to load random portions of the bitmaps for updates. For allocation, the Space Map is loaded and the log entries are replayed to build an AVL tree of free space sorted by offset. ZFS compacts the persistent Space Map as needed by looking at the in-memory AVL tree. On a fairly aged file system with very little free space, the allocator has to load a lot of Space Map metadata to build this AVL tree. ZFS tries to amortize the cost of frees by logging them, but this forces the allocator to pay the cost of constructing the AVL tree.

There is clearly a trade-off that a file system can make in terms of when and how it processes a free that has been logged. Additionally, for the purposes of speeding up block allocation, file systems can choose to build complex data structures for tracking free space. However, that comes with the cost of maintaining those structures when blocks are freed, especially when they are random in their number space. Production-level file systems are usually built with specific goals in terms of workloads and features, and design principles are chosen to further those goals. They typically use a mix of persistent and in-memory free space tracking data structures that facilitate fast allocation assuming a certain buffer of free space, which lets the free space reclamation infrastructure play catch-up while the allocator looks for free blocks.

The WAFL block reclamation infrastructure stands apart because it maintains high and consistent performance while still supporting the various features of WAFL : snapshots, compression, deduplication (inline and background), FlexVol cloning, thin provisioning, file cloning, volume snaprestore, replication, single file restore, single file move on demand, etc. Unfortunately, the free space infrastructure interactions with many of these features are too lengthy to be presented here. WAFL has built-in data integrity mechanisms that protect memory-resident file system structures against scribbles and logic bugs [14]. The batching efficiency of the BFLog plays an important role in ensuring that these mechanisms work with negligible performance overhead.

The ability to absorb a high rate of reference count increments is critical to features like inline deduplication and rapid file cloning. One approach to batching refcount updates is an increment-decrement log; instead of just batching refcount decrements due to deletes, this log also collects increments created by new block sharing. This approach is especially useful in hierarchically reference counted file systems [19] since a single overwrite can generate hundreds or thousands of reference count updates [9]. This approach was explored in WAFL to support instantaneous cloning. In a hierarchically reference counted file system, where reference count updates are batched by an increment-decrement log, both the creation of and writes to cloned files can be performed efficiently [13]. By batching these updates, the decrements to block reference counts from writes can be canceled out by the increments of a clone create in the increment-decrement log without ever updating the persistent refcount file. However, there is also a need to query the log for pending increments or decrements to a particular block. Two implementations of B-trees that satisfy fast insertions and queries are presented in [1]. The queries can be satisfied in logarithmic time, and the insertions can be accomplished with guaranteed amortized updates to the B-trees.

8 Conclusion

The WAFL file system has evolved over more than two decades as technology trends have resulted in bigger and faster hardware, and larger file systems. It has also transformed from handling user directory style NFS/CIFS workloads to servicing a very wide range of SAN and NAS applications. It has been deployed to run in different configurations: purpose built platforms with all combinations of hard and solid state disks, software-defined solutions, and even NetApp Cloud Ontap®instances on AWS. The free space reclamation infrastructure has evolved to work well across all these permutations, providing a rich set of data management features and consistently high performance. Even though the pattern and rate of frees can be very random, we show that the proposed techniques allow the free space reclamation to keep up and behave deterministically without impacting the rest of the system. We describe elegant and efficient methods to track the space allocated to snapshots. Finally, we show how the infrastructure works across the extra layer of FlexVol virtualization.

9 Acknowledgement

The WAFL engineers who contributed to the designs presented in this paper are too many to list. We thank James Pitcairn-Hill, our reviewers, and our shepherd Marshall Kirk Mckusick for their invaluable feedback.

References

[1] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceeding of ACM Symposium on Parallel algorithms and architectures*, pages 81–92, 2007.

[2] Jeff Bonwick. Space maps. `https://blogs.oracle.com/bonwick/en/entry/space_maps`, 2007.

[3] Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In *Linux Symposium*, page 263, 2008.

[4] Standard Performance Evaluation Corporation. Spec sfs 2014. `https://www.spec.org/sfs2014/`.

[5] Storage Performance Council. Storage performance council-1 benchmark. `www.storageperformance.org/results/#spc1_overview`.

[6] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceeding of Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[7] Peter R. Denz, Matthew Curtis-Maury, and Vinay Devadas. Think global, act local: A buffer cache design for global ordering and parallel processing in the WAFL file system. In *Proceeding of 45th International Conference on Parallel Processing (ICPP)*, 2016.

[8] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. FlexVol: flexible, efficient file volume virtualization in WAFL. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 129–142, Jun 2008.

[9] Travis R. Grusecki. Improving block sharing in the write anywhere file layout file system. `http://hdl.handle.net/1721.1/76818`, 2012.

[10] Christopher Hertel. *Implementing CIFS: The Common Internet File System*. Prentice Hall Professional Technical Reference, 2003.

[11] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter 1994 Technical Conference*, pages 235–246, Jan 1994.

[12] NetApp Inc. Data ONTAP 8. `http://www.netapp.com/us/products/platform-os/ontap/`, 2010.

[13] Ram Kesavan, Sriram Venketaraman, Mohit Gupta, and Subramaniam Periyagaram. Systems and methods for instantaneous cloning. Patent US8812450, 2014.

[14] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2017.

[15] Avantika Mathur, Mingming Cao, and Andreas Dilger. ext4: the next generation of the ext3 file system. *Usenix Association*, 2007.

[16] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, August 1984.

[17] Sun Microsystems. ZFS at OpenSolaris community. `http://opensolaris.org/os/community/zfs/`.

[18] Peter M. Ridge and David Deming. *The Book of SCSI*. No Starch Press, San Francisco, CA, USA, 1995.

[19] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[20] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10:1–15, 1992.

[21] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation or the sun network filesystem, 1985.

[22] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *USENIX Annual Technical Conference*, 1996.

[23] Ralph H. Thornburgh and Barry Schoenborn. *Storage Area Networks*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[24] David D. Wright. Data deletion in a distributed data storage system. Patent US8819208, 2014.

NetApp, the NetApp logo, Cloud ONTAP, Data ONTAP, FlexVol, and WAFL are trademarks or registered trademarks of NetApp, Inc. in the United States and/or other countries.

Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs

Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong,
Swaminatahan Sundararaman†, Andrew A. Chien, and Haryadi S. Gunawi

University of Chicago †Parallel Machines

Abstract

TTFLASH *is a "tiny-tail" flash drive (SSD) that eliminates GC-induced tail latencies by circumventing GC-blocked I/Os with four novel strategies: plane-blocking GC, rotating GC, GC-tolerant read, and GC-tolerant flush. It is built on three SSD internal advancements: powerful controllers, parity-based RAIN, and capacitor-backed RAM, but is dependent on the use of intra-plane copyback operations. We show that* TTFLASH *comes significantly close to a "no-GC" scenario. Specifically, between 99–99.99th percentiles,* TTFLASH *is only 1.0 to 2.6× slower than the no-GC case, while a base approach suffers from 5–138× GC-induced slowdowns.*

1 Introduction

Flash storage has become the mainstream destination for storage users. The SSD consumer market continues to grow at a significant rate [8], SSD-backed cloud-VM instances are becoming the norm [7, 13], and flash/SSD arrays are a popular solution for high-end storage servers [23, 30, 44]. From the users side, they demand fast and stable latencies [25, 28]. However, SSDs do not always deliver the performance that users expect [15]. Some even suggest that flash storage "may not save the world" (due to the tail latency problem) [5]. Some recent works dissect why it is hard to meet SLAs with SSDs [36] and reveal high performance variability in 7 million hours of SSDs deployments [30].

The core problem of flash performance instability is the well-known and "notorious" *garbage collection* (GC) process. A GC operation causes long delays as the SSD cannot serve (blocks) incoming I/Os. Due to an ongoing GC, read latency variance can increase by 100× [5, 24]. In the last decade, there is a large body of work that *reduces* the number of GC operations with a variety of novel techniques [29, 35, 36, 37, 39, 43, 48]. However, we find almost no work in literature that attempts to *eliminate* the *blocking* nature of GC operations and deliver steady SSD performance in long runs.

We address this urgent issue with "tiny-tail" flash drive (TTFLASH), a GC-tolerant SSD that can deliver and guarantee stable performance. The goal of TTFLASH is to eliminate GC-induced tail latencies by *circumventing GC-blocked I/Os*. That is, ideally there should be *no* I/O that will be blocked by a GC operation, thus creating a flash storage that behaves close to a "no-GC" scenario. The key enabler is that SSD internal technology has changed in many ways, which we exploit to build novel GC-tolerant approaches.

Specifically, there are three major SSD technological advancements that we leverage for building TTFLASH. First, we leverage the increasing power and speed of today's flash controllers that *enable more complex logic* (*e.g.*, multi-threading, I/O concurrency, fine-grained I/O management) to be implemented at the controller. Second, we exploit the use of Redundant Array of Independent NAND (RAIN). Bit error rates of modern SSDs have increased to the point that ECC is no longer deemed sufficient [33, 37, 45]. Due to this increasing failure, modern commercial SSDs employ parity-based redundancies (RAIN) as a standard data protection mechanism [6, 12]. By using RAIN, we can *circumvent GC-blocked read I/Os with parity regeneration*. Finally, modern SSDs come with a large RAM buffer (hundreds of MBs) backed by "super capacitors" [10, 14], which we leverage to *mask write tail latencies* from GC operations.

The timely combination of the technology practices above enables four new strategies in TTFLASH: **(a)** *plane-blocking GC*, which shifts GC blocking from coarse granularities (controller/channel) to a finer granularity (plane level), which depends on intra-plane copyback operations, **(b)** *GC-tolerant read*, which exploits RAIN parity-based redundancy to proactively generate contents of read I/Os that are blocked by ongoing GCs, **(c)** *rotating GC*, which schedules GC in a rotating fashion to enforce at most one active GC in every plane group, hence the guarantee to always cut "one tail" with one parity, and finally **(d)** *GC-tolerant flush*, which evicts buffered writes from capacitor-backed RAM to flash pages, free from GC blocking.

One constraint of TTFLASH is its dependency on intra-plane copybacks where GC-ed pages move within a plane without the data flowing through the SSD controller, hence skipping ECC checks for garbage collected pages, which may reduce data reliability. The full extent

of this effect is not evaluated and left for future work. We recommend background ECC checks to be performed in the background to overcome this limitation (§7).

We first implemented TTFLASH in SSDSim [32] in order to simulate accurate latency analysis at the device level. Next, to run real file systems and applications, we also port TTFLASH to a newer QEMU/KVM-based platform based on VSSIM [50].

With a thorough evaluation (§6.1), we show that TTFLASH successfully eliminates GC blocking for a significant number of I/Os, reducing GC-blocked I/Os from 2–7% (base case) to only 0.003–0.7%. As a result, TTFLASH reduces tail latencies dramatically. Specifically, between the 99–99.99th percentiles, compared to the perfect no-GC scenario, a base approach suffers from 5.6–138.2× GC-induced slowdowns. TTFLASH on the other hand is only 1.0 to 2.6× slower than the no-GC case, which confirms our near-complete elimination of GC blocking and the resulting tail latencies.

We also show that TTFLASH is more stable than state-of-the-art approaches that reduce GC impacts such as preemptive GC [9, 40] (§6.2). Specifically, TTFLASH continuously delivers stable latencies while preemptive GC exhibits latency spikes under intensive I/Os. Furthermore, we contrast the fundamental difference of GC-impact elimination from reduction (§6.3, §8).

In summary, by leveraging modern SSD internal technologies in a unique way, we have successfully built novel features that provide a robust solution to the critical problem of GC-induced tail latencies. In the following sections, we present extended motivation (§2), SSD primer (§3), TTFLASH design (§4), implementation (§5), evaluation (§6), and limitations (§7), and related and conclusion (§7-9).

2 GC-Induced Tail Latency

We present two experiments that show GC cascading impacts, which motivate our work. Each experiment runs on a late-2014 128GB Samsung SM951, which can sustain 70 "KWPS" (70K of 4KB random writes/sec).

In Figure 1a, we ran a foreground thread that executes 16-KB random reads, concurrently with background threads that inject 4-KB random-write noises at 1, 2.5, and 5 KWPS (far below the max 70 KWPS) across three experiments. We measure L_i, the latency of every 16-KB foreground read. Figure 1a plots the CDF of L_i, clearly showing that *more frequent GCs (from more-intense random writes) block incoming reads and create longer tail latencies.* To show the tail is induced by GC, not queueing delays, we ran the same experiments but now with random-read noises (1, 2.5, and 5 KRPS. The read-noise results are plotted as the three overlapping thin lines marked "ReadNoise," which represents a

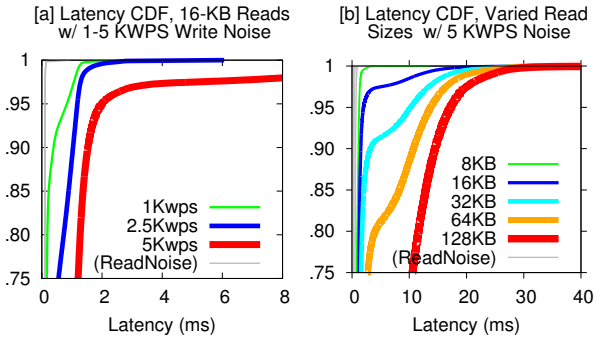


Figure 1: **GC-Induced Tail Latencies (Section 2).**

perfect no-GC scenario. As shown, with 5 KWPS noise, read operations become 15×, 19×, and 96× slower compared to no-GC scenarios, at 90th, 95th and 99th percentiles, respectively.

In Figure 1b, we keep the 5-KWPS noise and now vary the I/O size of the foreground random reads (8, 16, 32, 64, and 128 KB across five experiments). Supposedly, a 2× larger read should only consume 2× longer latency. However, the figure shows that *GC induces more tail latencies in larger reads*. For example, at 85th percentile, a 64-KB read is 4× slower than a 32-KB read. The core of the problem is this: *if a single page of a large read is blocked by a GC, the entire read cannot complete*; as read size increases, the probability of one of the pages being blocked by GC also increases, as we explain later (§3, §4.1). The pattern is more obvious when compared to the same experiments but with 5-KRPS noises (the five thin gray lines marked "ReadNoise").

For a fairer experiment, because flash read latency is typically 20× faster than write latency, we also ran read noises that are 20× more intense and another where read noises is 20× larger in size. The results are similar.

3 SSD Primer: GC Blocking

Before presenting TTFLASH, we first need to describe SSD internals that are essential for understanding GC blocking. This section describes how GC operates from the view of the physical hardware.

SSD Layout: Figure 2 shows a basic SSD internal layout. Data and command transfers are sent via *parallel channels* ($C_1..C_N$). A channel connects multiple flash *planes*; 1–4 planes can be packaged as a single chip (dashed box). A plane contains *blocks* of flash *pages*. In every plane, there is a 4-KB *register* support; all flash reads/writes must transfer through the plane register. The controller is connected to a *capacitor-backed RAM* used for multiple purposes (*e.g.*, write buffering). For clarity, we use concrete parameter values shown in Table 1.

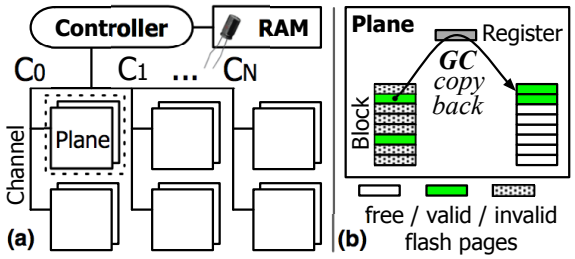


Figure 2: **SSD Internals (Section 3).**

GC operation (4 main steps): When used-page count increases above a certain threshold (*e.g.*, 70%), a GC will start. A possible GC operation reads *valid* pages from an old block, writes them to a free block, and erases the old block, within the same plane. Figure 2 shows two *copy-backs* in a GC-ing plane (two valid pages being copied to a free block). Most importantly, with 4-KB register support in every plane, page copybacks happen *within* the GC-ing plane *without* using the channel [11, 18].

The controller then performs the following *for-loop* of four steps for *every page copyback*: **(1)** send a flash-to-register read command through the channel (only 0.2μs) to the GC-ing plane, **(2)** *wait* until the plane executes the 1-page read command (40μs without using the channel), **(3)** send a register-to-flash write command, and **(4)** *wait* until the plane executes the 1-page write command (800μs without using the channel). Steps 1–4 are repeated until all valid pages are copied and then the old block is erased. The key point here is that copyback operations (steps 2 and 4; roughly 840μs) are done *internally* within the GC-ing plane *without* crossing the channel.

GC Blocking: GC blocking occurs when some resources (*e.g.*, controller, channel, planes) are used by a GC activity, which will delay subsequent requests, similar to head-of-line blocking. Blocking designs are used as they are simple and cheap (small gate counts). But because GC latencies are long, blocking designs can produce significant tail latencies.

One simple approach to implement GC is with a blocking controller. That is, even when only *one plane* is performing GC, the *controller is busy* communicating with the GC-ing plane and unable to serve outstanding I/Os that are designated to *any* other planes. We refer to this as *controller-blocking GC*, as illustrated in Figure 3a. Here, a single GC (the striped plane) blocks the controller, thus technically all channels and planes are blocked (the bold lines and dark planes). *All* outstanding I/Os cannot be served (represented by the non-colored I/Os). OpenSSD [4], VSSIM [50], and low-cost systems such as eMMC devices adopt this implementation.

Another approach is to support multi-threaded/multi-CPU with channel queueing. Here, while a thread/CPU is communicating to a GC-ing plane (in a for-loop) and

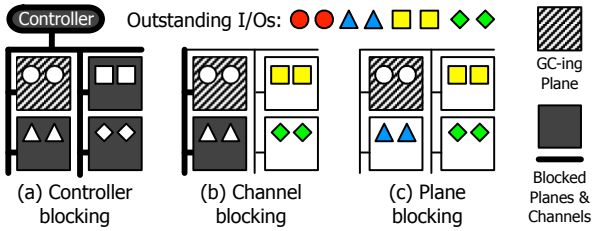


Figure 3: **Various levels of GC blocking.** *Colored I/Os in bright planes are servable while non-colored I/Os in dark planes are blocked.* **(a)** *In controller-blocking (§3), a GC blocks the controller/entire SSD.* **(b)** *In channel-blocking (§3), a GC blocks the channel connected to the GC-ing plane.* **(c)** *In plane-blocking (§4.1), a GC only blocks the GC-ing plane.*

blocking the plane's channel (*e.g.*, bold line in Figure 3b), other threads/CPUs can serve other I/Os designated to other channels (the colored I/Os in bright planes). We refer this as *channel-blocking GC* (*i.e.*, a GC blocks the channel of the GC-ing plane). SSDSim [32] and disksim$_{+SSD}$ [18] adopt this implementation. Commodity SSDs do not come with layout specifications, but from our experiments (§2), we suspect some form of channel-blocking (at least in client SSDs) exists.

Figure 1 also implicitly shows how blocked I/Os create cascading queueing delays. Imagine the "Outstanding I/Os" represents a full device queue (*e.g.*, typically 32 I/Os). When this happens, the host OS cannot submit more I/Os, hence user I/Os are blocked in the OS queues. We show this impact in our evaluation.

4 TTFLASH Design

We now present the design of TTFLASH, a new SSD architecture that achieves guaranteed performance close to a no-GC scenario. We are able to remove GC blocking at all levels with the following four key strategies:

1. Devise a non-blocking controller and channel protocol, pushing any resource blocking from a GC to only the affected planes. We call this fine-grained architecture *plane-blocking GC* (§4.1).
2. Exploit RAIN parity-based redundancy (§4.2) and combine it with GC information to proactively regenerate reads blocked by GC at the plane level, which we name *GC-tolerant read* (§4.3).
3. Schedule GC in a rotating fashion to enforce at most one GC in every plane group, such that no reads will see more than one GC; one parity can only "cut one tail." We name this *rotating GC* (§4.4).
4. Use capacitor-backed write buffer to deliver fast durable completion of writes, allowing them to be evicted to flash pages at a later time in GC-tolerant manner. We name this *GC-tolerant flush* (§4.5).

Sizes		Latencies	
SSD Capacity	256 GB	Page Read	40μs
#Channels	8	(flash-to-register)	
#Planes/channel	8	Page Write	800μs
Plane size	4 GB	(register-to-flash)	
#Planes/chip	** 1	Page data transfer	100μs
#Blocks/plane	4096	(via channel)	
#Pages/block	256	Block erase	2 ms
Page size	4 KB		

Table 1: **SSD Parameters.** *This paper uses the above parameters. (**) 1 planes/chip is for simplicity of presentation and illustration. The latencies are based on average values; actual latencies can vary due to read retry, different voltages, etc. Flash reads/writes must use the plane register.*

For clarity of description, the following sections will use concrete parameter values shown in Table 1.

4.1 Plane-Blocking GC (PB)

Controller- and channel-blocking GC are often adopted due to their simplicity of hardware implementation; a GC is essentially a `for`-loop of copyback commands. This simplicity, however, leads to severe tail latencies as independent planes are unnecessarily blocked. Channel-blocking is no better than controller-blocking GC for large I/Os; as every large I/O is typically striped across multiple channels, one GC-busy channel still blocks the entire I/O, negating the benefit of SSD parallelism. Furthermore, as SSD capacity increases, there will be more planes blocked in the same channel. Worse, GC period can be significantly long. A GC that copybacks 64 valid pages (25% valid) will lead to *54 ms* (64×840μs) of blocked channel, which potentially leaves *hundreds* of other I/Os unservable. An outstanding read operation that supposedly only takes less than 100 μs is now delayed longer by order(s) of magnitude [5, 24].

To reduce this unnecessary blocking, we introduce *plane-blocking GC*, as illustrated in Figure 3c. Here, *the only outstanding I/Os blocked by a GC are the ones that correspond to the GC-ing plane* (○ labels). All I/Os to non-GCing planes (non-○ labels) are servable, including the ones in the same channel of the GC-ing plane. As a side note, plane-blocking GC can be interchangeably defined as chip-blocking GC; in this paper, we use 1 plane/chip for simplicity of presentation.

To implement this concept, the controller must perform a fine-grained I/O management. For illustration, let us consider the four GC steps (§3). In TTFLASH, after a controller CPU/thread sends the flash-to-register read/write command (Steps 1 and 3), it will *not* be idle waiting (for 40μs and 800μs, respectively) until the next step is executable. (Note that in a common implementation, the controller is idling due to the simple for-loop and the need to access the channel to check the plane's copyback status). With plane-blocking GC, after Steps 1 and 3 (send read/write commands), the controller creates a future event that marks the completion time. The controller can reliably predict how long the intra-plane read/write commands will finish (*e.g.*, 40 and 800 μs on average, respectively). To summarize, with plane-blocking GC, TTFLASH *overlaps* intra-plane copyback and channel usage for other outstanding I/Os. As shown in Figure 3c, for the duration of an intra-plane copyback (the striped/GC-ing plane), the controller can continue serving I/Os to other non-GCing planes in the corresponding channel (▲ I/Os).

Plane-blocking GC potentially frees up hundreds of previously blocked I/Os. However, there is an unsolved GC blocking issue and a new ramification. The unsolved GC blocking issue is that the I/Os to the GC-ing plane (○ labels in Figure 3c) are *still blocked* until the GC completes; in other words, with only plane-blocking, we cannot entirely remove GC blocking. The new ramification of plane-blocking is a potentially *prolonged* GC operation; when the GC-ing plane is ready to take another command (end of Steps 2 and 4), the controller/channel might still be in the middle of serving other I/Os, due to overlaps. For example, the controller cannot start GC write (Step 3) exactly 40 μs after GC read completes (Step 1), and similarly, the next GC read (Step 1) cannot start exactly 800 μs after the previous GC write. If GC is prolonged, I/Os to the GC-ing plane will be blocked longer. Fortunately, the two issues above can be masked with RAIN and GC-tolerant read.

4.2 RAIN

To prevent blocking of I/Os to GC-ing planes, we leverage RAIN, a recently-popular standard for data integrity [6, 12]. RAIN introduces the notion of parity pages inside the SSD. Just like the evolution of disk-based RAIDs, many RAIN layouts have been introduced [33, 37, 41, 42], but they mainly focus on data protection, write optimization, and wear leveling. On the contrary, we design a RAIN layout that also targets tail tolerance. This section briefly describes our basic RAIN layout, enough for understanding how it enables GC-tolerant read (§4.3); our more advanced layout will be discussed later along with wear-leveling issues (§7).

Figure 4 shows our RAIN layout. For simplicity of illustration, we use 4 channels (C_0−C_3) and the RAIN *stripe width* matches the channel count (N=4). The planes at the same position in each channel form a *plane group* (*e.g.*, G_1). A stripe of pages is based on logical page numbers (LPNs). For every stripe (N−1 consecutive LPNs), we allocate a parity page. For example, for LPNs 0-2, we allocate a parity page P_{012}.

Regarding the FTL design (LPN-to-PPN mapping), there are two options: dynamic or static. Dynamic map-

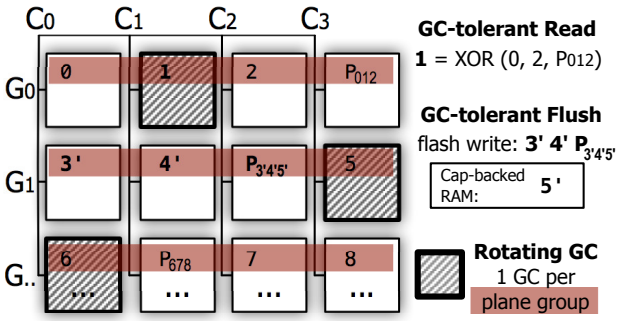


Figure 4: **TTFLASH Architecture.** *The figure illustrates our RAIN layout (§4.2), GC-tolerant read (§4.3), rotating GC (§4.4), and GC-tolerant flush (§4.5). We use four channels (C_0–C_3) for simplicity of illustration. Planes at the same "vertical" position form a plane group (G_0, G_1, etc.). A RAIN stripe is based on $N-1$ LPNs and a parity page (e.g., $012P_{012}$).*

ping, where an LPN can be mapped to *any* PPN, is often used to speed-up writes (flexible destination). However, in modern SSDs, write latency issues are absorbed by capacitor-backed RAM (§4.5); thus, writes are spread across multiple channels. Second, dynamic mapping works well when individual pages are independent; however, RAIN pages are *stripe dependent*. With dynamic mapping, pages in a stripe can be placed behind one channel, which will underutilize channel parallelism.

Given the reasons above, we create a page-level hybrid static-dynamic mapping. The static allocation policies are: **(a)** an LPN is statically mapped to a plane (*e.g.*, LPN 0 to plane G_0C_0 in Figure 4), **(b)** $N-1$ consecutive LPNs and their parity form a stripe (*e.g.*, $012P_{012}$), and **(c)** the stripe pages are mapped to planes across the channels within one plane group (*e.g.*, $012P_{012}$ in G_0). Later, we will show how all of these are crucial for supporting GC-tolerant read (§4.3) and rotating GC (§4.4).

The dynamic allocation policy is: inside each plane/chip, an LPN can be dynamically mapped to *any* PPN (hundreds of thousands of choices). An overwrite to the same LPN will be redirected to a free page in the same plane (*e.g.*, overwrites to LPN 0 can be directed to any PPN inside G_0C_0 plane).

To prevent parity-channel bottleneck (akin to RAID-4 parity-disk bottleneck), we adopt RAID-5 with a slightly customized layout. First, we treat the set of channels as a RAID-5 group. For example, in Figure 4, P_{012} and P_{345} are in different channels, in a diagonal fashion. Second, as SSD planes form a 2-dimensional layout (G_iC_j) with wearout issues (unlike disk's "flat" LPNs), we need to ensure hot parity pages are spread out evenly. To handle this, we will present dynamic migrations later (§7).

4.3 GC-Tolerant Read (GTR)

With RAIN, we can easily support *GC-tolerant read (GTR)*. For a full-stripe read (which uses $N-1$ channels), GTR is straightforward: *if a page cannot be fetched due to an ongoing GC, the page content is quickly regenerated by reading the parity from another plane*. In Figure 4, given a full-stripe read of LPNs 0–2, and if LPN 1 is unavailable temporarily, the content is rapidly regenerated by reading the parity (P_{012}). Thus, the full-stripe read is *not* affected by the ongoing GC. The resulting latency is *order(s) of magnitude faster* than waiting for GC completion; parity computation overhead only takes less than 3μs for $N \leq 8$ and the additional parity read only takes a minimum of 40+100μs (read+transfer latencies; Table 1) and does not introduce much contention.

For a partial-stripe read (R pages where $R<N-1$), GC-tolerant read will generate in total $N-R$ *extra* reads; the worst case is when $R=1$. These $N-R$ extra parallel reads will add contention to each of the $N-R$ channels, which might need to serve other outstanding I/Os. Thus, we only perform extra reads if $T_{GCtoComplete} > B \times (40+100)\mu$s where B is the number of busy channels in the $N-R$ extra reads (for non-busy channels the extra reads are free). In our experience, this simple policy cuts GC tail latencies effectively and fairly without introducing heavy contention. In the opposing end, a "greedy" approach that always performs extra reads causes high channel contention.

We emphasize that unlike tail-tolerant speculative execution, often defined as an optimization task that may *not* be actually needed, GC-tolerant read is *affirmative*, not speculative; the controller knows exactly when and where GC is happening and how long it will complete. GTR is effective but has a limitation: it does *not* work when *multiple* planes in a plane group perform GCs simultaneously, which we address with rotating GC.

4.4 Rotating GC (RGC)

As RAIN distributes I/Os evenly over all planes, multiple planes can reach the GC threshold and thus perform GCs simultaneously. For example, in Figure 4, if planes of LPNs 0 and 1 (G_0C_0 and G_0C_1) both perform GC, reading LPNs 0–2 will be delayed. The core issue is: one parity can only cut "one tail". Double-parity RAIN is not used due to the larger space overhead.

To prevent this, we develop *rotating GC (RGC)*, which enforces that *at most one plane in each plane group can perform one GC at a time*. Concurrent GCs in different plane groups are still allowed (*e.g.*, one in each G_i as depicted in Figure 4). Note that rotating GC depends on our RAIN layout that ensures every stripe to be statically mapped to a plane group.

We now emphasize our most important message: *there will be zero GC-blocked I/Os if rotating GC holds true all the time*. The issue here is that our rotating approach can delay a plane's GC as long as $(N-1)\times T_{gc}$ (the GC duration). During this period, when all the free pages are exhausted, *multiple* GCs in a plane group *must* execute concurrently. This could happen depending on the combination of N and the write intensity. Later in Appendix A, we provide a proof sketch showing that with stripe-width $N \leq 26$, rotating GC can always be enforced under realistic write-intensive scenarios.

Employing a large stripe width (*e.g.*, $N=32$) is possible but can violate rotating GC, implying that GC tail latencies cannot be eliminated all the time. Thus, in many-channel (*e.g.*, 32) modern SSDs, we can keep $N=8$ or 16 (*e.g.*, create four 8-plane or two 16-plane groups across the planes within the same vertical position). Increasing N is unfavorable not only because of rotating GC violation, but also due to reduced reliability and the more extra I/Os generated for small reads by GTR (§4.3). In our evaluation, we use $N=8$, considering 1/8 parity overhead is bearable.

4.5 GC-Tolerant Flush (GTF)

So far, we only address read tails. Writes are more complex (*e.g.*, due to write randomness, read-and-modify parity update, and the need for durability). To handle write complexities, we leverage the fact that flash industry heavily employs *capacitor-backed RAM* as a durable write buffer (or "cap-backed RAM" for short) [14]. To prevent data loss, the RAM size is adjusted based on the capacitor discharge period after power failure; the size can range from tens to hundreds of MB, backed by "super capacitors" [10].

We adopt cap-backed RAM to absorb all writes quickly. When the buffer occupancy is above 80%, a *background flush* will run to evict some pages. When the buffer is full (*e.g.*, due to intensive large writes), a *foreground flush* will run, which will *block* incoming writes until some space is freed. The challenge to address here is that foreground flush can induce write tails when the evicted pages must be sent to GC-ing planes.

To address this, we introduce *GC-tolerant flush (GTF)*, which ensures that *page eviction is free from GC blocking, which is possible given rotating GC*. For example, in Figure 4, pages belonging to 3', 4' and $P_{3'4'5'}$ can be evicted from RAM to flash while page 5' eviction is delayed until the destination plane finishes the GC. With proven rotating GC, GTF can evict $N-1$ pages in every N pages per stripe without being blocked. Thus, the minimum RAM space needed for the pages yet to be flushed is small. Appendix A suggests that modern SSD RAM size is sufficient to support GTF.

For partial-stripe writes, we perform the usual RAID read-modify-write eviction, but still without being blocked by GC. Let us imagine a worst-case scenario of updates to pages 7' and 8' in Figure 4. The new parity should be $P_{67'8'}$, which requires read of page 6 first. Despite page 6 being unreachable, it can be regenerated by reading the old pages P_{678}, 7, and 8, after which pages 7', 8', and $P_{67'8'}$ can be evicted.

We note that such an expensive parity update is rare as we prioritize the eviction of full-stripe dirty pages to non-GCing planes first and then full-stripe pages to mostly non-GCing planes with GTF. Next, we evict partial-stripe dirty pages to non-GCing planes and finally partial-stripe pages to mostly non-GCing planes with GTF. Compared to other eviction algorithms that focus on reducing write amplification [35], our method adds GC tail tolerance.

5 Implementation

This section describes our implementations of TTFLASH, which is available on our website [1].

• `ttFlash-Sim` **(SSDSim):** To facilitate accurate latency analysis at the device level, we first implement TTFLASH in SSDSim [32], a recently-popular simulator whose accuracy has been validated against a real hardware platform. We use SSDSim due to its clean-slate design. We implemented all the TTFLASH features by adding 2482 LOC to SSDSim. This involves a substantial modification (+36%) to the vanilla version (6844 LOC). The breakdown of our modification is as follow: plane-blocking (523 LOC), RAIN (582), rotating GC (254), GC-tolerant read (493) and write (630 lines).

• `ttFlash-Emu` **("VSSIM++"):** To run Linux kernel and file system benchmarks, we also port TTFLASH to VSSIM, a QEMU/KVM-based platform that "facilitates the implementation of the SSD firmware algorithms" [50]. VSSIM emulates NAND flash latencies on RAM disk. Unfortunately, VSSIM's implementation is based on 5-year old QEMU-v0.11 IDE interface, which only delivers 10K IOPS. Furthermore, as VSSIM is a single-threaded design, it essentially mimics a controller-blocking SSD (1K IOPS under GC).

These limitations led us to make major changes. First, we migrated VSSIM's single-threaded logic to a multi-threaded design within the QEMU AIO module, which enables us to implement channel-blocking. Second, we migrated this new design to a recent QEMU release (v2.6) and connected it to the PCIe/NVMe interface. Our modification, which we refer as *"VSSIM++"*, can sustain 50K IOPS. Finally, we port TTFLASH features to VSSIM++, which we refer as `ttFlash-Emu`, for a total of 869 LOC of changes.

• **Other attempts (OpenSSD and LightNVM):** We attempted implementing TTFLASH on real hardware platforms (2011 Jasmine and 2015 Cosmos OpenSSD boards [4]). After a few months trying, we hit many limitations of OpenSSD: single threaded (no pthread support), single logic for all channels (cannot control channel queues), no accessible commands for data transfer from flash RAM to host DRAM (preventing parity regeneration), no support for wall-clock time (preventing GC time prediction), inaccessible request queues and absence of GC queues (OpenSSD is whole-blocking). We would like to reiterate that these are not hardware limitations, but rather, the ramifications of the elegant simplicity of OpenSSD programming model (which is its main goal). Nevertheless, our conversations with hardware architects suggest that TTFLASH is implementable on a real firmware (*e.g.*, roughly a 1-year development and testing project on a FPGA-based platform).

Finally, we also investigated the LightNVM (OpenChannel SSD) QEMU test platform [16]. LightNVM [21] is an in-kernel framework that manages OpenChannel SSD (which exposes individual flash channels to the host, akin to Software-Defined Flash [44]). Currently, neither OpenChannel SSD nor LightNVM's QEMU test platform support intra-SSD copy-page command. Without such support and since GC is managed by the host OS, GC-ed pages must cross back and forth between the device and the host. This creates heavy background-vs-foreground I/O transfer contention between GC and user I/Os. For example, the user's maximum 50K IOPS can downgrade to 3K IOPS when GC is happening. We leave this integration for future work after the intra-SSD copy-page command is supported.

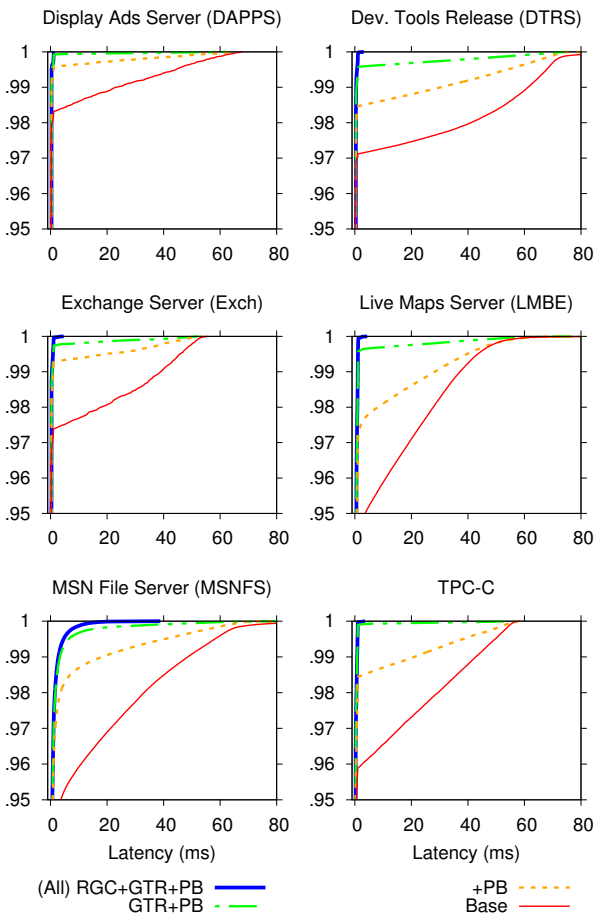


Figure 5: **Tail Latencies.** *The figures show the CDF of read latencies (x=0–80ms) in different workloads as we add each* TTFLASH *strategy: +PB (§4.1), +GTR (§4.3), and +RGC (§4.4). The y-axis shows 95–100th percentiles.*

6 Evaluation

We now present extensive evaluations showing that TTFLASH significantly eliminates GC blocking (§6.1), delivers more stable latencies than the state-of-the-art preemptive GC (§6.2) and other GC optimization techniques (§6.3), and does not significantly increase P/E cycles beyond the RAIN overhead (§6.4).

Workloads: We evaluate two implementations: `ttFlash-Sim` (on SSDSim) and `ttFlash-Emu` (on VSSIM++), as described in Section 5. For `ttFlash-Sim` evaluation, we use 6 real-world block-level traces from Microsoft Windows Servers as listed in the figure titles of Figure 5. Their detailed characteristics are publicly reported [3, 34]. By default, for each trace, we chose the busiest hour (except the 6-minute TPCC trace). For `ttFlash-Emu` evaluation, we use filebench [2] with six personalities as listed in the x-axis of Figure 8.

Hardware parameters: For `ttFlash-Sim`, we use the same 256-GB parameter values provided in Table 1 with 64 MB cap-backed RAM and a typical device queue size of 32. `ttFlash-Emu` uses the same parameters but its SSD capacity is only 48 GB (limited by the machine's DRAM). We use a machine with 2.4GHz 8-core Intel Xeon Processor E5-2630-v3 and 64-GB DRAM. The simulated and emulated SSD drives are pre-warmed up with the same workload.

6.1 Main Results

• **Tiny tail latencies:** Figure 5 shows the CDF of read latencies from the six trace-driven experiments run on `ttFlash-Sim`. Note that we only show read latencies; write latencies are fast and stable as all writes are absorbed by cap-backed RAM (§4.5). As shown in Figure 5, the base approach ("`Base`" = the default SSDSim with channel-blocking and its most-optimum FTL [32] and without RAIN) exhibits long tail latencies. In contrast, as we add each TTFLASH feature one at a time on top of the other: `+PB` (§4.1), `+GTR` (§4.3), and `+RGC` (§4.4), sig-

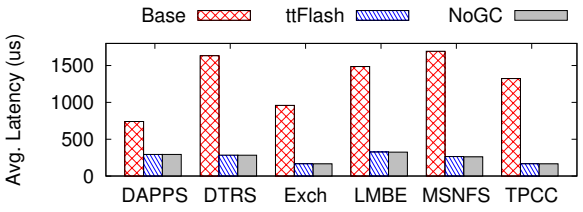


Figure 6: **Average Latencies.** *The figure compares the average read latencies of* `Base`, TTFLASH, *and* `NoGC` *scenarios from the same experiments in Figure 5.*

Percentile:	DAP	DTRS	Exch	LMBE	MSN	TPCC
99.99^{th}	1.00x	1.24	1.18	1.96	1.00	2.56
99.9^{th}	1.00x	1.01	1.01	1.02	1.01	1.01
99^{th}	1.00x	1.02	1.10	1.01	1.03	1.02

Table 2: **TTFLASH vs. NoGC (almost no tail).** *The numbers above represent the* ***slowdown ratio*** *of* TTFLASH *read latencies compared to* `NoGC` *at high percentiles. For example, in DTRS, at 99.99^{th} percentile,* TTFLASH*'s read latency is only 1.24× slower than* `NoGC`*'s read latency.*

nificant improvements are observed. When all features are added (`RGC+GTR+PB`), the tiny tail latencies are close to those of the no-GC scenario, as we explain later.

Figure 6 plots the average latencies of the same experiments. This graph highlights that although the latencies of TTFLASH and `Base` are similar at 90^{th} percentile (Figure 5), the `Base`'s long tail latencies severely impact its average latencies. Compared to `Base`, TTFLASH's average latencies are 2.5–7.8× faster.

• **TTFLASH vs. NoGC:** To characterize the benefits of TTFLASH's tail latencies, we compare TTFLASH to a perfect "no-GC" scenario ("`NoGC`" = TTFLASH without GC and with RAIN). In `NoGC`, the same workload runs without any GC work (with a high GC threshold), thus all I/Os observe raw flash performance.

Table 2 shows the slowdown from `NoGC` to TTFLASH at various high percentiles. As shown, TTFLASH significantly reduces GC blocking. Specifically, at 99–99.9^{th} percentiles, TTFLASH's slowdowns are only *1.00* to *1.02*×. Even at 99.99^{th} percentile, TTFLASH's slowdowns are only *1.0* to *2.6*×. In comparison, `Base` suffers from 5.6–138.2× slowdowns between 99–99.99^{th} percentiles (as obvious in Figure 5); for readability, `NoGC` lines are not plotted in the figure. In terms of average latencies, Figure 6 shows that TTFLASH performs the same with or without GC.

• **GC-blocked I/Os:** To show what is happening inside the SSD behind our speed-ups, we count the percentage of read I/Os that are blocked by GC (*"%GC-blocked I/Os"*), as plotted in Figure 7. As important, we emphasize that GC-blocked I/Os fill up the device queue, creating queueing delays that prevent new host I/Os from entering the device, which we count as *"%queue-blocked I/Os."* Thus, each bar in the figure has two parts: %GC-blocked (bottom, bold edge) and %queue-blocked I/Os (top), divided with a small horizontal borderline.

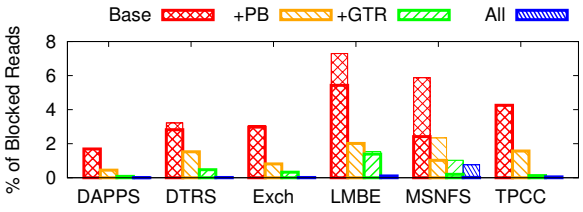


Figure 7: **%GC-blocked read I/Os.** *The figure above corresponds to the results in Figure 5. The bars represent the ratio (in percent) of read I/Os that are GC-blocked (bottom bar) and queue-blocked (top bar) as explained in §6.1. "All" implies PB+GTR+RGC (please see Figure 5's caption).*

Figure 7 shows that with `Base`, *without* GC tolerance, 2–5% of reads are blocked by GC. As they further cause queueing delays, in total, there are *2–7% of blocked I/Os* that cannot be served. As each TTFLASH feature is added, more I/Os are unblocked. With *all* the features in place ("`All`" bars), there are only *0.003–0.05% of blocked I/Os*, with the exception of MSNFS (*0.7%*). The only reason why it is not 0% is that for non-full-stripe reads, TTFLASH will wait for GC completion *only if* the remaining time is shorter than the overhead of the extra reads (as explained in §4.3). We still count these I/Os as blocked, albeit only momentarily.

We next evaluate `ttFlash-Emu` with filebench [2]. Figure 8 shows the average latencies of filebench-level read operations (including kernel, file-system, and QEMU overheads in addition to device-level latencies) and the percentage of GC-blocked reads measured inside `ttFlash-Emu`. We do not plot latency CDF as filebench only reports average latencies. Overall, `ttFlash-Emu` shows the same behavior as `ttFlash-Sim`.

6.2 TTFLASH vs. Preemptive GC

As mentioned before, many existing work optimize GC, but does not eliminate its impact. One industry standard in eliminating ("postponing") GC impact is preemptive GC [9]. We implement preemptive GC in SSDSim based on existing literature [40]. The basic idea is to interleave user I/Os with GC operations. That is, if a user I/O arrives while a GC is happening, future copybacks should be postponed.

Figure 9a compares `ttFlash-Sim`, preemptive, and `NoGC` scenarios for the DTRS workload (other workloads lead to the same conclusion). As shown, TTFLASH is closer to `NoGC` than preemptive GC. The reason is that preemptive GC must incur a delay from waiting for

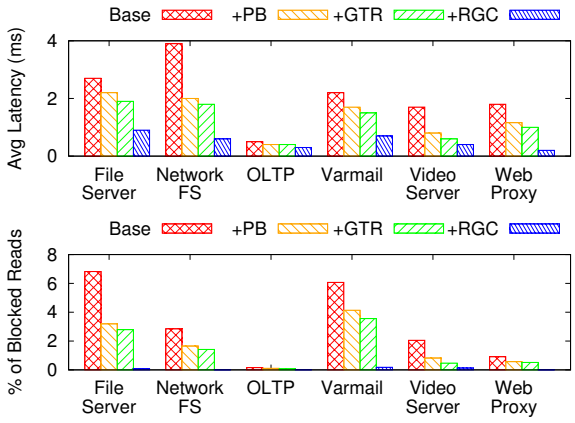


Figure 8: **Filebench on `ttFlash-Emu`.** *The top and bottom figures show the average latencies of read operations and the percentage of GC-blocked reads, respectively, across six filebench personalities. "`Base`" represents our VSSIM++ with channel-blocking (§5).*

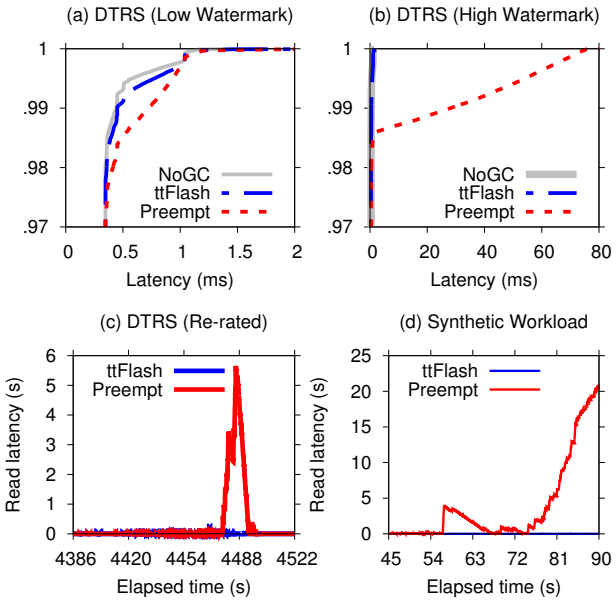


Figure 9: **TTFLASH vs. Preemptive GC.** *The figures are explained in Section 6.2.*

the block erase (up to 2 ms) or the current page copyback to finish (up to 800 μs delay), mainly because the finest-grained preemption unit is a page copyback (§3). TTFLASH on the other hand can rapidly regenerate the delayed data.

Most importantly, TTFLASH does *not* postpone GC indefinitely. In contrast, preemptive GC *piles up* GC impact to the future, with the hope that there will be idle time. However, with a continuous I/O stream, at one point, the SSD will hit a GC high water mark (not enough free pages), which is when preemptive GC becomes non-preemptive [40]. To create this scenario, we run the same workload but make SSDSim GC threshold hit the high water mark. Figure 9b shows that as preemptive GC becomes non-preemptive, it becomes GC-intolerant and creates long tail latencies.

To be more realistic with the setup, we perform a similar experiment as in the Semi-Preemptive GC paper [40, §IV]. We re-rate DTRS I/Os by 10× and re-size them by 30×, in order to reach the high GC water mark (which we set to 75% to speed up the experiment). Figure 9c shows the timeline of observed latencies with TTFLASH and preemptive GC. We also run a synthetic workload with continuous I/Os to prevent idle time (Figure 9d); the workload generates 28-KB I/Os (full-stripe) every 130μs with 70% read and 30% write). Overall, Figures 9c–d highlight that preemptive GC creates backlogs of GC activities, which will eventually cause SSD "lock-down" when page occupancy reaches the high watermark. On the other hand, TTFLASH can provide stable latencies without postponing GC activities indefinitely.

The last two experiments above create high intensity of writes, and within the same experiments, our GC-tolerant flush (GTF; §4.5) provides stable latencies, as implicitly shown in Figures 9c–d.

6.3 TTFLASH vs. GC Optimizations

GC can be optimized and reduced with better FTL management, special coding, novel write buffer scheme or SSD-based log-structured file system. For example, in comparison to base approaches, Value locality reduces erase count by 65% [29, Section 5], flash-aware RAID by 40% [33, Figure 20], BPLRU by 41% [35, Section 4 and Figure 7], eSAP by 10-45% [37, Figures 11–12], F2FS by 10% [39, Section 3], LARS by 50% [41, Figure 4], and FRA by 10% [42, Figure 12], SFS by 7.5× [43, Section 4], WOM codes by 33% [48, Section 6].

Contrary to these efforts, our approach is fundamentally different. We do not focus in reducing the number of GCs, but instead, we eliminate the blocking nature of GC operations. With reduction, even if GC count is reduced by multiple times, it only makes GC-induced tail latencies shorter, but not disappear (*e.g.*, as in Figure 5). Nevertheless, the techniques above are crucial in extending SSD lifetime, hence orthogonal to TTFLASH.

6.4 Write (P/E Cycle) Overhead

Figure 10 compares the number of GCs (P/E cycles) completed by the `Base` approach and TTFLASH within the experiments in Figure 5. We make two observations. First, TTFLASH does not delay GCs; it actively performs GCs at a similar rate as in the base approach, but yet still delivers predictable performance. Second, TTFLASH introduces 15–18% of additional P/E cycles (in 4 out of 6 workloads), which mainly comes from RAIN; as we use N=8, there are roughly 15% (1/7) more writes in

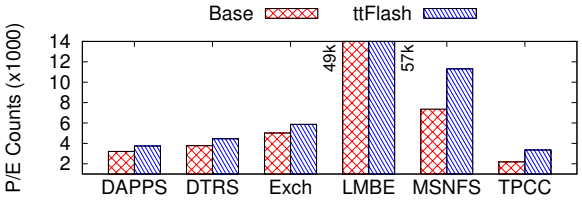


Figure 10: **GC Completions (P/E Cycles).** *The figure is explained in Section 6.4.*

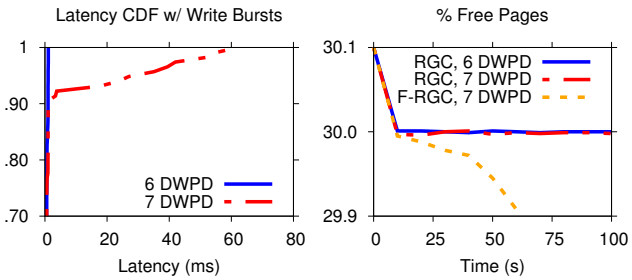


Figure 11: **TTFLASH under Write Bursts.** *The figure is explained in Section 6.6.*

minimum, from one parity write for every seven ($N-1$) consecutive writes. The exceptions are 53% of additional P/E cycles in MSNFS and TPCC, which happen because the workloads generate many small random writes, causing one parity write for almost every write. For this kind of workload, large buffering does not help. Overall, higher P/E cycles is a limitation of TTFLASH, but also a limitation of any scheme that employs RAIN.

6.5 TTFLASH vs. No RAIN

Earlier, in Figure 6, we show that TTFLASH has about the same average latencies as NoGC (TTFLASH without GC and *with* RAIN). In further experiments (not shown due to space), we also compare TTFLASH to "$\texttt{NoGC}_{!R}$" (*i.e.*, Base without GC and *without* RAIN). We observed TTFLASH's average latencies are 1.09–1.33× of $\texttt{NoGC}_{!R}$'s. The RAIN-less $\texttt{NoGC}_{!R}$ is faster because it can utilize all channels. This is a limitation of TTFLASH; that is, as TTFLASH (or any SSD that) employs RAIN, the channels experience a slight contention. In Figure 4 for example, reading LPNs 0–3 will incur contention on channel-0 (from LPNs 0 and 3). In a RAIN-less setup, the same read will utilize all four channels.

6.6 TTFLASH under Write Bursts

TTFLASH can circumvent GC blocking when rotating GC is enforced (§4.4). A limitation of TTFLASH is that under heavy write bursts, multiple GCs per plane group must be allowed to keep the number of free pages stable. Figure 11a shows the limit of our 256GB drive setup (Table 1) with N=8. As shown, at 6 DWPD (55 MB/s), there is almost no GC-blocked reads, hence tiny tail latencies. 1 DWPD ("Drive Writes Per Day") implies 256GB/8hours (9.1 MB/s) of writes; we generously use 8 hours to represent a "Day" (Appendix A). However, at 7 DWPD (64 MB/s), TTFLASH exhibits some tail latencies, observable at the 90^{th} percentile. We emphasize that this is still much better than the Base approach, where the tail latencies are observed starting at 20^{th} percentile (not shown). We also believe that such intensive writes are hopefully rare; for 3-5yr lifespans, modern MLC/TLC drives must conform to 1-5 DWPD [17]. Figure 11b shows that if we force only one GC per plane group all the time ("F-RGC"), at 7 DWPD, the percentage of free pages (the y-axis) continuously drops over time (the x-axis). That is, RGC cannot keep up with the write bursts. Thus, to keep the number of free pages stable, under write bursts, we must allow multiple GCs to happen per plane group (the "RGC, 7DWPD" line).

7 Limitations and Discussions

We now summarize the limitations of TTFLASH. First, TTFLASH depends on RAIN, hence the loss of one channel per N channels (as evaluated in §6.5). Increasing N will reduce channel loss but cut less tail latencies under write bursts (Appendix A). Under heavy write bursts, TTFLASH cannot cut all tails (as evaluated in §6.6 and discussed in Appendix A). Finally, TTFLASH requires intra-plane copybacks, skipping ECC checks, which requires future work as we address below.

• **ECC checking** *(with scrubbing)*: ECC-check is performed when data pass through the ECC engine (part of the controller). On foreground reads, before data is returned to the host, ECC is *always* checked (TTFLASH does *not* modify this property). Due to increasing bit errors, it is suggested that ECC checking runs more frequently, for example, by forcing all background GC copyback-ed pages read out from the plane and through the controller, albeit reduced performance.

TTFLASH, however, depends on *intra-plane* copybacks, which implies *no* ECC checking on copyback-ed pages, potentially compromising data integrity. A simple possible solution to compensate this problem is periodic idle-time scrubbing within the SSD, which will force flash pages (user and parity) flow through the ECC engine. This is a reasonable solution for several reasons. First, SSD scrubbing (unlike disk) is fast given the massive read bandwidth. For example, a 2 GB/s 512-GB client SSD can be scrubbed under 5 minutes. Second, scrubbing can be easily optimized, for example, by only selecting blocks that are recently GC-ed or have higher P/E counts and history of bit flips, which by implication can also reduce read disturbs. Third, periodic background operations can be scheduled without affecting

foreground performance (a rich literature in this space [19]). However, more future work is required to evaluate the ramifications of background ECC checks.

• **Wear leveling** *(via horizontal shifting and vertical migration)*: Our static RAIN layout (§4.2) in general does not lead to wear-out imbalance in common cases. However, rare cases such as random-write transactions (*e.g.*, MSNFS) cause imbalanced wear-outs (at chip/plane level).

Imbalanced wear-outs can happen due to the two following cases: (1) There is write imbalance *within* a stripe (MSNFS exhibits this pattern). In Figure 4 for example, if in stripe S_0 $\{012P_{012}\}$, LPN 1 is more frequently updated than the rest, the planes of LPN 1 and P_{012} will wear out faster than the other planes in the same group. (2) There is write imbalance *across* the stripes. For example, if stripes in group G_0 (*e.g.*, stripe $\{012P_{012}\}$) are more frequently updated than stripes in other groups, then the planes in G_0 will wear out faster.

The two wear-out problems above can be fixed by *dynamic* horizontal shifting and vertical migration, respectively. With horizontal shifting, we can shift the parity locations of stripes with imbalanced hot pages. For example, S_0 can be mapped as $\{12P_{012}0\}$ across the 4 planes in the same group; LPN 1 and P will now be directed to colder planes. With vertical migration, hot stripes can be migrated from one plane group to another ("vertically"), balancing the wear-out across plane groups.

As a combined result, an LPN is still and always statically mapped to a stripe number. A stripe, by default, is statically mapped to a plane group and has a static parity location (*e.g.*, S_0 is in group G_0 with P_{012} behind channel C_3). However, to mark dynamic modification, we can add a "mapping-modified" bit in the standard FTL table (LPN-PPN mapping). If the bit is zero, the LPN-PPN translation performs as usual, as the stripe mapping stays static (the common case). If the bit is set (the rare case in rare workloads), the LPN-PPN translation must consult a new stripe-information table that stores the mapping between a stripe (S_k) to a group number (G_i) and parity channel position (C_j).

8 Related Work

We now discuss other works related to TTFLASH.

GC-impact reduction: Our work is about eliminating GC impacts, while many other existing works are about reducing GC impacts. There are two main reduction approaches: *isolation* and *optimization*, both with drawbacks. First, isolation (*e.g.*, OPS isolation [36]) only isolates a tenant (*e.g.*, sequential) from another one (*e.g.*, random-write). It does not help a tenant with both random-write and sequential workloads on the same dataset. OPS isolation must differentiate users while TTFLASH is user-agnostic. Second, GC optimization, which can be achieved by better page layout management (*e.g.*, value locality [29], log-structured [23, 39, 43]) only helps in reducing GC period but does not eliminate blocked I/Os.

GC-impact elimination: We are only aware of a handful of works that attempt to eliminate GC impact, which fall into two categories: without or with redundancy. Without redundancy, one can eliminate GC impact by *preemption* [22, 40, 47]. We already discussed the limitations of preemptive GC (§6.2; Figure 9). With redundancy, one must depend on RAIN. To the best of our knowledge, our work is the first one that leverages SSD internal redundancy to eliminate GC tail latencies. There are other works that leverage redundancy in flash array (described later below).

RAIN: SSD's internal parity-based redundancy (RAIN) has become a reliability standard. Some companies reveal such usage but unfortunately without topology details [6, 12]. In literature, we are aware of only four major ones: eSAP [37], PPC[33], FRA [42] and LARS [41]. These efforts, however, mainly concern about write optimization and wear leveling in RAIN but do not leverage RAIN to eliminate GC tail latencies.

Flash array: TTFLASH works within a single SSD. In the context of SSD array, we are aware of two published techniques on GC tolerance: Flash on Rails [46] and Harmonia [38]. Flash on Rails [46] eliminates read blocking (read-write contention) with a ring of multiple drives where 1–2 drives are used for write logging and the other drives are used for reads. The major drawback is that read/write I/Os cannot utilize the aggregate bandwidth of the array. In Harmonia [38], the host OS controls all the SSDs to perform GC at the same time (*i.e.*, it is better that all SSDs are "unavailable" at the same time, but then provide stable performance afterwards), which requires more complex host-SSD communication.

Storage tail latencies: A growing number of works recently investigated sources of storage-level tail latencies, including background jobs [19], file system allocation policies [31], block-level I/O schedulers [49], and disk/SSD hardware-level defects [26, 27, 30]. An earlier work addresses load-induced tail latencies with RAID parity [20]. Our work specifically addresses GC-induced tail latencies.

9 Conclusion

SSD technologies have changed rapidly in the last few years; faster and more powerful flash controllers are cable of executing complex logic; parity-based RAIN has become a standard means of data protection; and capacitor-backed RAM is a de-facto solution to address write inefficiencies. In our work, we leverage a combina-

tion of these technologies in a way that has not been done before. This in turn enables us to build novel techniques such as plane-blocking GC, rotating GC, GC-tolerant read and flush, which collectively deliver a robust solution to the critical problem of GC-induced tail latencies.

10 Acknowledgments

We thank Sam H. Noh (our shepherd), Nisha Talagala, and the anonymous reviewers for their tremendous feedback. This material is based upon work supported by the NSF (grant Nos. CCF-1336580, CNS-1350499, CNS-1526304, CNS-1405959, and CNS-1563956) as well as generous donations from EMC, Google, Huawei, NetApp, and CERES Research Center.

A Proof Sketch

Limitation of maximum stripe width (N): We derive the maximum stripe width allowable (N) such that rotating GC (§4.4) is always enforced. That is, as we can only cut one tail, there should be *at most one GC per plane group* at all time. Thus, a plane might need to *postpone* its GC until other planes in the same group complete their GCs (*i.e.*, delayed by $(N-1)\times T_{gc}$). We argue that N should be at least 8 for a reasonable parity space overhead (12.5%); a lower stripe width will increase space overhead. Below we show that N=8 is safe even under *intensive* write. Table 3 summarizes our proof, which is based on a *per-plane, per-second* analysis. We first use concrete values and later generalize the proof.

• Table 3a: We use typical parameters: 4-KB page (S_{page}), 4-KB register size (S_{reg}), 25% valid pages ($\%_{validPg}$), 840μs of GC copyback time per page ($T_{copyback}$), and 900μs of user write latency per page (T_{usrWrt}). Due to intensive copybacks (tens of ms), the 2ms erase time is set to "0" for proving simplicity.

• Table 3b: Each plane's bandwidth (BW_{pl}) defines the maximum write bandwidth, which is 4.5 MB/s, from the register size (S_{reg}) divided by the user-write latency (T_{usrWrt}); all writes must go through the register.

• Table 3c: With the 4.5 MB/s maximum plane bandwidth, there are 1152 pages written per second ($\#W_{pg/s}$), which will eventually be GC-ed.

• Table 3d: *Intensive* writes imply frequent overwrites; we assume 25% valid pages ($\%_{validPg}$) to be GC-ed, resulting in 288 pages copybacked per second ($\#CB_{pg/s}$). The $\%_{validPg}$ can vary depending on user workload.

• Table 3e: With 288 page copybacks, the total GC time per second per plane ($T_{gc/s}$) is 242 ms.

• Table 3f: N planes in each group must *finish* their GCs in rotating manner. As each plane needs T_{gc} time every second, **the constraint is:** $N<1/T_{gc}$. With our concrete values above, for rotating GC to hold true all the time, N must be less than 4 (T_{gc} of 242 ms). *Fortunately, N can be larger in practice* (Table 3g-j). To show this, below we first generalize the proof.

a.	S_{page}=4KB; S_{reg}=4KB; $\%_{validPg}$=25%; T_{prog}=800μs; T_{read}=40μs; $T_{channel}$=100μs; $T_{copyback}$=T_{prog}+T_{read}=840μs; (T_{erase}="0"); T_{usrWrt}=T_{prog}+$T_{channel}$=900μs;		
b.	BW_{pl} =	S_{reg}/T_{usrWrt}	=4.5 MB/s
c.	$\#W_{pg/s}$ =	BW_{pl}/S_{page}	=1152 pg/s
d.	$\#CB_{pg/s}$ =	$\%_{validPg}\times \#W_{pg/s}$	=288 pg/s
e.	$T_{gc/s}$ =	$\#CB_{pg/s}\times T_{copyback}$	=242 ms
f.	$N <$	$1 / T_{gc}$	<4
g.	$N <$	$\frac{S_{page}}{BW_{plane}\times\%_{validPg}\times T_{copyback}}$	
h.	$DWPD$=5; $PWPD$=5; S_{pl}=4GB; day=8hrs		
i.	BW_{pl} =	$S_{pl}\times DWPD/day$	(in practice)
	=	$4GB \times 5\ /8hrs$	=0.7 MB/s
j.	$T_{gc/s}$ =	plug (i) to (c,d,e)	=38 ms
	$N <$	$1 / T_{gc}$	$<$ **26**

Table 3: **Proof Sketch (Appendix A).**

• Table 3g: We combine all the equations above to the equation in Table 3g, which clearly shows that N goes down if BW_{pl} or $\%_{validPg}$ is high. Fortunately, we find that the *constant* 4.5 MB/s throughput (BW_{pl}) in Table 3b is *unrealistic* in practice, primarily due to *limited SSD lifetime*. MLC block is only limited to about 5000–10,000 erase cycles and TLC block 3000 erase cycles. To ensure multi-year (3–5) lifespan, users typically conform to the *Drive Writes Per Day (DWPD)* constraint (1–5 DWPD for MLC/TLC drives) [17].

• Table 3h: Let us assume a worst-case scenario of 5 DWPD, which translates to 5 $PWPD$ (planes write per day) per plane. To make it worse, let us assume a "day" is 8 hours. We set plane size (S_{pl}) to 4 GB (§3).

• Table 3i: The more realistic parameters above suggest that a plane only receives 0.7 MB/s (4GB*5/8hrs), which is 6.5× less intense than the raw bandwidth (3b).

• Table 3j: If we plug in 0.7 MB/s to the equations in Table 3c-e, the GC time per plane (T_{gc}) is only 38 ms, which implies that N **can be as large as 26**.

In conclusion, N=8 is likely to always satisfy rotating GC in practice. In 32-channel SSD, N=32 can violate rotating GC; GC-tolerant read (§4.3) cannot always cut the tails. Overall, Table 3g defines the general constraint for N. We believe the most important value is BW_{pl}. The other parameters relatively stay the same; S_{page} is usually 4 KB, $\%_{validPg}$ is low with high overwrites, and $T_{copyback}$ can increase by 25% in TLC chips (vs. MLC).

Minimum size of cap-backed RAM: With rotating GC, the RAM needs to only hold at most $1/N$ of the pages whose target planes are GC-ing (§4.5). In general, the minimum RAM size is $1/N$ of the SSD maximum write bandwidth. Even with an extreme write bandwidth of the latest datacenter SSD (*e.g.*, 2 GB/s) the minimum RAM size needed is only 256 MB.

References

[1] http://ucare.cs.uchicago.edu/projects/tinyTailFlash/.

[2] Filebench. http://filebench.sourceforge.net/wiki/index.php/Main_Page.

[3] SNIA IOTTA: Storage Networking Industry Association's Input/Output Traces, Tools, and Analysis. http://iotta.snia.org.

[4] The OpenSSD Project. http://www.openssd-project.org.

[5] Google: Taming The Long Latency Tail - When More Machines Equals Worse Results. http://highscalability.com/blog/2012/3/12/google-taming-the-long-latency-tail-when-more-machines-equal.html, 2012.

[6] The Crucial M550 SSD. http://www.crucial.com/usa/en/storage-ssd-m550, 2013.

[7] New SSD-Backed Elastic Block Storage. https://aws.amazon.com/blogs/aws/new-ssd-backed-elastic-block-storage/, 2014.

[8] Report: SSD market doubles, optical drive shipment rapidly down. http://www.myce.com/news/report-ssd-market-doubles-optical-drive-shipment-rapidly-down-70415/, 2014.

[9] Sandisk: Pre-emptive garbage collection of memory blocks. https://www.google.com/patents/US8626986, 2014.

[10] Supercapacitors have the power to save you from data loss. http://www.theregister.co.uk/2014/09/24/storage_supercapacitors/, 2014.

[11] L74A NAND datasheet. https://www.micron.com/parts/nand-flash/mass-storage/mt29f256g08cmcabh2-12z, 2015.

[12] Micron P420m Enterprise PCIe SSD Review. http://www.storagereview.com/micron_p420m_enterprise_pcie_ssd_review, 2015.

[13] Microsoft Rolls Out SSD-Backed Azure Premium Cloud Storage. http://www.eweek.com/cloud/microsoft-rolls-out-ssd-backed-azure-premium-cloud-storage.html, 2015.

[14] What Happens Inside SSDs When the Power Goes Down? http://www.army-technology.com/contractors/data_recording/solidata-technology/presswhat-happens-ssds-power-down.html, 2015.

[15] Why SSDs don't perform. http://www.zdnet.com/article/why-ssds-dont-perform/, 2015.

[16] Open-Channel Solid State Drives. http://lightnvm.io/, 2016.

[17] What's the state of DWPD? endurance in industry leading enterprise SSDs. http://www.storagesearch.com/dwpd.html, 2016.

[18] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[19] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic Storage Maintenance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[20] Yitzhak Birk. Random RAIDs with Selective Exploitation of Redundancy for High Performance Video Servers. In *Proceedings of 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 1997.

[21] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[22] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time Garbage Collection for Flash-memory Storage Systems of Real-time Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(4), November 2004.

[23] 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.

[24] Jeffrey Dean and Luiz Andr Barroso. The Tail at Scale. *Communications of the ACM*, 56(2), February 2013.

[25] 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 21st ACM SIGOPS symposium on Operating systems principles (SOSP)*, 2007.

[26] 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.

[27] 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.

[28] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[29] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging Value Locality in Optimizing NAND Flash-based SSDs. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST)*, 2011.

[30] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[31] 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.

[32] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity. In *Proceedings of the 25th International Conference on Supercomputing (ICS)*, 2011.

[33] Soojun Im and Dongkun Shin. Flash-Aware RAID Techniques for Dependable and High-Performance Flash Memory SSD. *IEEE Transactions on Computers (TOC)*, 60(1), October 2010.

[34] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[35] Hyojun Kim and Seongjun Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)*, 2008.

[36] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[37] Jaeho Kim, Jongmin Lee, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Improving SSD Reliability with RAID via Elastic Striping and Anywhere Parity. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2013.

[38] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. Harmonia: A Globally Coordinated Garbage Collector for Arrays of Solid-State Drives. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2011.

[39] Changman Lee, Dongho Sim, Joo-Young 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)*, 2015.

[40] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. A Semi-Preemptive Garbage Collector for Solid State Drives. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[41] Sehwan Lee, Bitna Lee, Kern Koh, and Hyokyung Bahn. A Lifespan-aware Reliability Scheme for RAID-based Flash Storage. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC)*, 2011.

[42] Yangsup Lee, Sanghyuk Jung, and Yong Ho Song. FRA: A Flash-aware Redundancy Array of Flash Storage Devices. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System (CODES+ISSS)*, 2009.

[43] 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)*, 2012.

[44] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage System. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[45] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[46] 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.

[47] 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.

[48] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. Write Once, Get 50% Free: Saving SSD Erase Costs Using WOM Codes. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[49] 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.

[50] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choi, Sungroh Yoon, and Jaehyuk Cha. VSSIM: Virtual Machine based SSD Simulator. In *Proceedings of the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2013.

The Logic of Physical Garbage Collection in Deduplicating Storage

Fred Douglis[*], Abhinav Duggal[*], Philip Shilane[*], Tony Wong[*], Shiqin Yan[*†], Fabiano Botelho[+]
[*]*Dell EMC* [†]*University of Chicago* [+]*Rubrik, Inc.*

Abstract

Most storage systems that write in a log-structured manner need a mechanism for *garbage collection* (GC), reclaiming and consolidating space by identifying unused areas on disk. In a deduplicating storage system, GC is complicated by the possibility of numerous references to the same underlying data. We describe two variants of garbage collection in a commercial deduplicating storage system, a *logical* GC that operates on the files containing deduplicated data and a *physical* GC that performs sequential I/O on the underlying data. The need for the second approach arises from a shift in the underlying workloads, in which exceptionally high duplication ratios or the existence of millions of individual small files result in unacceptably slow GC using the file-level approach. Under such workloads, determining the liveness of chunks becomes a slow phase of logical GC. We find that physical GC decreases the execution time of this phase by up to two orders of magnitude in the case of extreme workloads and improves it by approximately 10–60% in the common case, but only after additional optimizations to compensate for its higher initialization overheads.

1 Introduction

Since the advent of log-structured file systems (LFS) [20], there has been work to optimize the cost of "cleaning" the file system to consolidate live data and create large contiguous areas of free space [15]. Most past efforts in this area have been to optimize I/O costs, as any effort to read and rewrite data reduces the throughput available for new data. With deduplicating storage systems [17, 28] there is an additional complication, that of identifying what data is live in the first place. As new data are written to a system, duplicate *chunks* are replaced with references to previously stored data, so it is essential to track each reference.

Further, as workloads evolve, some systems experience very different usage than traditional deduplicating backup storage systems were intended to support [1]. The Data Domain File System (DDFS) [29] was designed to handle a relatively low number (thousands) of relatively large files (GBs), namely the full and incremental backups that have been the mainstay of computer backups for decades [2, 24]. Data in these backups would be deduplicated, with the remaining content packed into "compression regions" that would be further reduced via standard compression such as Lempel-Ziv (LZ). *Total compression* (TC) is the product of these two factors; DDFS was optimized for TC in the $20-40\times$ range, because of the number of "full backups" a system would typically store. This has been changing dramatically in some environments, with technology trends increasing the deduplication ratio as well as the numbers of files represented in storage systems (§2).

DDFS uses a *mark-and-sweep* [27] algorithm that determines the set of live chunks reachable from the live files and then frees up unreferenced space. There are alternatives such as reference counting [8], but as we discuss in §6, complexity and scalability issues have led to our current approach.

We initially performed garbage collection at the *logical* level, meaning the system analyzed each live file to determine the set of live chunks in the storage system. The shift to using individual file-level backups, rather than tar-like aggregates, meant that the number of files in some systems increased dramatically. This results in high GC overhead during the *mark* phase, especially due to the amount of random I/O required. At the same time, the high deduplication ratios in some systems cause the same live chunks to be repeatedly identified, again greatly increasing GC overhead. The time to complete a single cycle of GC in such systems could be on the order of several days. Since backing up data concurrently with GC results in contention for disk I/O and processing, there is a significant performance implication to such long GC cycles; in addition, a full system might run out of capacity while awaiting space to be reclaimed.

Therefore, we redesigned GC to work at the *physical* level: instead of GC *enumerating* all live files and their referenced chunks, entailing random access, GC performs a series of sequential passes through the *storage containers* containing numerous chunks [17]. Be-

cause the I/O pattern is sequential and because it scales with the physical capacity rather than the deduplication ratio or the number of files, the overhead is relatively constant and proportional to the size of the system [11].

This paper describes the two versions of our garbage collection subsystem, logical and physical GC (respectively LGC and PGC). A detailed description of LGC is useful both for understanding its shortcomings and because this was the version used within the commercial product for many years. We recognize that since LGC was first implemented, there have been several publications describing other GC systems in detail [8, 9, 21, 22], and we view the technical contributions of this paper to be the insights leading to the new and greatly improved PGC subsystem. PGC has undergone an evolution, starting with a change to the order of container access and then being further optimized to lower memory usage and avoid the need for multiple passes over the data. We compare LGC to the earlier implementation of PGC, which has been deployed at customer sites for an extended time, and to the newer "phase-optimized physical GC" (PGC$^+$), incorporating additional optimizations.

In summary, the contributions of this paper include:

- A detailed description of two approaches to garbage collection in a deduplicating storage system.
- An analysis of the changing workloads that have caused the previous approach to be replaced by a new GC algorithm whose enumeration time scales with the physical capacity of the system rather than the logical (pre-deduplication) capacity or the number of files.
- A comparison of GC performance on deployed systems that upgraded from LGC to PGC, demonstrating up to a 20× improvement in enumeration times.
- A detailed comparison of the performance of the various GC algorithms in a controlled environment, demonstrating up to a 99× improvement in enumeration times.

In the remainder of the paper, we provide additional motivation into the problem of scalable GC (§2) and then describe the two GC algorithms, logical and physical GC (§3). §4 describes our evaluation methodology, and §5 analyzes customer systems to compare the techniques in the field and lab testbeds to compare them in controlled environments. §6 discusses related work, and we conclude with final remarks (§7).

2 Background and Motivation

Freeing unreferenced space is a basic storage system operation. While there are multiple ways space management is implemented in traditional storage, when a file is deleted, blocks referenced from its inodes can be freed immediately by marking a free bitmap or updating a free list. For deduplicated storage, determining which chunks are referenced has added complexity as a chunk may have numerous references both within a single file and across many files written at various times. While some deduplication research assumed a FIFO deletion pattern [8], file deletion can generally be in any order.

There are a number of properties to consider when designing and evaluating a garbage collection algorithm:

- All referenced chunks must be preserved so user data can be retrieved.
- Most unreferenced chunks must be removed to free space.
- The system must support client reads and writes during garbage collection.
- System overheads should be minimized.

We specifically state that *most* unreferenced chunks must be removed instead of *all*, since it is often more efficient to focus cleaning on containers that are mostly unreferenced, rather than relocating numerous live chunks to reclaim a small amount of space. This is particularly relevant in log-structured storage systems commonly used by deduplicated storage, which tend to exhibit a bimodal distribution of container liveness [19], where containers tend to be mostly dead or mostly live. We see similar distributions in our workloads.

2.1 Deletion Algorithms

2.1.1 Reference Counts

An intuitive technique to determine when chunks are referenced is to maintain a count for each chunk. When a duplicate chunk is discovered during the write-path, a counter is incremented, and when a file is deleted, counters are decremented; at zero, the chunk can be freed. While this could take place in-line as files are written and deleted [25], it is more common to implement at least partially-offline reference counts [8, 22]. Strzelczak et al. [22] implemented an epoch-based deletion system with counters to support concurrent writes and deletes. We discuss the drawbacks of reference counts, such as snapshot overheads, in related work (§6).

2.1.2 Mark-and-Sweep

Another approach is to run an asynchronous algorithm to determine which chunks are referenced by live files. Mark-and-sweep proceeds in two phases. The first phase walks all of the live files and *marks* them in a data structure. The second phase scans the containers and *sweeps* unreferenced chunks. Guo et al. [9] implemented a Grouped Mark and Sweep to mark referenced containers. While the *sweep* phase is often the slowest phase [5]

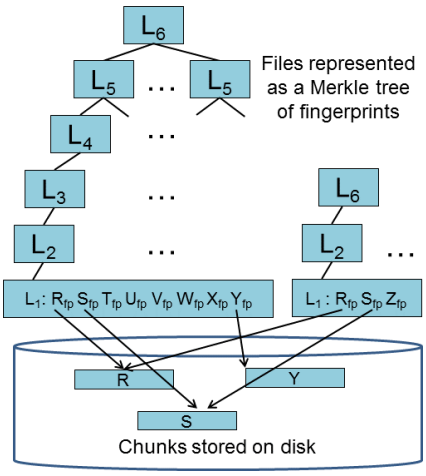


Figure 1: Files in deduplicated storage are often represented with a tree of fingerprints.

because live chunks are read from disk and written to new containers, we have found that the *mark* phase can be the bottleneck in newer systems because enumerating live fingerprints involves random disk I/O. There are also complexities regarding data being deduplicated against chunks in the process of being garbage collected (§3.5).

2.2 Deduplicated File Representation

Figure 1 shows our use of Merkle trees [16] for deduplicated storage. We refer to chunks directly written by users as L_0, meaning the lowest level of the tree. Consecutive L_0 chunks are referenced with an array of fingerprints by an L_1 chunk, which itself is identified by a fingerprint. An array of L_1 fingerprints is referenced by an L_2 chunk, continuing to the root of the tree; the root is always labeled L_6 for convenience, even if the file is small enough not to need intermediate nodes such as the example on the right side of the figure. We refer to the L_1-L_6 chunks as L_p chunks, where p is a parameter that ranges from 1 to 6 and indicates metadata representing the file. Deduplication takes place because a chunk can be referenced multiple times. The file system is a forest of Merkle trees, but these trees are not disjoint, particularly at the lowest level.

Though not shown in the figure, L_p chunks are themselves stored on disk in containers, which include a relatively small (hundreds of KB) metadata section with a list of fingerprints for the chunks within the container. Thus they may be read more quickly than the full container.

As an example, consider a system with 100TB of physical capacity, 20× TC, 8KB L_0 chunks, and 20-byte fingerprints. The logical capacity, *i.e.*, the storage that can be written by applications, is 2PB ($20 * 100$TB). Since each 8KB logically written by a client requires a 20-byte fingerprint stored in an L_1, the L_1 chunks are 5TB. The upper levels of the tree (L_2–L_6) are also stored in containers but are smaller. This example highlights that the *mark* phase cannot be fully performed in memory, as the L_1 references must be read from disk.

Data Domain supports a *fastcopy* [7] system command to efficiently copy an existing file using the same underlying Merkle tree. It creates the new file with a new name, and therefore a new L_6 root of the tree, but that tree then references the identical L_P chunks. As this operation involves only the root of the tree, it is trivially fast and does not increase physical space in use beyond the one chunk containing the L_6.

2.3 Performance Issues with Enumeration

The *mark* phase of mark-and-sweep involves enumerating all of the live fingerprints referenced from live files and marking a data structure for each fingerprint. A number of issues affect performance:

- **Deduplication and compression**. The enumeration cost grows with logical space; some datasets have extremely high TC, making the logical space very large and unreasonably slow to enumerate. Note that deduplication can vary considerably, while LZ compression is usually within a small range (2–4).
- **Number of files** in the system and file size distribution. Every file has metadata overhead to be processed when enumerating a file, and for small files, the overhead may be as large as the enumeration time itself. Also, if the system issues parallel enumeration threads at the file level (processing several files at once), then one extremely large file that cannot be further parallelized might slow overall enumeration, though such skew would be rare.
- **Spatial locality** of the L_p tree. Traversing the L_p tree stops at the L_1 level since all L_0 references are recorded in L_1 chunks. Fragmentation of L_p chunks on disk increases the amount of random I/O needed for enumeration [12, 13]. While sequentially written files tend to have high locality of L_p chunks, creating a new file from incremental changes harms locality as enumeration has to jump among branches of the L_p tree.

2.4 Technology Trends and Enumeration

Two main technology trends increase enumeration time [1]. First, clients are creating many frequent backups by writing the incremental changes such as the changed blocks or files since the previous backup. Since only incremental changes are transferred, backups are faster. *Synthetic* full backups are created by copying a previous full backup and applying the user's changes

since it was created, which is an efficient operation with support within deduplicated storage. A new file is generated by referencing unmodified portions of the previous backup, causing deduplication ratios to increase. This allows a user to view and restore a full backup without the need to restore an earlier full backup and apply intervening incremental changes: the traditional backup strategy of daily incrementals and weekly full backups. For a given period of time, this means backup storage systems may now contain numerous full backups (with the corresponding high deduplication ratio) instead of numerous incremental backups (with a lower deduplication ratio). (Unlike *fastcopy*, synthetic full backups create a full new L_P tree; thus the overhead for storing the metadata is about 1% of the logical size of the data.)

As the speed of backup completion has increased, and applications such as databases desire more frequent point-in-time backups, these backups have become more frequent (multiple times per day). Similarly, applications may take frequent snapshots on deduplicated storage that effectively make a virtual copy of the data [2]. As a result of these technology trends, some systems see deduplication ratios as much as $100-300\times$ or higher (with correspondingly high TC), whereas $10\times$ deduplication was typical for traditional backup environments [24].

A second technology trend is the increase in the file count. This arises from shifts in usage as described above and from the use of deduplication to support backups of individual files rather than aggregates (similar to *tar* files) that represent an entire backup as a single file [24]. It also can arise from users accessing a deduplicating appliance via another interface like NFS, treating it more like primary storage than backup.

As TC and number of small files increase, enumeration time becomes a large component of overall garbage collection. In order to support these new workloads and continue to scale performance, we developed a new *mark* algorithm that replaces logical enumeration in depth-first order with physical enumeration that proceeds in breadth-first order with sequential disk scans.

3 Architecture

This section describes the original LGC algorithm and the changes to enable PGC and then PGC$^+$. One issue common to both forms of GC is how to deal with online updates to the state of the file system, which we discuss after the various algorithms (§3.5).

3.1 Logical GC

Data Domain's original garbage collection technique is logical, using a mark-and-sweep algorithm. We briefly give an overview of the algorithm before describing each phase in more detail. The first step involves walking through all of the file L_p trees and recording the chunk fingerprints in a Bloom filter [18] as chunks that are live. We use a Bloom filter since it reduces memory requirements compared to an index of fingerprints, though even the Bloom filter is quite large (§3.1.2). GC then "sweeps" away any chunks that are unreferenced by reading in container metadata regions and determining whether the chunks are recorded in the Bloom filter. Container metadata regions are shown in Figure 2 and contain a list of fingerprints for chunks within the multi-megabyte container. Live chunks are copied forward into new containers, and old containers are deleted.

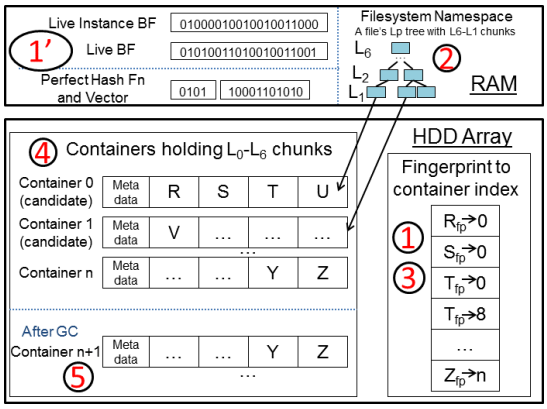


Figure 2: Components of LGC and PGC. The red numbers correspond to GC phases (*1'* applies only to PGC).

A fingerprint match in the Bloom filter could be a false positive, which results in a dead chunk being misidentified as live and retained, but such examples would be rare; these can be bounded by sizing the Bloom filter to limit its false positive rate. Also, the hash functions used by the Bloom filter are changed between runs so that false positives are likely to be removed the next time. Note that live chunks will never be misidentified as dead.

3.1.1 LGC Steps

LGC proceeds in five phases shown in Figure 2: Merge, Enumeration, Filter, Select, and Copy. The first four collectively implement *mark* while the Copy phase is *sweep*. A second set of these phases may be necessary (§3.1.2).

1. Merge: This synchronizes recent index updates from memory to disk so that the next phase can iterate over the on-disk index. In this phase, we also take a snapshot of the file system so we can clean from a consistent viewpoint, which consists of copying the root directory of the file system name space.

2. Enumeration: To identify the live chunks, we enumerate all of the files referenced from the root. Enumeration proceeds in a depth-first order from the top level (L_6) and progresses down to the bottom interior node (L_1).

Enumeration stops at L_1, because it consists of fingerprints referencing data chunks (L_0) that are leaves of the tree. We record the fingerprints for live files in the Live Bloom Filter, indicating that the corresponding chunks should be preserved.

3. Filter: Because the storage system may contain a small[1] number of duplicate chunks stored in containers (*i.e.*, not perfectly deduplicated such as T_{fp} in the fingerprint index of Figure 2), we next determine which instance of the live fingerprints to preserve. The fingerprint-to-container index has the complete list of fingerprints in the system; its entries are organized into buckets of size 512 bytes to 384KB. We then iterate through the index. While any instance of a duplicate fingerprint could be preserved, our default policy is to preserve the instance in the most recently written container, which will prioritize read performance for the most recently written data over older backups [12, 13]. We use the *Live Bloom Filter* to track the existence of a chunk in the system and the Live *Instance* Bloom Filter to track the most recent instance of each chunk in the presence of duplicates. Thus, if a fingerprint exists in the Live Bloom filter, we record the combination of <fingerprint, container_ID> (specifically, the XOR of the two values) in the Live Instance Bloom Filter.

4. Select: We next estimate the liveness of containers to focus cleaning on those where the most space can be reclaimed with the least effort. Since containers tend to either be mostly live or mostly dead, we can reclaim the most space by focusing on mostly dead containers [20]. We iterate through the containers reading the metadata region holding chunk fingerprints. We calculate the <fingerprint, container_ID> value and check the Live Instance Bloom Filter. The liveness of each container is calculated as the number of live fingerprints divided by the total number of fingerprints in the container. We sort the liveness records in memory and select containers below a liveness threshold set dynamically based on our target goal for space to reclaim.

5. Copy: New containers are formed from live chunks copied out of previous containers. When a new container is full, it is written to disk, and dead containers are freed.

6. Summary (not shown): In systems where a system-wide Bloom filter is used to avoid wasted index lookups on disk [29], the Bloom filter is rebuilt to reflect the current live chunks. Its time is approximately the same as the Select phase and is typically dominated by the copy phase. As the Summary phase is being completely eliminated in future versions of DDFS [1], we omit it from further discussion.

While we have presented this algorithm as linear, there are opportunities for parallelism within phases such as reading multiple files during enumeration, multiple index buckets during filtering, and multiple containers during the Select and Copy phases.

3.1.2 Memory Optimization

As described, this algorithm assumes that there is sufficient memory for the Bloom filters to have a low false positive rate. Depending on the system configuration, memory may be limited; a Bloom filter tracking 200B (billion) entries would be hundreds of GBs in size, depending on the desired false positive rate. For such systems, we developed a technique to focus the cleaning algorithm on containers with the most dead chunks.

Before running the four phases described previously, we set a sampling rate based on the memory available for the Bloom filter and the number of chunks currently stored in the system. We then run the *mark* phases (Enumeration, Filter, and Select) but only consider fingerprints that match the sampling criteria before inserting into the Bloom filters. During the Select phase, we choose candidate containers that are mostly dead, but we also limit the total number of containers based on our memory limitation. We then create a Candidate Bloom Filter covering those containers by reading the container metadata regions of the candidate containers and inserting the fingerprints into the Candidate Bloom Filter. The above steps then are repeated, with the exception of Select, though limited by the Candidate Bloom Filter.[2] As an example, the Enumeration phase only adds live fingerprints to the Live Bloom Filter if the fingerprint is in the Candidate Bloom Filter. While this introduces a third Bloom filter, we only need two simultaneously because the Candidate Bloom Filter can be freed after the Enumeration phase. The result is that we apply cleaning in a focused manner to the candidate containers.

However, although the *sweep* phase of actually copying and cleaning containers usually dominates overhead, the need for two sets of *mark* phases adds considerable processing time. Typically, the more data stored on an appliance, the more likely the system will need two sets of *mark* phases. A system that is largely empty will still have enough DRAM to support GC on a loaded system, so the Bloom filters for the *mark* phase may fit in memory. Empirically, we find that about half of GC runs require two phases: typically, systems start less loaded, and over time they store more data and exceed the threshold. In a study of EMC Data Domain systems in 2011, Chamness found that half the systems would be expected to reach full capacity within about six months of the point studied [6].

[1] In practice, we find that about 10% of the data removed during GC is due to duplicates rather than unreferenced chunks.

[2] We refer to the first set of phases as *pre* phases, such as *pre-enumeration*.

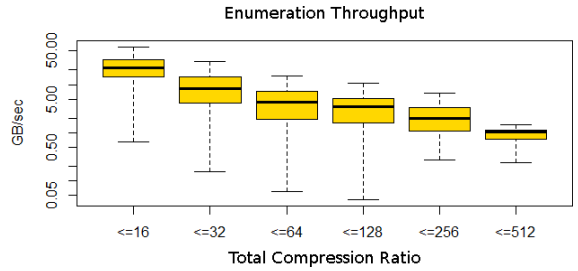


Figure 3: LGC enumeration throughput decreases as TC, bucketed into ranges, increases. Shown for the DD990 on a log-log scale.

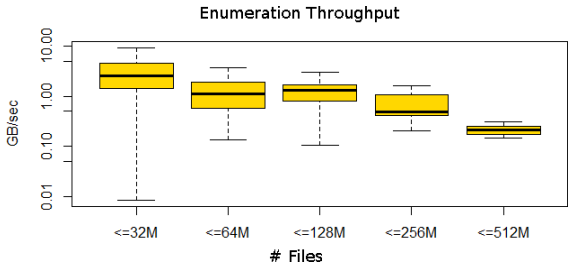


Figure 4: LGC enumeration throughput decreases with high file counts as a function of the number of files, bucketed into ranges. Shown for the DD2500 on a log-log scale.

3.1.3 The Need for Faster Enumeration

While logical enumeration has been successfully used in production systems for over a decade, technology trends discussed in §2 are increasing the total compression ratio and the number of small files. This creates a larger name space and increases enumeration time. Figure 3 shows the LGC enumeration throughput as a function of TC for thousands of deployed systems, using one of our largest available models, the DD990[3] (analysis of a smaller system shows similar results). We define enumeration throughput as GB of physical storage used in the system, divided by GC enumeration time; this normalizes GC performance based on the capacity in use. We show the enumeration range for a bucket with box-and-whisker plots: the whiskers show the 90th and 10th percentile, while the box shows the 75th and 25th, with the median result drawn as a horizontal line within the box. Throughput decreases steadily with TC, demonstrating that LGC enumeration time is related to the logical size of the dataset rather than its physical size. We show this more explicitly in lab experiments in §5.

In Figure 4, we show enumeration throughput versus file counts. We select a smaller system, the DD2500, which is the model[4] with the most files; we see that throughput decreases steadily with higher file counts. We therefore developed PGC, which replaces random I/O for logical enumeration with sequential I/O for physical enumeration. More details about the dataset used to create these figures are presented in §4.1.

3.2 Physical GC

PGC differs from LGC in several ways. First, we use *perfect hashing* [3, 14] (PH) as an auxiliary data structure to drive the breadth-first traversal of the forest of live files. Second, a new method of enumeration identifies live chunks from the containers rather than by iterating through each file. The version of GC that is currently generally in use runs PGC, and as we will describe below, it improves enumeration overhead for the new usage patterns, high deduplication and large numbers of individual files. Then, to eliminate some moderate performance degradation for the traditional use cases, we add additional optimizations (see §3.4).

3.2.1 Perfect Hashing

Determining whether a fingerprint is referenced by a live file can be represented as a membership query, and PH is a data structure to minimize memory requirements. Briefly, PH consists of a hash function that uniquely maps from a key to a position in a hash vector, called a Perfect Hash Vector (PHV). The hash function is generated by analyzing the static set of fingerprints under consideration, and the PHV is used to record membership by setting specific bit(s) assigned to each fingerprint. Previous work has focused on generating a PH function while reducing memory/analysis overheads [3,4,14]. Our technique builds directly from Botelho et al., which used PH in Data Domain to trade off analysis time and memory for secure deletion [5].

As shown in Figure 2, we allocate memory for a PH function and PHV. There is one PH function for a set of fingerprints; using the function, we get the position of the bit for a fingerprint in the PHV. We use PHs to represent the L_6 through L_1 layers of the file tree during enumeration for multiple reasons. First, the false positive rate associated with a Bloom filter would cause the entire branch of chunks under an incorrectly-labeled L_p chunk to remain in the system. Second, our PH is more compact than a Bloom filter, hence it uses less memory for representing these levels. Third, PH assists in level-by-level checksum matching to ensure that there is no corruption (§3.2.2).

[3] Specifications for Data Domain systems appear in §4.

[4] We do not have a good explanation for why customers would use this model to store extremely large numbers of files, more so than other platforms, as no Data Domain system is optimized for that sort of workload. However, if systems with many small files do not require large physical capacities, it would be natural to use lower-end systems.

3.2.2 PGC Steps

PGC has a new Analysis phase (labeled phase 1' in Figure 2) but otherwise has the same phases as LGC, with changes to the Enumeration phase. Again, *mark* is everything but Copy.

1. Merge: This is the same as LGC.

1'. Analysis: We create the PH function by analyzing the fingerprints in the on-disk index. The result is the unique mapping from fingerprint to offset in a PHV. The PH function and vector, in total, use less than 4 bits per L_p fingerprint.

2. Enumeration: Unlike LGC, instead of walking the file tree structure, we perform a series of sequential container scans. We first record the live L_6 fingerprints in the PHV based on files referenced in the namespace by setting a walk bit. Our system has an in-memory structure that records which L_p types (L_6, L_5, ... L_0) exist in each container, so we can specifically scan containers with L_6 chunks. If the L_6 is marked as walk in the PHV, we mark the confirm bit, and we then parse the L_5 fingerprints stored in the L_6 and set the walk bits for those entries in the PHV. We then repeat these steps scanning for L_5 chunks, L_4 chunks, etc. until L_1 chunks are read. When setting the walk bit for any chunk (L_6-L_1), we also record the fingerprint in the Live Bloom Filter, as in LGC.

While it is possible for a container to be read multiple times because it holds chunks of multiple types (e.g. both L_6s and L_5s), the dominant cost comes from L_1 containers, which are read just once. Thus we can decrease the random I/O and overall run time compared to logical enumeration of files, especially with high file counts and deduplication ratios.

3. Filter, **4. Select**, **5. Copy:** These follow LGC.

The correctness of a garbage collection algorithm is critical to prevent data loss. First, GC has to ensure that no chunks are missed during enumeration. Second, if there is already a corruption, GC should stop and alert an administrator to attempt data recovery. The problem is more severe due to deduplication, as a corrupted L_p chunk can result in a large number of files being corrupted due to sharing of the L_p trees. To meet these requirements, physical enumeration calculates checksums per L_P level by making use of the PHV. Due to space considerations we cannot include details, but we briefly describe the approach.

We use two checksums for each level, one for parent chunks (P) and another for child chunks (C). When processing a chunk at level L_k, we XOR its array of referenced hashes for level $L_{k\text{-}1}$ and add it to child checksum C of level $L_{k\text{-}1}$. When processing the chunk at level $L_{k\text{-}1}$, we add the chunk's hash to the P checksum for $L_{k\text{-}1}$. If there is no corruption, at the end of the $L_{k\text{-}1}$ scan, P and C for $L_{k\text{-}1}$ should match. The checksums are calculated and matched for all L_1-L_6 levels, and using the PH function prevents including a hash value repeatedly. Before calculating an XOR, we check whether a chunk is already included in the checksum by checking the appropriate bit in the PHV. We use a PH function that maps from a chunk hash value to two bits, representing 1) finding a chunk as a member of its parent's array of hashes and 2) finding the chunk itself at the next lower level.

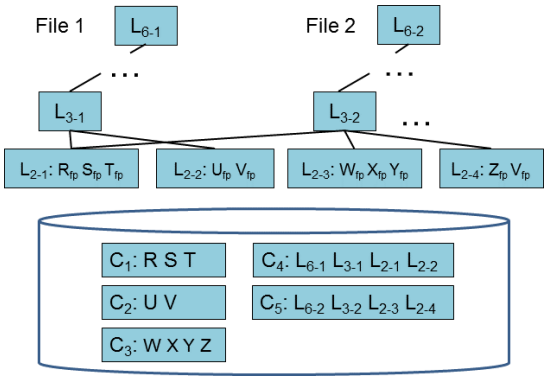


Figure 5: Enumeration example showing the L_p tree for two files and chunks stored in containers on disk.

3.3 Enumeration Example

We next present an example of logical and physical enumeration shown in Figure 5. The L_p trees for two files are shown as well as the location of chunks within containers (C_1-C_5) on disk. File 2 was created by copying File 1 and inserting new content after chunk T. The L_p trees are simplified for this example. Starting with logical enumeration, we begin with File 1. We first read container C_4 holding the root chunk for the file, $L_{6\text{-}1}$, which also reads $L_{3\text{-}1}$, $L_{2\text{-}1}$, and $L_{2\text{-}2}$ into memory. Accessing the L_1 chunks referenced from $L_{2\text{-}1}$ and $L_{2\text{-}2}$ involves reading containers C_1 and C_2. Logically enumerating File 2 follows similar steps; because $L_{3\text{-}2}$ refers to $L_{2\text{-}1}$, container C_1 is read a second time.

Physical enumeration reads all of the L_6 chunks before the L_5 chunks continuing down to the L_1 chunks. In this example, containers C_4 and C_5 are read to access the L_6 chunks. These containers are read again to access the L_3 and L_2 chunks (depending on whether the containers are still in the memory cache). Next the L_1 chunks are read, so containers C_1, C_2, and C_3 are read once to access their L_1 chunks.

3.4 Phase-optimized Physical GC (PGC$^+$)

Even though PGC reduces the time taken by LGC for enumeration in cases of high deduplication or file counts, it adds up to 2 extra analysis phases. PGC$^+$ improves upon PGC in several ways, especially by reducing the

GC phases. As discussed in §3.1.2, LGC and PGC often must run focused cleaning because there is not enough memory for a Bloom filter containing all the fingerprints in the system. On our systems, GC only has sufficient memory to store 25–40% of the total fingerprints in the Bloom filter while keeping a low enough rate of false positives. PGC$^+$ improves on PGC/LGC by making more efficient use of memory, virtually eliminating the need for focused cleaning, using these key ideas:

1. Replace the Live Vector of L_0 chunks with a PHV. This reduces the memory requirement from 6 bits per fingerprint to 2.8 bits per fingerprint (a 2.1× reduction).
2. Remove the Live Instance Vector and eliminate duplicates using a dynamic duplicate removal algorithm in the copy phase (see §3.4.2). This reduces memory requirements by 2×.

Between these changes, we get a 4.2× reduction in memory consumption, enough to use PGC$^+$ by default. However, the system detects whether there is enough memory to store all the fingerprints; if not, it falls back automatically to PGC with its multiple passes. We expect this to happen only in cases with exceptionally high (Lempel-Ziv) compression, resulting in many more chunks being stored. An analysis of customer systems found that only 0.02% of systems exceed the threshold for unique fingerprints, a value that varies across platforms from 10B at the low end to over 200B for the largest systems.[5]

3.4.1 PGC$^+$ Phases

1. Merge: Same as LGC and PGC.

1'. Analysis: Walk the index and create a PH L_P vector and a PH Live Vector. Build a per-container duplicate counter at the same time. This is a hash table indexed by container IDs which contains a count of duplicates across all the other container IDs. This counter is used for the duplicate count estimation algorithm in the select phase.

2. Filter: This phase is omitted for PGC$^+$ (see §3.4.2).

3. Enumeration: Same as PGC, but for every live L_0 chunk, add its fingerprint to the PHV.

4. Select: Iterate through the containers and use the live PHV and the duplicate counter to estimate the liveness of the container.

5. Copy: Copy live chunks forward by iterating containers backwards and dynamically remove duplicates. Backward iteration is needed to preserve the *latest* copy of a fingerprint.

[5]The maximum encountered on any system is 170B, which fits on that system without the need to revert to PGC.

3.4.2 PGC$^+$ Optimizations

Replacing the remaining Bloom filter with a PHV requires optimizations to compensate for extra overheads. Some, *e.g.* parallelization, could be applied to LGC as well.

Parallelization. One problem was an implementation requirement that entries in the on-disk index be returned in order. Originally processing the index was single-threaded, but by dividing the index into distinct partitions, these could be processed by threads in parallel. This parallelism compensates for the extra work of calculating PH functions for all the fingerprints in the system.

Memory lookups. PH functions are organized in an array, so the first step is to find the position in the array. The header information at this location refers to the actual location of the hash function in memory (second access). Accessing the hash function (third access) produces the offset of the bit to set/check. The fourth access is to the actual bit. In addition, initially these accesses were not NUMA-aware. Through a combination of restructuring the memory locations, adding prefetching, and adding NUMA-awareness, we improved PH latency to be comparable to the original Bloom filters.

Dynamic Duplicate Estimation. Just like with LGC and PGC, we estimate the liveness of each container in the PGC$^+$ Select phase to clean the mostly dead containers. LGC and PGC have a Pre-Filter phase before the Select phase, so when the <fingerprint `XOR` container_ID> lookup is done in the Live Instance Vector, we know the count of unique live chunks in that container. With PGC$^+$, there is no Pre-Filter before the Select phase. Thus the lookup in the Live Vector indicates whether chunks are live or dead but for live chunks does not distinguish among duplicates.

To estimate the unique liveness (which excludes live duplicates) of a container in PGC$^+$, we first build a duplicate counter per container (`ANALYSIS_DUP_CNTR`) in the Analysis phase. This counter tracks the number of chunks in a container with duplicates in other containers. Since the Analysis phase is before the Enumeration phase, this counter includes both live and dead duplicate chunks. Then in the Select phase, a Bloom filter is created to track the dead chunks in the container set. During the Select phase, we walk the container set in reverse, from the end to the beginning, to find the latest copy of any duplicate chunk. For every container, we read the container metadata to get the fingerprints and look up these fingerprints in the Live Vector. If the Live Vector indicates the chunk is dead, we insert the dead chunk into the Bloom filter. If the Dead Vector already includes the chunk, we increment a dead duplicate counter (`SELECT_DEAD_DUP_CNTR`) for that container.

Finally, the live duplicate count per container is:

$$LIVE_DUP_CNTR = ANALYSIS_DUP_CNTR - SELECT_DEAD_DUP_CNTR$$

Hence the container liveness is given by

$$CONTAINER_LIVENESS = LIVE_CHUNK_COUNT - LIVE_DUP_CNTR$$

In general, the number of dead chunks is much lower than live chunks (a $\frac{1}{10}$ ratio). Thus the memory needed for the Dead Vector is small. By keeping a Dead Vector to count the dead duplicates, we are able to estimate the correct liveness of containers in a manner similar to LGC and PGC.

Dynamic Duplicate Removal. In LGC and PGC, copy forward works from the lowest to highest container ID and copies live chunks into new containers. In PGC$^+$, to delete older duplicates and preserve the latest copy of the chunk, the copy forward happens in reverse order. Each time a live chunk is copied forward, subsequent chunks with the same fingerprint are treated as duplicates. To mark these duplicates as dead, we clear the corresponding bit in the PHV. The clearing of the liveness bit works because the PHV maps each fingerprint uniquely to the vector. While iterating backward, the next time we see the chunk whose bit is cleared in the PHV, we treat it as dead. Hence, unlike PGC where the Live Instance Vector is needed for removing duplicates, duplicate detection in PGC$^+$ happens in the copy phase. As discussed above, this uses half the memory and can represent 2× more fingerprints.

The copy phase only focuses on the set of containers selected in the select phase, which frees the maximum space in the allotted GC time. But in order to remove duplicates, the PGC$^+$ copy phase must flip the PHV bit for chunks belonging to containers that are not selected for copy forward. That way, older duplicates corresponding to those chunks are treated as dead and can be cleaned. For this, PGC$^+$ must read metadata sections of unselected containers too, and update the PHV. This involves extra container metadata reads, thus it slightly increases the overhead of the copy phase compared to PGC (see §5.2).

3.5 Online GC

It is unreasonable to take the system offline during GC, so while GC is running, our customers are also doing backups. Because our deduplication process checks against any chunks in the system, if a client writes a chunk that is a duplicate of an otherwise unreferenced (dead) chunk, it switches the unreferenced chunk to a referenced (live) state, known as a chunk resurrection. Another point is that open files may not yet be reachable from the root directory, so we specifically track open files and enumerate their L_p trees.

We added a process to inform GC of incoming chunks while GC runs, so chunks can be added to the Live (Instance) Vector. When resurrections take place during the Copy phase, there is a danger that a resurrected chunk may be in the process of being deleted. We see if the resurrection is in the container range currently being processed; if so, we write a (potentially) duplicate chunk for safety, but duplicates written during the Copy phase can be removed during the next GC run. If the resurrection takes place outside of the container range, then we add the <fingerprint, container_ID> to the Live Instance Vector, or the fingerprint to the Live Vector for PGC$^+$.

GC needs not only to support online ingest (adding new data to the system while GC is running), it must also support other tasks such as restores and replication. Thus we have added controls to limit the resource usage of GC, which are configurable on a per-system basis. We call this control *throttling*, but the manner in which GC is throttled has changed from LGC to PGC. Since PGC limits GC performance in ways LGC does not, direct comparisons between the two are problematic.

We take two tacts to address the differences in performance. The first is to consider head-to-head comparisons of LGC and PGC only in the extreme cases that PGC was intended to address. If PGC gives better performance than LGC even when PGC is potentially throttling itself more, we can see that PGC is strictly better than LGC (§5.1). The second is to run in-lab experiments with throttling manually disabled for all GC algorithms (§5.2-5.3).

4 Methodology

This section describes our experimental methodology: we used a combination of telemetry from deployed systems and controlled experiments.

4.1 Deployed Systems

Like many vendors [2, 10], our products can send optional reports on system behavior [24]. We use these reports to extract the following information from each report: serial number, model, system version, date, logical and physical capacity in use, file counts, and GC timings.

We exclude any reports showing <1TB physical capacity in use, <1 hour or >2 weeks of GC execution, or other outliers.[6] This leaves us with one or more reports per system, which may reflect LGC or PGC taking place.

[6]The outliers with low usage or unusually fast GC are uninteresting (for example, a system that is not in active use and has no garbage to collect). Any rare cases of unusually high times are attributed to bugs that were subsequently fixed.

Most of our evaluation focuses on a large set of deployed systems running LGC. We use a second set of deployed systems running PGC (PGC$^+$ not yet being available) for head-to-head comparisons of the extreme cases (high deduplication rates or file counts). We looked at systems that were upgraded from LGC to PGC and did not change capacity, file counts, or deduplication ratio by more than 5%. We found that for the more "typical" systems, there was moderate (and occasionally significant) degradation when running PGC instead of LGC, but it is not possible to attribute that effect to the PGC algorithm versus throttling effects or one-time overheads occurring in the process of the upgrade. We exclude throttling in the in-lab experiments reported later. The timings were collected from Jan-2014 to Sep-2016; both sets (LGC and PGC) contain thousands of systems.

There are various ways to evaluate GC performance, and in particular, evaluate the change in the context of a particular system. Our usual approach to compare similar systems is to sort them by some metric, such as physical capacity, and divide them into buckets. Note that these buckets are marked as $\leq$ some value, but they reflect only those points that are greater than the next lower bucket (*i.e.*, they are not cumulative across all lower buckets). In addition we generally find that showing the buckets on a log scale emphasizes the outliers. Since PGC$^+$ is only recently available to customers, we cannot use telemetry to evaluate systems using PGC$^+$.

4.2 Controlled Experiments

To compare LGC, PGC, and PGC$^+$ in a controlled manner, we perform a number of GC experiments in a lab environment. To ensure that the data is the same across experiments, we obtain or generate the test dataset, then reset to the state of the test dataset anytime the system is modified. For most experiments, we use a load generator, the same one used to evaluate content sanitization in DDFS [5]; it is similar to the work of Tarasov, *et al.* [23]. The tool creates files of a specified size, using a combination of random and predictable content to get desired levels of deduplication and compression. Each new file contains significant content from the previous generation of the same "lineage," while some content is deleted as new content is added. There should not be deduplication across independent lineages, which can be written in parallel to achieve a desired throughput.

For some of the experiments, we went through some extra steps to create what we call our *canonical dataset*. As we wrote new data, we ran GC several times to age the system and degrade locality: for example, ingesting to 5% of capacity and using GC to reset to 3% of capacity before ingesting to 7% and running GC again. (We note that the number of unfiltered duplicates removed in this process is higher than the number of duplicates that typically appear in the field (10% or less.)) Ingestion was halted when we reached 10% of the physical capacity of the largest system, the DD990. The canonical dataset is 1.1 PiB of logical data deduplicated and compressed to 30.1 TiB of physical data (36.6× TC). We used this dataset to compare different appliances and also as the starting point for evaluating high deduplication rates.

The canonical dataset is about 25% of the capacity of the DD860, which we used for other experiments. To evaluate the impact of physical capacity in use, we wanted to start with a lower capacity, so we ingested 20% of physical capacity at a time. For this experiment we did not run destructive GC (physically reorganizing data and removing extra duplicates) during ingestion. The TC for that experiment varied from 17.9×–18.3×.

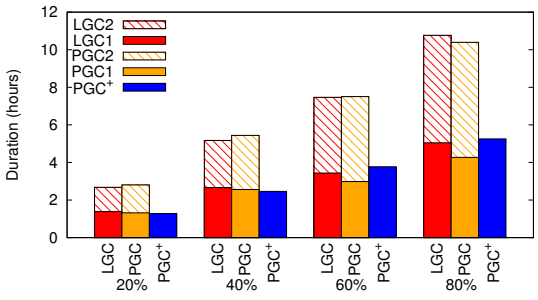
To start with the same predictable content, we use *collection replication* [7] to make a total copy of the data on another appliance. To focus on the *mark* part of GC, we can identify what data should be reclaimed but terminate GC before initiating *sweep* and modifying the file system. Multiple *mark* runs (of any GC type) can be measured without modifying the state of the system. If we include the copy phase, i.e. *sweep*, we replicate back from the saved copy to restore the system to the original state. Each 4MB container, as well as the index, will be the same, though containers may be stored in different physical locations on disk. We use replication to evaluate high file counts, copying from a quality assurance system that has been seeded with 900M files.

Since we find that many GC runs require two *mark* phases, it is important to evaluate runs with both one *mark* pass and two. We do this by collecting timings of both sets and indicating them separately in the results via stacked bar graphs. LGC-1 and PGC-1 are the lower (solid) bars; these may be compared directly with the single PGC$^+$ bar, or the higher shaded bars labeled LGC-2 and PGC-2 may be considered. In general, less full systems need one pass and more full systems need two.

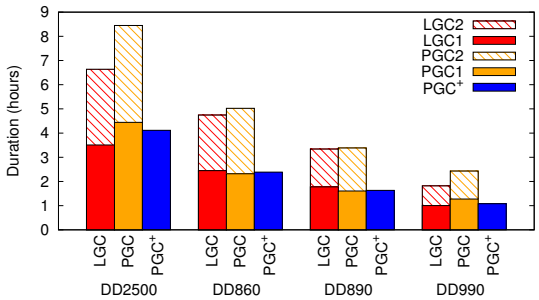
4.2.1 Test Environment

We use a variety of backup appliances for our tests. We report the time taken for the *mark* phase for our canonical dataset on four separate systems. We then use a specific system to evaluate some aspect of the workload, such as *copy* phase variability, capacity in use, high file counts, or variation in duplication ratios.

Table 1 shows specifications for the systems we used in our tests. It shows cores, memory, physical capacity available, and the amount of that capacity used by the canonical dataset, ranging from 10–25%. After comparing all four systems on one dataset, we use the DD860 for extensive comparisons on different workloads.



(a) GC duration by physical space



(b) GC duration by platform

Figure 6: (a) GC duration is a linear function of physical space, but PGC$^+$ is consistently faster than the other algorithms. PGC is consistently slightly slower than LGC. (b) GC duration varies across platforms. PGC$^+$ consistently outperforms LGC and PGC on all platforms. PGC is consistently slower than LGC when two phases are required, but the difference is variable for just one phase, with PGC slightly faster in two cases.

Systems	DD2500	DD860	DD890	DD990
CPU (cores×GHz)	8×2.20	16×2.53	24×2.80	40×2.40
Mem (GB)	64	70	94	256
Phy. Capacity(TiB)	122	126	167	319
Canonical Dataset util.(%)	25	25	19	10

Table 1: System Specifications.

5 Evaluation

In this section we provide a detailed comparison of LGC, PGC, and PGC$^+$. §5.1 provides a high-level comparison across a set of deployed systems with anomalous enumeration performance (thus these omit PGC$^+$). §5.2 presents lab-based experiments comparing all three GC variants on basic datasets (nominal deduplication and file counts), investigating the penalty from PH analysis when moving from LGC to PGC and the compensating benefit from eliminating multi-round *marking* before the sweep phase (PGC$^+$). §5.3 then focuses on specific challenges for LGC, high deduplication ratios and large file counts.

5.1 Deployed Systems

We start by evaluating the change from LGC to PGC in deployed systems. When comparing an upgraded system, we can consider the time it took to run LGC before the upgrade and the time it took to run PGC after the upgrade. (Recall that we only consider upgraded systems that are within 5% of each other in various metrics in both cases).

Across all systems, we find that 75% of systems suffer some form of performance degradation when moving from LGC to PGC. As stated before, this is due to various factors including the change to throttling. As customers are able to upgrade to PGC$^+$, we focus on emphasizing the benefits of PGC over LGC in the extreme scenarios, then discuss lab-based comparisons in the remainder of this section. We find that PGC improves enumeration performance over LGC by as much as 20× in the case of the greatest improvement for high deduplication, and by 7× for the greatest improvement for high file counts.

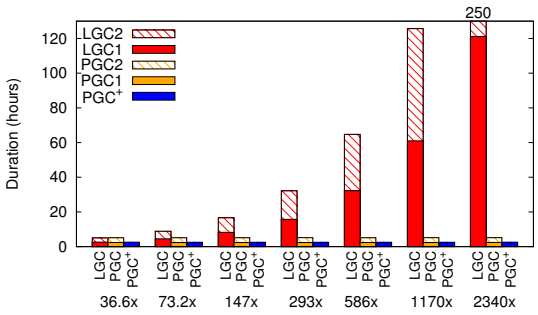
5.2 Standard Workloads

To start, we explain why we focus on *mark* performance rather than *sweep* (Copy). The copy times are proportional to physical space used, and they are not problematic (the maximum time for any system is always "reasonable" because resources such as cores and memory scale with disk capacity). Copy for PGC$^+$ is slightly slower than for PGC or LGC (which use an identical algorithm) due to the PHV overhead, but we measured this to be only a 2–3% increase. Running Copy on the same workload three times per algorithm, we found a tiny variation among multiple runs (a standard deviation less than 0.5%). Thus, for the remainder of the paper, we focus on *mark* performance, though we give some examples of the PGC$^+$ *sweep* times for comparison.

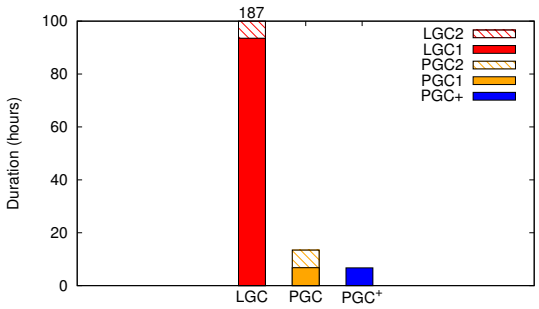
Within the *mark* phase, for the canonical dataset, the analysis portion of PGC$^+$ is 10–20% of total elapsed time. It is about 20% of elapsed time on the DD990 and about 10% on the other platforms.

We evaluated performance in a controlled environment using the load generator described in §4.2. We ingested data to different fractions[7] of the overall capacity of a specific system (DD860) and measured the time taken for the *mark* phases of LGC, PGC and PGC$^+$ to complete. As we can see in Figure 6(a), there is roughly a linear relationship between the GC duration and the physical capacity in use.

[7]The amount ingested is somewhat approximate, but the trend is clear.



(a) GC duration by total compression (log scale)



(b) GC duration with high file count

Figure 7: (a) As the deduplication ratio increases, LGC duration increases substantially while PGC and PGC$^+$ stay constant. (b) While LGC performance is poor when the number of files is unusually high, PGC and PGC$^+$ are substantially faster.

As mentioned in §3.1.2, the need for two phases arises from the number of chunks stored in the system; this is usually due to higher capacity in use, but higher compression can result in more chunks being stored in the same physical space. PGC is generally slightly slower than LGC at lower utilizations and slightly faster at higher ones; PGC$^+$ consistently completes at least 2× faster than either LGC or PGC when two phases are required. For low space utilization PGC$^+$ is slightly faster than LGC and PGC when a single phase is required; at the highest level tested, it would be about 20% slower than PGC if only one phase were required; however, two phases are needed for this dataset, so PGC$^+$ is uniformly faster.

Figure 6(b) shows the results of *mark* for different GC algorithms running with the canonical dataset on the four platforms. In general, higher-end platforms have better GC performance due to more resources (e.g., CPU, memory, storage bandwidth). There is some variation in the relative performance of the algorithms across platforms, but in general, PGC$^+$ is close to LGC and PGC for a single *mark* phase and much faster for two. DD2500, the lowest-end platform, shows the greatest degradation (17%) moving from LGC to PGC$^+$ when just one phase is needed. To give an idea of the relative performance of the *copy* phase, across these platforms the time for *copy* is $2.3-2.6\times$ that of the PGC$^+$ *mark* times.

5.3 Problematic Workloads

The move to PGC was spurred by two types of problematic workloads, high deduplication ratios and high file counts. These are depicted in Figure 7 using the DD860. In Figure 7(a), the *fastcopy* [7] system command is used to repeatedly double the TC from a moderate 36.6× up to an extreme 2340×. The enumeration time for LGC increases with deduplication ratio, while the times for PGC and PGC$^+$ are a function of the physical space, which is nearly constant. In the worst case, PGC$^+$ is nearly two orders of magnitude faster than LGC, and even if LGC needs only one *mark* phase, LGC is 47× slower. Note that the values for a 36.6× TC are the same as the DD860 in Figure 6(b); refer there for detail.

Figure 7(b) shows the GC *mark* times for a system with over 900M files, for different GC algorithms.[8] The system has a TC of only 2.3×, meaning there is almost no deduplication. PGC and PGC$^+$ take substantially less time than LGC because LGC has to traverse the L_p tree multiple times based on the number of logical files. LGC also induces substantial random I/Os. In contrast to LGC, PGC/PGC$^+$ can traverse the L_p tree in a limited sequence of passes and enumerate files with sequential I/Os. Compared with LGC, PGC$^+$ is 13.8×–27.8× faster for one or two *mark* phases, respectively. PGC$^+$ is virtually identical to a single phase of PGC and twice as fast than the two-phase run.

6 Related Work

While many deduplicated storage papers mention their technique to remove unreferenced chunks, only a few present a detailed implementation. We presented an overview of the differences between reference counts and mark-and-sweep algorithms in §2 and now provide more discussion of related work.

Fu et al. [8] maintained reference counts for containers as part of a technique to reduce fragmentation for faster restores. A limitation of their work is that their garbage collection technique only supports first-in-first-out deletion, while our system supports an arbitrary deletion pattern. Strzelczak et al. [22] describe

[8] This is data replicated to a DD860 from an internal Quality Assurance system that was seeded over the period of many months; it takes too long to write the individual files for us to use other high file counts, so we demonstrate the savings at the extreme.

a distributed reference count algorithm for HYDRAstor. They use an epoch-based approach to allow cleaning to take place while the system allows new writes. Their indexing mechanism is not described, but there is an indication that they have sufficient memory to track reference counts. Our system has smaller chunks (*e.g.* ~8KB) and uses compact data structures to reduce memory requirements. Simha et al. [21] describe a technique for efficiently handling incremental changes, though they do not generate full backups from incremental writes as our system supports, so they have a smaller name space to enumerate than our system. Their reference count technique leverages known expiration times for snapshots, while we support any deletion order. Without expiration times in advance, it would be necessary to walk an entire tree to update reference counts in the case of a snapshot or *fastcopy* [7], and deletions similarly require mass updating. Grouped Mark and Sweep [9] marks referenced containers. Their file representation has direct references to physical blocks, so it is unclear how they can copy forward live chunks to consolidate partially dead containers. In contrast, our file representation uses fingerprints and we use a fingerprint-to-container index to support container cleaning. One might use SSDs to store the reference counts, but then one must address write amplification from these updates. In future years other forms of memory, nonvolatile or otherwise, may be cost-effective for systems of this scale. Finally, reference counts are also difficult to maintain correctly in the presence of complex error conditions.

The most similar work to our own is the sanitization technique presented by Botelho et al. [5]. Like us, they used perfect hashes to compactly represent live references in deduplicating storage. A key difference is that they focused on sanitization rather than garbage collection. Sanitization is the process of securely deleting data to prevent the leakage of confidential information, so sanitization has the requirement of removing all unreferenced chunks. This means that they created a PH function and vector over *all* chunks (L_0-L_6) so that any unreferenced chunk could be removed; they still performed logical rather than physical enumeration of the file system. Our physical enumeration technique could possibly replace logical enumeration in their algorithm, and our other optimizations to PH are also applicable.

Techniques that defer cleaning and focus on the most efficient areas to clean tend to have some basis in early work on cleaning log structured storage [15, 20]. Other LFS optimizations such as hole-plugging [26] do not work well at the granularity of chunks that are packed and compressed in large units.

While our work focuses on garbage collection, the differences between the logical and physical view of a storage system were noted by Hutchinson, et al. [11]. In that work, the authors found that backing up a system by accessing its blocks at a physical level provided better performance than accessing blocks one file at a time. Thus the optimization of PGC to iterate through the physical locations of system metadata is similar to their optimization when copying data in the first place, but deduplication magnifies the distinction.

7 Conclusion

The shift in workloads has required a new approach to garbage collection in a deduplicating storage system. Rather than a depth-first mark-and-sweep GC algorithm, tracking live data from the perspective of individual files, we have moved to a breadth-first approach that takes advantage of the physical layout of the data.

PGC turns large numbers of random I/Os into a set of sequential scans of the entire storage system. It works well when deduplication is high (100–1000× rather than 10–20×), and it works well when the storage system is used for hundreds of millions of relatively small files rather than thousands of large tar-like backup files.

Because of the other overheads of PGC, including the analysis necessary to use the space-efficient and accurate perfect hash functions [3, 14], the original PGC approach is not uniformly faster than the LGC approach it replaced. The improved PGC algorithm, referred to as PGC$^+$, uses PH in an additional way to reduce memory requirements enough to avoid two *mark* sequences during GC. We have demonstrated that it is comparable to the original LGC on traditional workloads whose fingerprints fit into memory (requiring a single *mark* phase), significantly faster when two passes are required, and orders of magnitude better on problematic workloads.

Acknowledgments

We thank the many Data Domain engineers who have contributed to its GC system over the years, especially Ed Lee, Guilherme Menezes, Srikant Varadan, and Ying Xie. We appreciate the feedback of our shepherd André Brinkmann, the anonymous reviewers, and several people within Dell EMC: Bhimsen Banjois, Orit Levin-Michael, Lucy Luo, Stephen Manley, Darren Sawyer, Stephen Smaldone, Grant Wallace, and Ian Wigmore. We very much appreciate the assistance of Mark Chamness and Rachel Traylor with data analysis; Lang Mach, Murali Mallina, and Mita Mehanti for the QA dataset access; and Duc Dang, David Lin, and Tuan Nguyen for debugging and performance tuning.

References

[1] ALLU, Y., DOUGLIS, F., KAMAT, M., SHILANE, P., PATTERSON, H., AND ZHU, B. Evolution of the Data Domain file system. *IEEE Computer*. To appear.

[2] AMVROSIADIS, G., AND BHADKAMKAR, M. Identifying trends in enterprise data protection systems. In *USENIX Annual Technical Conference (ATC'15)* (July 2015), pp. 151–164.

[3] BELAZZOUGUI, D., BOTELHO, F. C., AND DIETZFELBINGER, M. Hash, displace, and compress. In *Algorithms-ESA 2009*. Springer, 2009, pp. 682–693.

[4] BOTELHO, F. C., PAGH, R., AND ZIVIANI, N. Practical perfect hashing in nearly optimal space. *Information Systems* (June 2012). http://dx.doi.org/10.1016/j.is.2012.06.002.

[5] BOTELHO, F. C., SHILANE, P., GARG, N., AND HSU, W. Memory efficient sanitization of a deduplicated storage system. In *USENIX Conference on File and Storage Technologies (FAST'13)* (Feb 2013).

[6] CHAMNESS, M. Capacity forecasting in a backup storage environment. In *25th Large Installation System Administration Conference (LISA'11)* (2011).

[7] EMC CORP. *EMC® Data Domain® Operating System, Version 5.7, Administration Guide*, Mar. 2016. `https://support.emc.com/docu61787_Data-Domain-Operating-System-5.7.1-Administration-Guide.pdf?language=en_US`.

[8] FU, M., FENG, D., HUA, Y., HE, X., CHEN, Z., XIA, W., HUANG, F., AND LIU, Q. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *USENIX Annual Technical Conference (ATC'14)* (2014).

[9] GUO, F., AND EFSTATHOPOULOS, P. Building a high-performance deduplication system. In *USENIX Annual Technical Conference (ATC'11)* (2011).

[10] HAO, M., SOUNDARARAJAN, G., KENCHAMMANA-HOSEKOTE, D., CHIEN, A. A., AND GUNAWI, H. S. The tail at store: a revelation from millions of hours of disk and ssd deployments. In *14th USENIX Conference on File and Storage Technologies (FAST'16)* (2016), pp. 263–276.

[11] HUTCHINSON, N. C., MANLEY, S., FEDERWISCH, M., HARRIS, G., HITZ, D., KLEIMAN, S., AND O'MALLEY, S. Logical vs. physical file system backup. In *Third Symposium on Operating Systems Design and Implementation (OSDI'99)* (1999).

[12] KACZMARCZYK, M., BARCZYNSKI, M., KILIAN, W., AND DUBNICKI, C. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference* (2012), ACM, p. 15.

[13] LILLIBRIDGE, M., ESHGHI, K., AND BHAGWAT, D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *USENIX Conference on File and Storage Technologies (FAST'13)* (Feb 2013).

[14] MAJEWSKI, B. S., WORMALD, N. C., HAVAS, G., AND CZECH, Z. J. A family of perfect hashing methods. *The Computer Journal 39*, 6 (1996), 547–554.

[15] MATTHEWS, J. N., ROSELLI, D., COSTELLO, A. M., WANG, R. Y., AND ANDERSON, T. E. Improving the performance of log-structured file systems with adaptive methods. In *16th ACM Symposium on Operating Systems Principles* (1997), SOSP'97.

[16] MERKLE, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology CRYPTO'87* (1988), Springer, pp. 369–378.

[17] PAULO, J., AND PEREIRA, J. A survey and classification of storage deduplication systems. *ACM Computing Surveys 47*, 1 (2014).

[18] PUTZE, F., SANDERS, P., AND SINGLER, J. Cache-, hash-and space-efficient bloom filters. In *Experimental Algorithms, 6th International Workshop, WEA 2007, Lecture Notes in Computer Science 4525*. Springer, 2007, pp. 108–121.

[19] ROSELLI, D. S., LORCH, J. R., ANDERSON, T. E., ET AL. A comparison of file system workloads. In *USENIX Annual Technical Conference, general track (ATC'00)* (2000), pp. 41–54.

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

[21] SIMHA, D. N., LU, M., AND CHIUEH, T.-C. A scalable deduplication and garbage collection engine for incremental backup. In *6th International Systems and Storage Conference (SYSTOR'13)* (2013).

[22] STRZELCZAK, P., ADAMCZYK, E., HERMAN-IZYCKA, U., SAKOWICZ, J., SLUSARCZYK, L., WRONA, J., AND DUBNICKI, C. Concurrent deletion in a distributed content-addressable storage system with global deduplication. In *USENIX Conference on File and Storage Technologies (FAST'13)* (2013).

[23] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating realistic datasets for deduplication analysis. In *USENIX Annual Technical Conference (ATC'12)* (Jun 2012).

[24] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *USENIX Conference on File and Storage Technologies (FAST'12)* (2012).

[25] WEI, J., JIANG, H., ZHOU, K., AND FENG, D. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *Massive Storage Systems and Technologies (MSST'10)* (2010), IEEE, pp. 1–14.

[26] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS) 14*, 1 (1996), 108–136.

[27] WILSON, P. R. Uniprocessor garbage collection techniques. In *Memory Management*. Springer, 1992, pp. 1–42.

[28] XIA, W., JIANG, H., FENG, D., DOUGLIS, F., SHILANE, P., HUA, Y., FU, M., ZHANG, Y., AND ZHOU, Y. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE 104*, 9 (Sept. 2016), 1681–1710.

[29] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *USENIX Conference on File and Storage Technologies (FAST'08)* (Feb 2008).

File Systems Fated for Senescence? Nonsense, Says Science!

Alex Conway*, Ainesh Bakshi*, Yizheng Jiao‡, Yang Zhan‡,
Michael A. Bender†, William Jannen†, Rob Johnson†, Bradley C. Kuszmaul§,
Donald E. Porter‡, Jun Yuan¶, and Martin Farach-Colton*

*Rutgers University, †Stony Brook University,
‡The University of North Carolina at Chapel Hill,
§Oracle Corporation and Massachusetts Institute of Technology,
¶Farmingdale State College of SUNY

Abstract

File systems must allocate space for files without knowing what will be added or removed in the future. Over the life of a file system, this may cause suboptimal file placement decisions which eventually lead to slower performance, or ***aging***. Traditional file systems employ heuristics, such as collocating related files and data blocks, to avoid aging, and many file system implementors treat aging as a solved problem.

However, this paper describes realistic as well as synthetic workloads that can cause these heuristics to fail, inducing large performance declines due to aging. For example, on ext4 and ZFS, a few hundred git pull operations can reduce read performance by a factor of 2; performing a thousand pulls can reduce performance by up to a factor of 30. We further present microbenchmarks demonstrating that common placement strategies are extremely sensitive to file-creation order; varying the creation order of a few thousand small files in a real-world directory structure can slow down reads by $15-175\times$, depending on the file system.

We argue that these slowdowns are caused by poor layout. We demonstrate a correlation between read performance of a directory scan and the locality within a file system's access patterns, using a ***dynamic layout score***.

In short, many file systems are exquisitely prone to read aging for a variety of write workloads. We show, however, that aging is not inevitable. BetrFS, a file system based on write-optimized dictionaries, exhibits almost no aging in our experiments. BetrFS typically outperforms the other file systems in our benchmarks; aged BetrFS even outperforms the unaged versions of these file systems, excepting Btrfs. We present a framework for understanding and predicting aging, and identify the key features of BetrFS that avoid aging.

1 Introduction

File systems tend to become fragmented, or ***age***, as files are created, deleted, moved, appended to, and truncated [18, 23].

Fragmentation occurs when logically contiguous file blocks—either blocks from a large file or small files from the same directory—become scattered on disk. Reading these files requires additional seeks, and on hard drives, a few seeks can have an outsized effect on performance. For example, if a file system places a 100 MiB file in 200 disjoint pieces (i.e., 200 seeks) on a disk with 100 MiB/s bandwidth and 5 ms seek time, reading the data will take twice as long as reading it in an ideal layout. Even on SSDs, which do not perform mechanical seeks, a decline in logical block locality can harm performance [19].

The state of the art in mitigating aging applies best-effort heuristics at allocation time to avoid fragmentation. For example, file systems attempt to place related files close together on disk, while also leaving empty space for future files [7,17,18,25]. Some file systems (including ext4, XFS, Btrfs, and F2FS among those tested in this paper) also include defragmentation tools that attempt to reorganize files and file blocks into contiguous regions to counteract aging.

Over the past two decades, there have been differing opinions about the significance of aging. The seminal work of Smith and Seltzer [23] showed that file systems age under realistic workloads, and this aging affects performance. On the other hand, there is a widely held view in the developer community that aging is a solved problem in production file systems. For example, the Linux System Administrator's Guide [26] says:

> Modern Linux file systems keep fragmentation at a minimum by keeping all blocks in a file close together, even if they can't be stored in consecutive sectors. Some file systems, like ext3, effectively allocate the free block that is nearest to other blocks in a file. Therefore it is not necessary to worry about fragmentation in a Linux system.

There have also been changes in storage technology

and file system design that could substantially affect aging. For example, a back-of-the-envelope analysis suggests that aging should get worse as rotating disks get bigger, as seek times have been relatively stable, but bandwidth grows (approximately) as the square root of the capacity. Consider the same level of fragmentation as the above example, but on a new, faster disk with 600MiB/s bandwidth but still a 5ms seek time. Then the 200 seeks would introduce four-fold slowdown rather than a two-fold slowdown. Thus, we expect fragmentation to become an increasingly significant problem as the gap between random I/O and sequential I/O grows.

As for SSDs, there is a widespread belief that fragmentation is not an issue. For example, PCWorld measured the performance gains from defragmenting an NTFS file system on SSDs [1], and concluded that, "From my limited tests, I'm firmly convinced that the tiny difference that even the best SSD defragger makes is not worth reducing the life span of your SSD."

In this paper, we revisit the issue of file system aging in light of changes in storage hardware, file system design, and data-structure theory. We make several contributions: (1) We give a simple, fast, and portable method for aging file systems. (2) We show that fragmentation over time (i.e., aging) is a first-order performance concern, and that this is true even on modern hardware, such as SSDs, and on modern file systems. (3) Furthermore, we show that aging is not inevitable. We present several techniques for avoiding aging. We show that BetrFS [10–12, 27], a research prototype that includes several of these design techniques, is much more resistant to aging than the other file systems we tested. In fact, BetrFS essentially did not age in our experiments, establishing that aging is a solvable problem.

Results. We use realistic application workloads to age five widely-used file systems—Btrfs [21], ext4 [7,17,25], F2FS [15], XFS [24] and ZFS [6]—as well as the BetrFS research file system. One workload ages the file system by performing successive git checkouts of the Linux kernel source, emulating the aging that a developer might experience on her workstation. A second workload ages the file system by running a mail-server benchmark, emulating aging over continued use of the server.

We evaluate the impact of aging as follows. We periodically stop the aging workload and measure the overall read throughput of the file system—greater fragmentation will result in slower read throughput. To isolate the impact of aging, as opposed to performance degradation due to changes in, say, the distribution of file sizes, we then copy the file system onto a fresh partition, essentially producing a defragmented or "unaged" version of the file system, and perform the same measurement. We treat the differences in read throughput between the aged and unaged copies as the result of aging.

We find that:

- All the production file systems age on both rotating disks and SSDs. For example, under our git workload, we observe over 50× slowdowns on hard disks and 2–5× slowdowns on SSDs. Similarly, our mail-server slows down 4–30× on HDDs due to aging.
- Aging can happen quickly. For example, ext4 shows over a 2× slowdown after 100 git pulls; Btrfs and ZFS slow down similarly after 300 pulls.
- BetrFS exhibits essentially no aging. Other than Btrfs, BetrFS's aged performance is better than the other file systems' unaged performance on almost all benchmarks. For instance, on our mail-server workload, unaged ext4 is 6× slower than aged BetrFS.
- The costs of aging can be staggering in concrete terms. For example, at the end of our git workload on an HDD, all four production file systems took over 8 minutes to grep through 1GiB of data. Two of the four took over 25 minutes. BetrFS took 10 seconds.

We performed several microbenchmarks to dive into the causes of aging and found that performance in the production file systems was sensitive to numerous factors:

- If only 10% of files are created out of order relative to the directory structure (and therefore relative to a depth-first search of the directory tree), Btrfs, ext4, F2FS, XFS and ZFS cannot achieve a throughput of 5 MiB/s. If the files are copied completely out of order, then of these only XFS significantly exceeds 1 MiB/s. This need not be the case; BetrFS maintains a throughput of roughly 50 MiB/s.
- If an application writes to a file in small chunks, then the file's blocks can end up scattered on disk, harming performance when reading the file back. For example, in a benchmark that appends 4 KiB chunks to 10 files in a round-robin fashion on a hard drive, Btrfs and F2FS realize 10 times lower read throughput than if each file is written completely, one at a time. ext4 and XFS are more stable but eventually age by a factor of 2. ZFS has relatively low throughout but did not age. BetrFS throughput is stable, at two thirds of full disk bandwidth throughout the test.

2 Related Work

Prior work on file system aging falls into three categories: techniques for artificially inducing aging, for measuring aging, and for mitigating aging.

2.1 Creating Aged File Systems

The seminal work of Smith and Seltzer [23] created a methodology for simulating and measuring aging on a file system—leading to more representative benchmark results than running on a new, empty file system. The study is based on data collected from daily snapshots of

Feature	Btrfs	ext4	F2FS	XFS	ZFS	BetrFS
Grouped allocation within directories	✓	✓		✓	✓	✓
Extents	✓	✓		✓	✓	
Delayed allocation	✓	✓	✓	✓	✓	✓
Packing small files and metadata	✓ (by OID)					✓
Default Node Size	16 K	4 K	4 K	4 K	8 K	2–4 M
Maximum Node Size	64 K	64 K	4 K	64 K	128 K	2–4 M
Rewriting for locality						✓
Batching writes to reduce amplification			✓			✓

Table 1: Principal anti-aging features of the file systems measured in this paper. The top portion of the table are commonly-deployed features, and the bottom portion indicates features our model (§3) indicates are essential; an ideal node size should match the natural transfer size, which is roughly 4 MiB for modern HDDs and SSDs. OID in Btrfs is an object identifier, roughly corresponding to an inode number, which is assigned at creation time.

over fifty real file systems from five servers over durations ranging from one to three years. An overarching goal of Smith and Seltzer's work was to evaluate file systems with representative levels of aging.

Other tools have been subsequently developed for synthetically aging a file system. In order to measure NFS performance, TBBT [28] was designed to synthetically age a disk to create a initial state for NFS trace replay.

The Impressions framework [2] was designed so that users can synthetically age a file system by setting a small number of parameters, such as the organization of the directory hierarchy. Impressions also lets users specify a target layout score for the resulting image.

Both TBBT and Impressions create file systems with a specific level of fragmentation, whereas our study identifies realistic workloads that induce fragmentation.

2.2 Measuring Aged File Systems

Smith and Seltzer also introduced a ***layout score*** for studying aging, which was used by subsequent studies [2, 4]. Their layout score is the fraction of file blocks that are placed in consecutive physical locations on the disk. We introduce a variation of this measure, the ***dynamic layout score*** in Section 3.3.

The ***degree of fragmentation*** (***DoF***) is used in the study of fragmentation in mobile devices [13]. DoF is the ratio of the actual number of extents, or ranges of contiguous physical blocks, to the ideal number of extents. Both the layout score and DoF measure how one file is fragmented.

Several studies have reported file system statistics such as number of files, distributions of file sizes and types, and organization of file system namespaces [3, 9, 22]. These statistics can inform parameter choices in aging frameworks like TBBT and Impressions [2, 28].

2.3 Existing Strategies to Mitigate Aging

When files are created or extended, blocks must be allocated to store the new data. Especially when data is rarely or never relocated, as in an update-in-place file system like ext4, initial block allocation decisions determine performance over the life of the file system. Here we outline a few of the strategies use in modern file systems to address aging, primarily at allocation-time (also in the top of Table 1).

Cylinder or Block Groups. FFS [18] introduced the idea of ***cylinder groups***, which later evolved into block groups or allocation groups (XFS). Each group maintains information about its inodes and a bitmap of blocks. A new directory is placed in the cylinder group that contains more than the average number of free inodes, while inodes and data blocks of files in one directory are placed in the same cylinder group when possible.

ZFS [6] is designed to pool storage across multiple devies [6]. ZFS selects from one of a few hundred ***metaslabs*** on a device, based on a weighted calculation of several factors including minimizing seek distances. The metaslab with the highest weight is chosen.

In the case of F2FS [15], a log-structured file system, the disk is divided into segments—the granularity at which the log is garbage collected, or cleaned. The primary locality-related optimization in F2FS is that writes are grouped to improve locality, and dirty segments are filled before finding another segment to write to. In other words, writes with temporal locality are more likely to be placed with physical locality.

Groups are a best-effort approach to directory locality: space is reserved for co-locating files in the same directory, but when space is exhausted, files in the same directory can be scattered across the disk. Similarly, if a file is renamed, it is not physically moved to a new group.

Extents. All of the file systems we measure, except F2FS and BetrFS, allocate space using ***extents***, or runs of physically contiguous blocks. In ext4 [7, 17, 25], for example, an extent can be up to 128 MiB. Extents reduce bookkeeping overheads (storing a range versus an exhaustive list of blocks). Heuristics to select larger extents can improve locality of large files. For instance, ZFS selects from available extents in a metaslab using a first-fit policy.

Delayed Allocation. Most modern file systems, including ext4, XFS, Btrfs, and ZFS, implement delayed allocation, where logical blocks are not allocated until buffers are written to disk. By delaying allocation when a file is growing, the file system can allocate a larger extent for data appended to the same file. However, allocations can only be delayed so long without violating durability and/or consistency requirements; a typical file system

ensures data is dirty no longer than a few seconds. Thus, delaying an allocation only improves locality inasmuch as adjacent data is also written on the same timescale; delayed allocation alone cannot prevent fragmentation when data is added or removed over larger timescales.

Application developers may also request a persistent preallocation of contiguous blocks using fallocate. To take full advantage of this interface, developers must know each file's size in advance. Furthermore, fallocate can only help intrafile fragmentation; there is currently not an analogous interface to ensure directory locality.

Packing small files and metadata. For directories with many small files, an important optimization can be to pack the file contents, and potentially metadata, into a small number of blocks or extents. Btrfs [21] stores metadata of files and directories in copy-on-write B-trees. Small files are broken into one or more fragments, which are packed inside the B-trees. For small files, the fragments are indexed by object identifier (comparable to inode number); the locality of a directory with multiple small files depends upon the proximity of the object identifiers.

BetrFS stores metadata and data as key-value pairs in two B$^\varepsilon$-trees. Nodes in a B$^\varepsilon$-tree are large (2–4 MiB), amortizing seek costs. Key/value pairs are packed within a node by sort-order, and nodes are periodically rewritten, copy-on-write, as changes are applied in batches.

BetrFS also divides the namespace of the file system into ***zones*** of a desired size (512 KiB by default), in order to maintain locality within a directory as well as implement efficient renames. Each zone root is either a single, large file, or a subdirectory of small files. The key for a file or directory is its relative path to its zone root. The key/value pairs in a zone are contiguous, thereby maintaining locality.

3 A Framework for Aging

3.1 Natural Transfer Size

Our model of aging is based on the observation that the bandwidth of many types of hardware is maximized when I/Os are large; that is, sequential I/Os are faster than random I/Os. We abstract away from the particulars of the storage hardware by defining the ***natural transfer size*** (NTS) to be the amount of sequential data that must be transferred per I/O in order to obtain some fixed fraction of maximum throughput, say 50% or 90%. Reads that involve more than the NTS of a device will run near bandwidth.

From Figure 1, which plots SSD and HDD bandwidth as a function of read size, we conclude that a reasonable NTS for both the SSDs and HDDs we measured is 4MiB.

The cause of the gap between sequential- and random-I/O speeds differs for different hardware. For HDDs, seek times offer a simple explanation. For SSDs, this gap is hard to explain conclusively without vendor support, but common theories include: sequential accesses are easier to stripe across internal banks, better leveraging parallelism [14]; some FTL translation data structures have nonuniform search times [16]; and fragmented SSDs are not able to prefetch data [8] or metadata [13]. Whatever the reason, SSDs show a gap between sequential and random reads, though not as great as on disks.

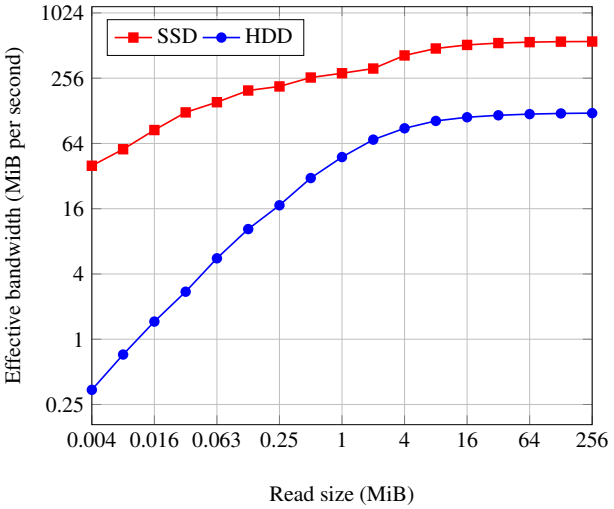


Figure 1: Effective bandwidth vs. read size (higher is better). Even on SSDs, large I/Os can yield an order of magnitude more bandwidth than small I/Os.

In order to avoid aging, file systems should avoid breaking large files into pieces significantly smaller than the NTS of the hardware. They should also group small files that are logically related (close in recursive traversal order) into clusters of size at least the NTS and store the clusters near each other on disk. We consider the major classes of file systems and explore the challenges each file system type encounters in achieving these two goals.

3.2 Allocation Strategies and Aging

The major file systems currently in use can be roughly categorized as B-tree-based, such as XFS, ZFS, and Btrfs, update-in-place, such as ext4, and log-structured, such as F2FS [15]. The research file system that we consider, BetrFS, is based on B$^\varepsilon$-trees. Each of these fundamental designs creates different aging considerations, discussed in turn below. In later sections, we present experimental validation for the design principles presented below.

B-trees. The aging profile of a B-tree depends on the leaf size. If the leaves are much smaller than the NTS, then the B-tree will age as the leaves are split and merged, and thus moved around on the storage device.

Making leaves as large as the NTS increases *write*

amplification, or the ratio between the amount of data changed and the amount of data written to storage. In the extreme case, a single-bit change to a B-tree leaf can cause the entire leaf to be rewritten. Thus, B-trees are usually implemented with small leaves. Consequently, we expect them to age under a wide variety of workloads.

In Section 6, we show that the aging of Btrfs is inversely related to the size of the leaves, as predicted. There are, in theory, ways to mitigate the aging due to B-tree leaf movements. For example, the leaves could be stored in a packed memory array [5]. However, such an arrangement might well incur an unacceptable performance overhead to keep the leaves arranged in logical order, and we know of no examples of B-trees implemented with such leaf-arrangement algorithms.

Write-Once or Update-in-Place Filesystems. When data is written once and never moved, such as in update-in-place file systems like ext4, sequential order is very difficult to maintain: imagine a workload that writes two files to disk, and then creates files that should logically occur between them. Without moving one of the original files, data cannot be maintained sequentially. Such pathological cases abound, and the process is quite brittle. As noted above, delayed allocation is an attempt to mitigate the effects of such cases by batching writes and updates before committing them to the overall structure.

B$^\varepsilon$-trees. B$^\varepsilon$-trees batch changes to the file system in a sequence of cascading logs, one per node of the tree. Each time a node overflows, it is flushed to the next node. The seeming disadvantage is that data is written many times, thus increasing the write amplification. However, each time a node is modified, it receives many changes, as opposed to B-tree, which might receive only one change. Thus, a B$^\varepsilon$-tree has asymptotically lower write amplification than a B-tree. Consequently, it can have much larger nodes, and typically does in implementation. BetrFS uses a B$^\varepsilon$-tree with 4MiB nodes.

Since 4MiB is around the NTS for our storage devices, we expect BetrFS not to age—which we verify below.

Log-structured merge trees (LSMs) [20] and other write-optimized dictionaries can resist aging, depending on the implementation. As with B$^\varepsilon$-trees, it is essential that node sizes match the NTS, the schema reflect logical access order, and enough writes are batched to avoid heavy write amplification.

3.3 Measuring File System Fragmentation

This section explains the two measures for file system fragmentation used in our evaluation: recursive scan latency and dynamic layout score, a modified form of Smith and Seltzer's layout score [23]. These measures are designed to capture both intra-file fragmentation and inter-file fragmentation.

Recursive grep test. One measure we present in the following sections is the wall-clock time required to perform a recursive grep in the root directory of the file system. This captures the effects of both inter- and intra-file locality, as it searches both large files and large directories containing many small files. We report search time per unit of data, normalizing by using ext4's du output. We will refer to this as the grep test.

Dynamic layout score. Smith and Seltzer's layout score [23] measures the fraction of blocks in a file or (in aggregate) a file system that are allocated in a contiguous sequence in the logical block space. We extend this score to the dynamic I/O patterns of a file system. During a given workload, we capture the logical block requests made by the file system, using blktrace, and measure the fraction that are contiguous. This approach captures the impact of placement decisions on a file system's access patterns, including the impact of metadata accesses or accesses that span files. A high dynamic layout score indicates good data and metadata locality, and an efficient on-disk organization for a given workload.

One potential shortcoming of this measure is that it does not distinguish between small and large discontiguities. Small discontiguities on a hard drive should induce fewer expensive mechanical seeks than large discontiguities in general, however factors such as track length, difference in angular placement and other geometric considerations can complicate this relationship. A more sophisticated measure of layout might be more predictive. We leave this for further research. On SSD, we have found that the length of discontiguities has a smaller effect. Thus we will show that dynamic layout score strongly correlates with grep test performance on SSD and moderately correlates on hard drive.

4 Experimental Setup

Each experiment compares several file systems: BetrFS, Btrfs, ext4, F2FS, XFS, and ZFS. We use the versions of XFS, Btrfs, ext4 and F2FS that are part of the 3.11.10 kernel, and ZFS 0.6.5-234_ge0ab3ab, downloaded from the zfsonlinux repository on `www.github.com`. We used BetrFS 0.3 in the experiments[1]. We use default recommended file system settings unless otherwise noted. Lazy inode table and journal initialization are turned off on ext4, pushing more work onto file system creation time and reducing experimental noise.

All experimental results are collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, a 500 GB, 7200 RPM ATA Seagate Barracuda ST500DM002 disk with a 4096 B block size, and a 240 GB Sandisk Extreme Pro—both disks used SATA 3.0. Each file system's block size is set to 4096 B. Unless

[1] Available at `github.com/oscarlab/betrfs`

otherwise noted, all experiments are cold-cache.

The system runs 64-bit Ubuntu 13.10 server with Linux kernel version 3.11.10 on a bootable USB stick. All HDD tests are performed on two 20GiB partitions located at the outermost region of the drive. For the SSD tests, we additionally partition the remainder of the drive and fill it with random data, although we have preliminary data that indicates this does not affect performance.

5 Fragmentation Microbenchmarks

We present several simple microbechmarks, each designed around a write/update pattern for which it is difficult to ensure both fast writes in the moment and future locality. These microbenchmarks isolate and highlight the effects of both intra-file fragmentation and inter-file fragmentation and show the performance impact aging can have on read performance in the worst cases.

Intrafile Fragmentation. When a file grows, there may not be room to store the new blocks with the old blocks on disk, and a single file's data may become scattered.

Our benchmark creates 10 files by first creating each file of an initial size, and then appending between 0 and 100 4KiB chunks of random data in a round-robin fashion until each file is 400KiB. In the first round the initial size is 400KiB, so each entire file is written sequentially, one at a time. In subsequent rounds, the initial size becomes smaller, so that the number of round-robin chunks increases until in the last round the data is written entirely with a round-robin of 4KiB chunks. After all the files are written, the disk cache is flushed by remounting, and we wait for 90 seconds before measuring read performance. Some file systems appear to perform background work immediately after mounting that introduced experimental noise; 90 seconds ensures the file system has quiesced.

The aging process this microbenchmark emulates is multiple files growing in length. The file system must allocate space for these files somewhere, but eventually the file must either be moved or fragment.

Given that the data set size is small and the test is designed to run in a short time, an fsync is performed after each file is written in order to defeat deferred allocation. Similar results are obtained if the test waits for 5 seconds between each append operation. If fewer fsyncs are performed or less waiting time is used, then the performance differences are smaller, as the file systems are able to delay allocation, rendering a more contiguous layout.

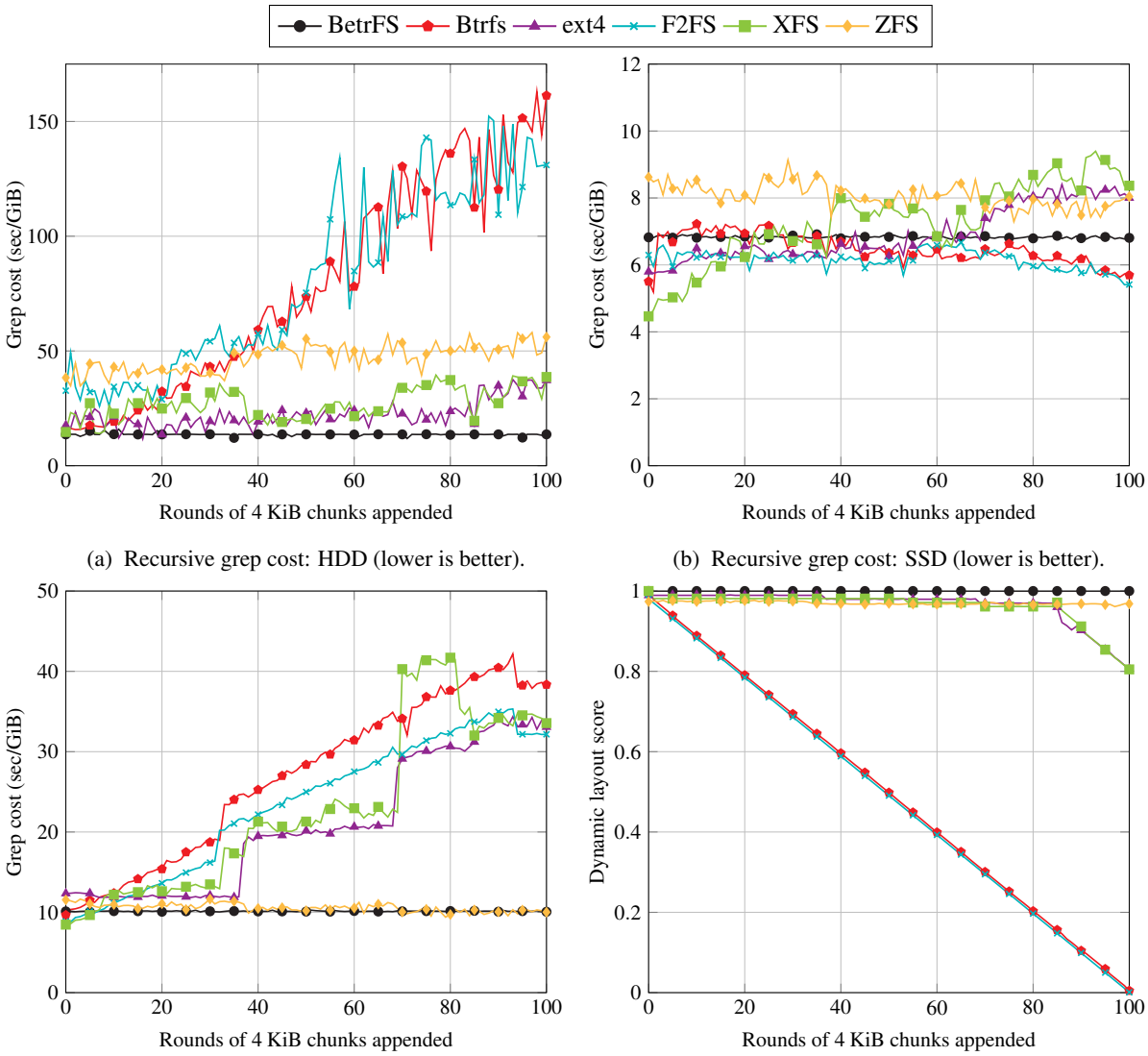
The performance of these file systems on an HDD and SSD are summarized in Figures 2. On HDD, the layout scores generally correlate (-0.93) with the performance of the file systems. On SSD, the file systems all perform similarly (note the scale of the y-axis). In some cases, such as XFS, ext4, and ZFS, there is a correlation, albeit at a small scale. For Btrfs, ext4, XFS, and F2FS, the performance is hidden by read-ahead in the OS or in the case of Btrfs also in the file system itself. If we disable read-ahead, shown in Figure 2c, the performance is more clearly correlated ($-.67$) with layout score. We do note that this relationship on an SSD is still not precise; SSDs are sufficiently fast that factors such as CPU time can also have a significant effect on performance.

Because of the small amount of data and number of files involved in this microbenchmark, we can visualize the layout of the various file systems, shown in Figure 3. Each block of a file is represented by a small vertical bar, and each bar is colored uniquely to one of the ten files. Contiguous regions form a colored rectangle. The visualization suggests, for example, that ext4 both tries to keep files and eventually larger file fragments sequential, whereas Btrfs and F2FS interleave the round robin chunks on the end of the sequential data. This interleaving can help explain why Btrfs and F2FS perform the way they do: the interleaved sections must be read through in full each time a file is requested, which by the end of the test takes roughly 10 times as long. ext4 and XFS manage to keep the files in larger extents, although the extents get smaller as the test progresses, and, by the end of the benchmark, these file systems also have chunks of interleaved data; this is why ext4 and XFS's dynamic layout scores decline. ZFS keeps the files in multiple chunks through the test; in doing so it sacrifices some performance in all states, but does not degrade.

Unfortunately, this sort of visualization doesn't work for BetrFS, because this small amount of data fits entirely in a leaf. Thus, BetrFS will read all this data into memory in one sequential read. This results is some read amplification, but, on an HDD, only one seek.

Interfile Fragmentation. Many workloads read multiple files with some logical relationship, and frequently those files are placed in the same directory. Interfile fragmentation occurs when files which are related—in this case being close together in the directory tree—are not collocated in the LBA space.

We present a microbenchmark to measure the impact of namespace creation order on interfile locality. It takes a given "real-life" file structure, in this case the Tensorflow repository obtained from `github.com`, and replaces each of the files by 4KiB of random data. This gives us a "natural" directory structure, but isolates the effect of file ordering without the influence of intrafile layout. The benchmark creates a sorted list of the files as well as two random permutations of that list. On each round of the test, the benchmark copies all of the files, creating directories as needed with `cp --parents`. However, on the nth round, it swaps the order in which the first n% of files appearing in the random permutations are copied. Thus, the first round will be an in-order copy, and subsequent



(a) Recursive grep cost: HDD (lower is better).

(b) Recursive grep cost: SSD (lower is better).

(c) Recursive grep cost: SSD, no readahead (lower is better).

(d) Dynamic layout score (higher is better).

Figure 2: Intrafile benchmark: 4KiB chunks are appended round-robin to sequential data to create 10 400KiB files. Dynamic layout scores generally correlate with read performance as measured by the recursive grep test; on an SSD, this effect is hidden by the readahead buffer.

rounds will be copied in a progressively more random order until the last round is a fully random-order copy.

The results of this test are shown in Figure 4. On hard drive, all the file systems except BetrFS and XFS show a precipitous performance decline even if only a small percentage of the files are copied out of order. F2FS's performance is poor enough to be out of scale for this figure, but it ends up taking over 4000 seconds per GiB at round 100; this is not entirely unexpected as it is not designed to be used on hard drive. XFS is somewhat more stable, although it is 13-35 times slower than drive bandwidth throughout the test, even on an in-order copy. BetrFS consistently performs around 1/3 of bandwidth, which by the end of the test is 10 times faster than XFS, and 25 times faster than the other file systems. The dynamic layout scores are moderately correlated with this performance (-0.57).

On SSD, half the file systems perform stably throughout the test with varying degrees of performance. The other half have a very sharp slowdown between the in-order state and the 10% out-of-order state. These two modes are reflected in their dynamic layout scores as well. While ext4 and ZFS are stable, their performance is worse than the best cases of several other file systems. BetrFS is the only file system with stable fast performance; it is faster in every round than any other file system even in their best case: the in-order copy. In this cases the performance strongly correlates with the dy-

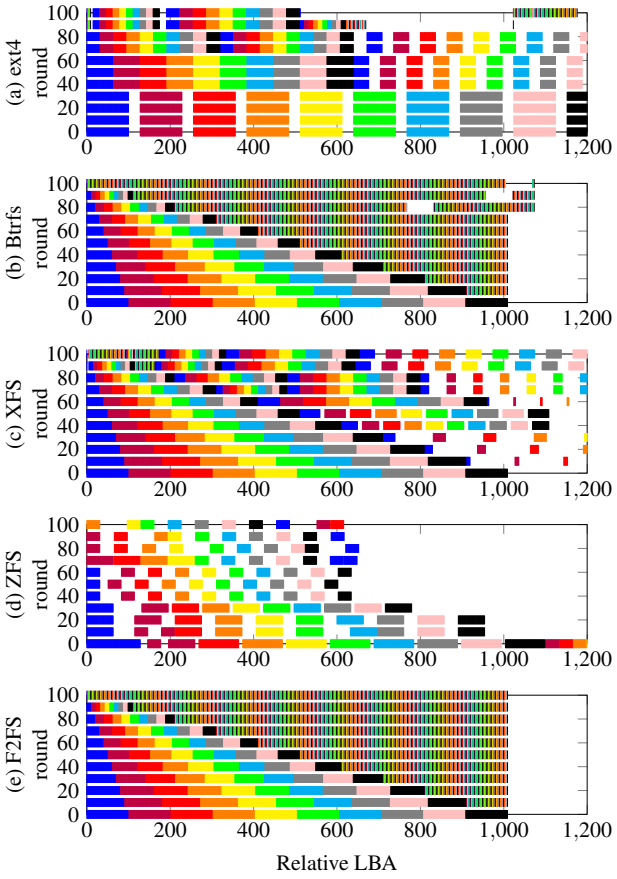


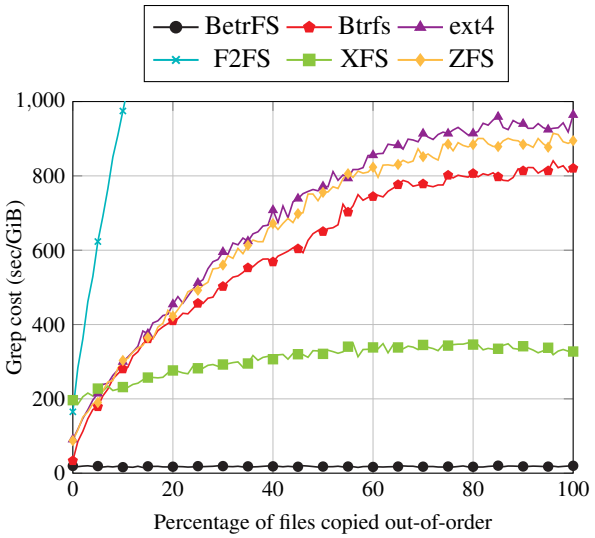
Figure 3: Intrafile benchmark layout visualization. Each color represents blocks of a file. The x-axis is the logical block address (LBA) of the file block relative to the first LBA of any file block, and y-axis is the round of the experiment. Rectangle sizes indicate contiguous placement, where larger is better. The brown regions with vertical lines indicate interleaved blocks of all 10 files. Some blocks are not shown for ext4, XFS and ZFS.

namic layout score (−0.83).

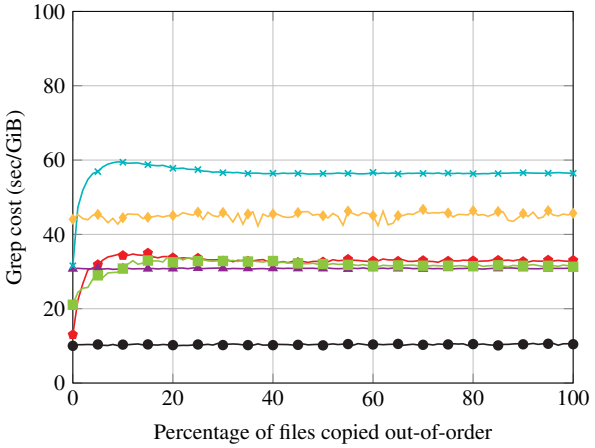
6 Application Level Read-Aging: Git

To measure aging in the "real-world," we create a workload designed to simulate a developer using git to work on a collaborative project.

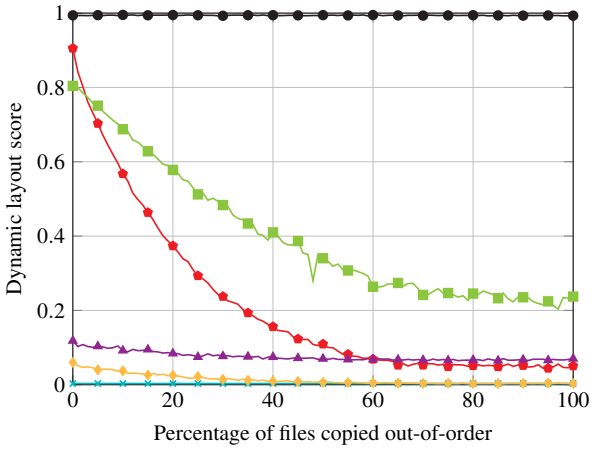
Git is a distributed version control system that enables collaborating developers to synchronize their source code changes. Git users ***pull*** changes from other developers, which then get merged with their own changes. In a typical workload, a Git user may perform pulls multiple times per day over several years in a long-running project. Git can synchronize all types of file system changes, so performing a Git pull may result in the creation of new source files, deletion of old files, file renames, and file modifications. Git also maintains its own internal data structures, which it updates during pulls.



(a) Recursive grep cost: HDD (Lower is better).



(b) Recursive grep cost: SSD (Lower is better).



(c) Dynamic layout score (higher is better).

Figure 4: Interfile benchmark: The TensorFlow github repository with all files replaced by 4KiB random data and copied in varying degrees of order. Dynamic layout scores again are predictive of recursive grep test performance.

Thus, Git performs many operations which are similar to those shown in Section 5 that cause file system aging.

We present a git benchmark that performs 10,000 pulls from the Linux git repository, starting from the initial commit. After every 100 pulls, the benchmark performs a recursive grep test and computes the file system's dynamic layout score. This score is compared to the same contents copied to a freshly formatted partition.

On a hard disk (Figure 5a), there is a clear aging trend in all file systems except BetrFS. By the end of the experiment, all the file systems except BetrFS show performance drops under aging on the order of at least 3x and as much as 15x relative to their unaged versions. All are at least 15x worse than BetrFS. In all of the experiments in this section, F2FS ages considerably more than all other file systems, commensurate with significantly lower layout scores than the other file systems—indicating less effective locality in data placement. The overall correlation between grep performance and dynamic layout score is moderate, at -0.41.

On an SSD (Figure 5c), Btrfs and XFS show clear signs of aging, although they converge to a fully aged configuration after only about 1,000 pulls. While the effect is not as drastic as on HDD, in all the traditional file systems we see slowdowns of 2x-4x over BetrFS, which does not slow down. In fact, aged BetrFS on the HDD outperforms all the other aged file systems on an SSD, and is close even when they are unaged. Again, this performance decline is strongly correlated (-0.79) with the dynamic layout scores.

The aged and unaged performance of ext4 and ZFS are comparable, and slower than several other file systems. We believe this is because the average file size decreases over the course of the test, and these file systems are not as well-tuned for small files. To test this hypothesis, we constructed synthetic workloads similar to the interfile fragmentation microbenchmark (Section 5), but varied the file size (in the microbenchmark it was uniformly 4KB). Figure 6 shows both the measured, average file size of the git workload (one point is one pull), and the microbenchmark. Overall, there is a clear relationship between the average file size and grep cost.

The zig-zag pattern in the graphs is created by an automatic garbage collection process in Git. Once a certain number of "loose objects" are created (in git terminology), many of them are collected and compressed into a "pack." At the file system level, this corresponds to merging numerous small files into a single large file. According to the Git manual, this process is designed to "reduce disk space and increase performance," so this is an example of an application-level attempt to mitigate file system aging. If we turn off the git garbage collection, as show in Figures 5b, 5d and 5f, the effect of aging is even more pronounced, and the zig-zags essentially disappear.

On both the HDD and SSD, the same patterns emerge as with garbage collection on, but exacerbated: F2FS aging is by far the most extreme. ZFS ages considerably on the HDD, but not on the SSD. ZFS on SSD and ext4 perform worse than the other file systems (except F2FS aged), but do not age particularly. XFS and Btrfs both aged significantly, around 2x each, and BetrFS has strong, level performance in both states. This performance correlates with dynamic layout score both on SSD (-0.78) and moderately so on HDD (-0.54).

We note that this analysis, both of the microbenchmarks and of the git workload, runs counter to the commonly held belief that locality is solely a hard drive issue. While the random read performance of solid state drives does somewhat mitigate the aging effects, aging clearly has a major performance impact.

Git Workload with Warm Cache. The tests we have presented so far have all been performed with a cold cache, so that they more or less directly test the performance of the file systems' on-disk layout under various aging conditions. In practice, however, some data will be in cache, and so it is natural to ask how much the layout choices that the file system makes will affect the overall performance with a warm cache.

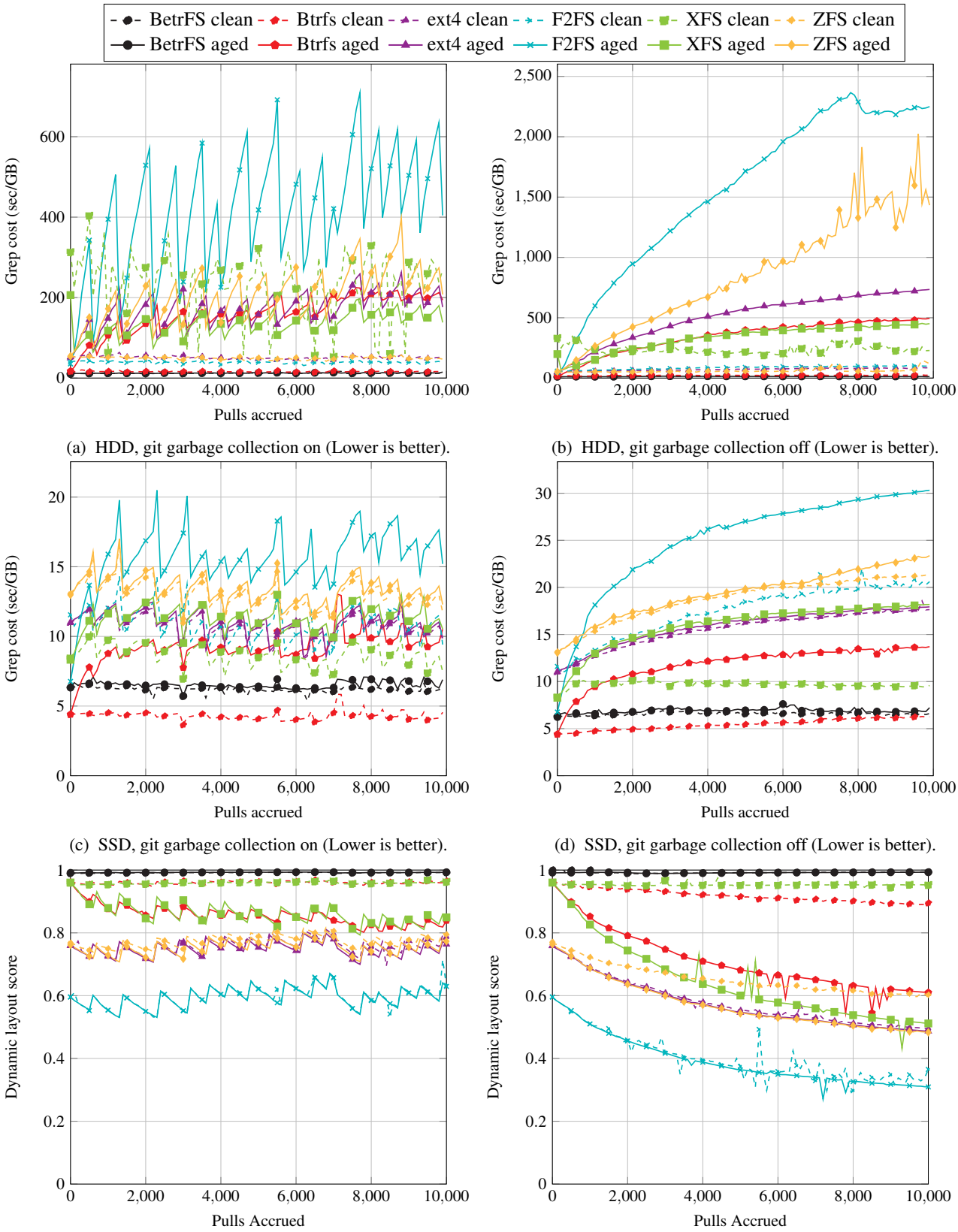
We evaluate the sensitivity of the git workloads to varying amounts of system RAM. We use the same procedure as above, except that we do not flush any caches or remount the hard drive between iterations. This test is performed on a hard drive with git garbage collection off. The size of the data on disk is initially about 280MiB and grows throughout the test to approximately 1GiB.

The results are summarized in Figure 7. We present data for ext4 and F2FS; the results for Btrfs, XFS and ZFS are similar. BetrFS is a research prototype and unstable under memory pressure; although we plan to fix these issues in the future, we omit this comparison.

In general, when the caches are warm and there is sufficient memory to keep all the data in cache, then the read is very fast. However, as soon as there is no longer sufficient memory, the performance of the aged file system with a warm cache is generally worse than unaged with a cold cache. In general, unless all data fits into DRAM, a good layout matters more than a having a warm cache.

Btrfs Node-Size Trade-Off. Btrfs allows users to specify the node size of its metadata B-tree at creation time. Because small files are stored in the metadata B-tree, a larger node size results in a less fragmented file system, at a cost of more expensive metadata updates.

We present the git test with a 4KiB node size, the default setting, as well as 8KiB, 16KiB, 32KiB, and 64KiB (the maximum). Figure 8a shows similar performance graphs to Figure 5, one line for each node size. The 4KiB node size has the worst read performance by the end of



(a) HDD, git garbage collection on (Lower is better).

(b) HDD, git garbage collection off (Lower is better).

(c) SSD, git garbage collection on (Lower is better).

(d) SSD, git garbage collection off (Lower is better).

(e) Dynamic layout score: git garbage collection on (Higher is better).

(f) Dynamic layout score: git garbage collection off (Higher is better).

Figure 5: Git read-aging experimental results: On-disk layout as measured by dynamic layout score generally is predictive of read performance.

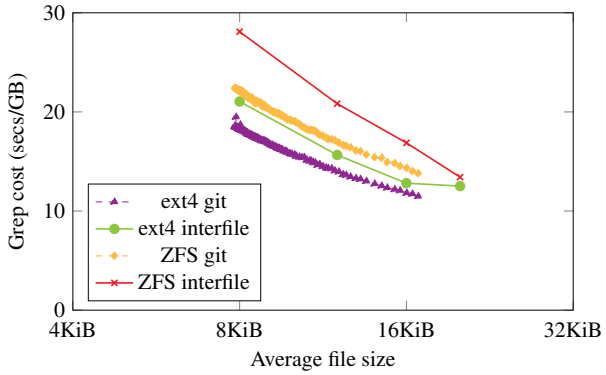


Figure 6: Average file size versus unaged grep costs (lower is better) on SSD. Each point in the git line is the average file size for the git experiment, compared to a microbenchmark with all files set to a given size.

the test, and the performance consistently improves as we increase the node size all the way to 64KiB. Figure 8b plots the number of 4KiB blocks written to disk between each test (within the 100 pulls). As expected, the 64KiB node size writes the maximum number of blocks and the 4KiB node writes the least. We thus demonstrate—as predicted by our model—that aging is reduced by a larger block size, but at the cost of write-amplification.

7 Application Level Aging: Mail Server

In addition to the git workload, we evaluate aging with the Dovecot email server. Dovecot is configured with the Maildir backend, which stores each message in a file, and each inbox in a directory. We simulate 2 users, each having 80 mailboxes receiving new email, deleting old emails, and searching through their mailboxes.

A cycle or "day" for the mailserver comprises of 8,000 operations, where each operation is equally likely to be a insert or a delete, corresponding to receiving a new email or deleting an old one. Each email is a string of random characters, the length of which is uniformly distributed over the range [1, 32K]. Each mailbox is initialized with 1,000 messages, and, because inserts and deletes are balanced, mailbox size tends to stay around 1,000. We simulate the mailserver for 100 cycles and after each cycle we perform a recursive grep for a random string. As in our git benchmarks, we then copy the partition to a freshly formatted file system, and run a recursive grep.

Figure 9 shows the read costs in seconds per GiB of the grep test on hard disk. Although the unaged versions of all file systems show consistent performance over the life of the benchmark, the aged versions of ext4, Btrfs, XFS and ZFS all show significant degradation over time. In particular, aged ext4 performance degrades by 4.4×, and is 28× slower than aged BetrFS. XFS slows down by a factor of 7 and Btrfs by a factor of 30. ZFS slows down drastically, taking about 20 minutes per GiB by

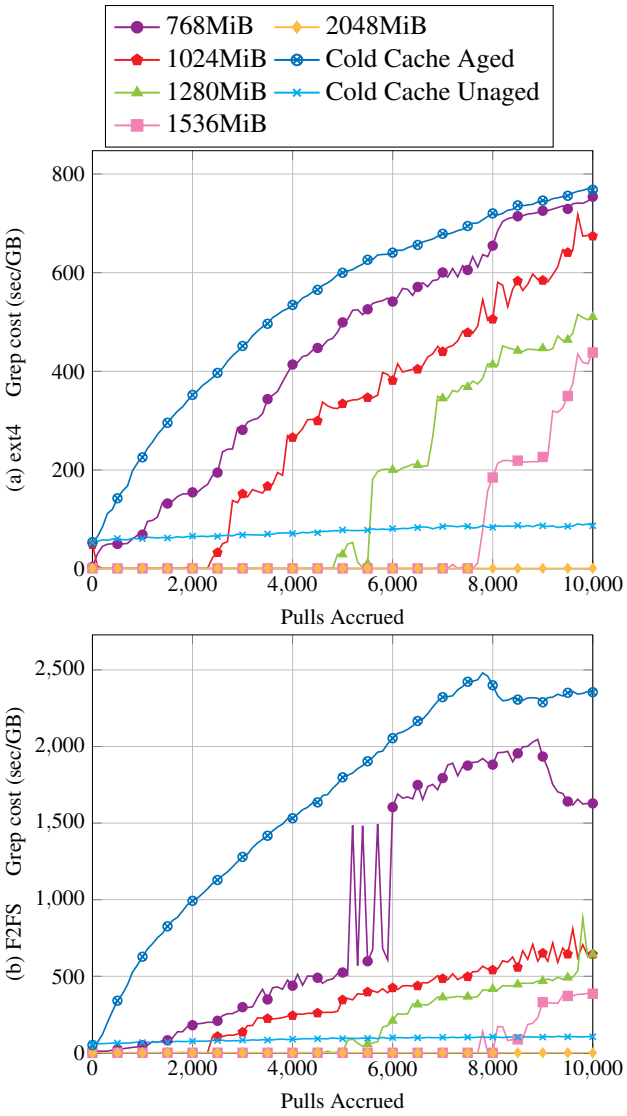


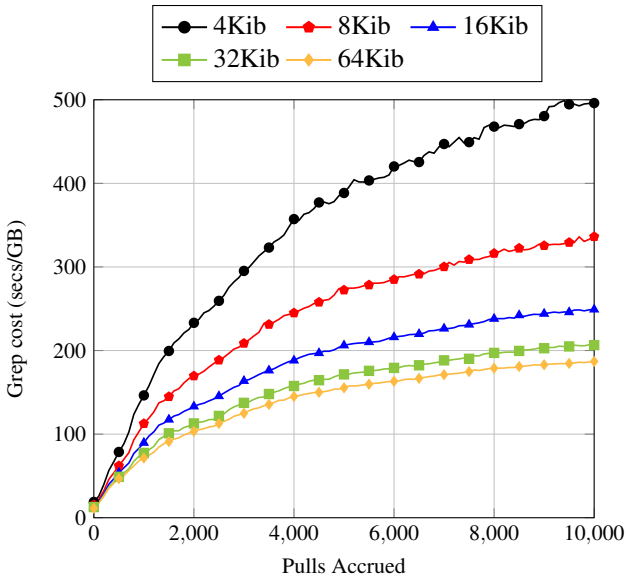
Figure 7: Grep costs as a function of git pulls with warm cache and varying system RAM on ext4 (top) and F2FS (bottom). Lower is better.

cycle 20. However, the aged version of BetrFS does not slow down. As with the other HDD experiments, dynamic layout score is moderately correlated (−0.63) with grep cost.

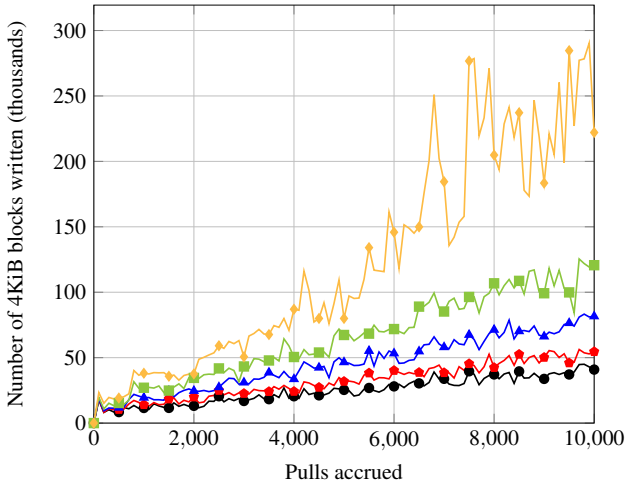
8 Conclusion

The experiments above suggest that conventional wisdom on fragmentation, aging, allocation and file systems is inadequate in several ways.

First, while it may seem intuitive to write data as few times as possible, writing data only once creates a tension between the logical ordering of the file system's current state and the potential to make modifications without disrupting the future order. Rewriting data multiple times allows the file system to maintain locality. The

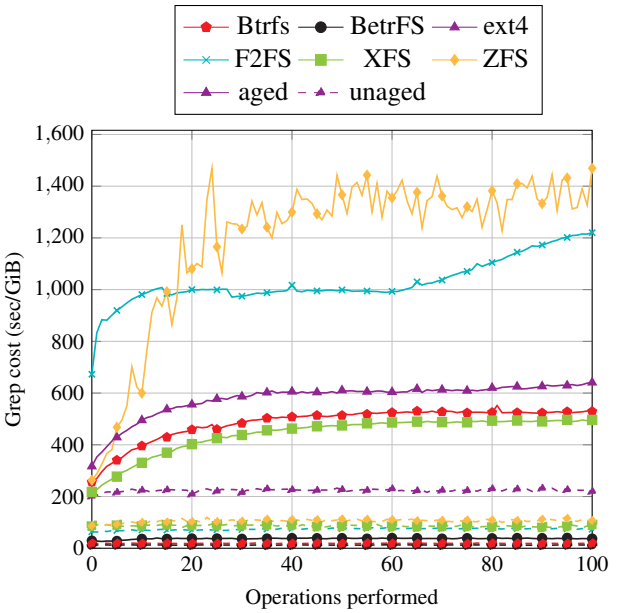


(a) Grep cost at different node sizes (lower is better).

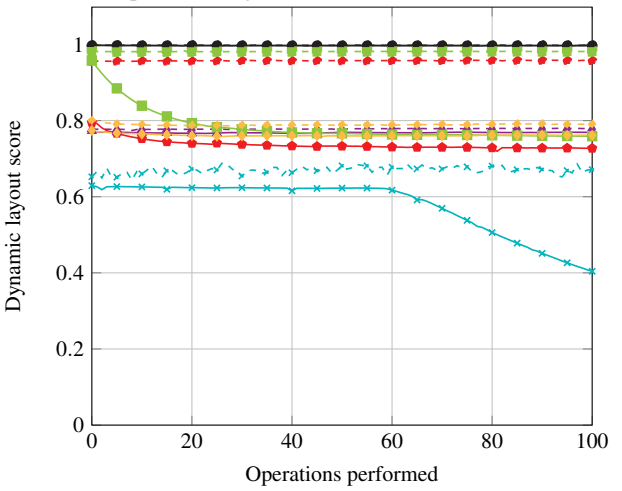


(b) Write amplification at different node sizes (lower is better).

Figure 8: Aging and write amplification on Btrfs, with varying node sizes, under the git aging benchmark.



(a) Grep cost during mailserver workload (lower is better).



(b) Mailserver layout (higher is better).

Figure 9: Mailserver performance and layout scores.

overhead of these multiple writes can be managed by rewriting data in batches, as is done in write-optimized dictionaries.

For example, in BetrFS, data might be written as many as a logarithmic number of times, whereas in ext4, it will be written once, yet BetrFS in general is able to perform as well as or better than an unaged ext4 file system and significantly outperforms aged ext4 file systems.

Second, today's file system heuristics are not able to maintain enough locality to enable reads to be performed at the disks natural transfer size. And since the natural transfer size on a rotating disk is a function of the seek time and bandwidth, it will tend to increase with time. Thus we expect this problem to possibly become worse with newer hardware, not better.

We experimentally confirmed our expectation that non-write-optimized file systems would age, but we were surprised by how quickly and dramatically aging impacts performance. This rapid aging is important: a user's experience with unaged file systems is likely so fleeting that they do not notice performance degradation. Instead, the performance costs of aging are built into their expectations of file system performance.

Finally, because representative aging is a difficult goal, simulating multi-year workloads, many research papers benchmark on unaged file systems. Our results indicate that it is relatively easy to quickly drive a file system into an aged state—even if this state is not precisely the state of the file system after, say, three years of typical use—and this degraded state can be easily measured.

Acknowledgments

We thank the anonymous reviewers and our shepherd Philip Shilane for their insightful comments on earlier drafts of the work. Part of this work was done while Jiao, Porter, Yuan, and Zhan were at Stony Brook University. This research was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, CNS-1161541, IIS-1247750, CCF-1314547, and VMware.

References

[1] Fragging wonderful: The truth about defragging your ssd. `http://www.pcworld.com/article/2047513/fragging-wonderful-the-truth-about-defragging-your-ssd.html`. Accessed 25 September 2016.

[2] AGRAWAL, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Generating realistic impressions for file-system benchmarking. *ACM Transactions on Storage (TOS)5*, 4 (Dec. 2009), art. 16.

[3] AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. A five-year study of file-system metadata. *Trans. Storage 3*, 3 (Oct. 2007).

[4] AHN, W. H., KIM, K., CHOI, Y., AND PARK, D. DFS: A de-fragmented file system. In *Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)* (2002), pp. 71–80.

[5] BENDER, M. A., DEMAINE, E., AND FARACH-COLTON, M. Cache-oblivious B-trees. *SIAM J. Comput. 35*, 2 (2005), 341–358.

[6] BONWICK, J., AND MOORE, B. ZFS: The last word in file systems. In *SNIA Developers Conference* (Santa Clara, CA, USA, Sept. 2008). Slides at `http://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf`, talk at `https://blogs.oracle.com/video/entry/zfs_the_last_word_in`. Accessed 10 May 2016.

[7] CARD, R., TS'O, T., AND TWEEDIE, S. Design and implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux* (Amsterdam, NL, Dec. 8–9 1994), pp. 1–6. `http://e2fsprogs.sourceforge.net/ext2intro.html`.

[8] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2009), SIGMETRICS '09, ACM, pp. 181–192.

[9] DOWNEY, A. B. The structural cause of file size distributions. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2001), SIGMETRICS '01, ACM, pp. 328–329.

[10] 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).

[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)* (Santa Clara, CA, USA, Feb. 22–25 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 (Nov. 2015), art. 18.

[13] JI, C., CHANG, L.-P., SHI, L., WU, C., LI, Q., AND XUE, C. J. An empirical study of file-system fragmentation in mobile storage systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (Denver, CO, 2016), USENIX Association.

[14] JUNG, M., AND KANDEMIR, M. Revisiting widely held ssd expectations and rethinking system-level implications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (New York, NY, USA, 2013), ACM, pp. 203–216.

[15] 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)* (Santa Clara, CA, USA, Feb. 22–25 2015), pp. 273–286.

[16] MA, D., FENG, J., AND LI, G. A survey of address translation technologies for flash memories. *ACM Comput. Surv. 46*, 3 (Jan. 2014), 36:1–36:39.

[17] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Ottowa Linux Symposium (OLS)* (Ottowa, ON, Canada, 2007), vol. 2, pp. 21–34.

[18] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS) 2*, 3 (Aug. 1984), 181–197.

[19] MIN, C., KIM, K., CHO, H., LEE, S., AND EOM, Y. I. SFS: random write considered harmful in solid state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (San Jose, CA, USA, Feb. 14–17 2012), art. 12.

[20] 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. http://dx.doi.org/10.1007/s002360050048doi: 10.1007/s002360050048.

[21] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (Aug. 2013), art. 9.

[22] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A comparison of file system workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2000), ATEC '00, USENIX Association, pp. 4–4.

[23] SMITH, K. A., AND SELTZER, M. File system aging — increasing the relevance of file system benchmarks. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (Seattle, WA, June 15–18 1997), pp. 203–213.

[24] 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* (San Diego, CA, USA, Jan.22–26 1996), art. 1.

[25] TWEEDIE, S. EXT3, journaling filesystem. In *Ottowa Linux Symposium* (Ottowa, ON, Canada, July 20 2000).

[26] WIRZENIUS, L., OJA, J., STAFFORD, S., AND WEEKS, A. *Linux System Administrator's Guide*. The Linux Documentation Project, 2004. `http://www.tldp.org/LDP/sag/sag.pdf`. Version 0.9.

[27] YUAN, J., ZHAN, Y., JANNEN, W., PANDEY, P., AKSHINTALA, A., CHANDNANI, K., DEO, P., KASHEFF, Z., WALSH, L., BENDER, M., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Optimizing every operation in a write-optimized file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (Santa Clara, CA, USA, Feb. 22–25 2016), pp. 1–14. `https://www.usenix.org/conference/fast16/technical-sessions/presentation/yuan`.

[28] ZHU, N., CHEN, J., AND CHIUEH, T.-C. TBBT: Scalable and accurate trace replay for file server evaluation. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (Santa Clara, CA, USA, Feb. 16–19 2005), pp. 323–336.

To FUSE or Not to FUSE: Performance of User-Space File Systems

Bharath Kumar Reddy Vangoor[1], Vasily Tarasov[2], and Erez Zadok[1]
[1]*Stony Brook University* *and* [2]*IBM Research—Almaden*

Abstract

Traditionally, file systems were implemented as part of OS kernels. However, as complexity of file systems grew, many new file systems began being developed in user space. Nowadays, user-space file systems are often used to prototype and evaluate new approaches to file system design. Low performance is considered the main disadvantage of user-space file systems but the extent of this problem has never been explored systematically. As a result, the topic of user-space file systems remains rather controversial: while some consider user-space file systems a toy not to be used in production, others develop full-fledged production file systems in user space. In this paper we analyze the design and implementation of the most widely known user-space file system framework—FUSE—and characterize its performance for a wide range of workloads. We instrumented FUSE to extract useful statistics and traces, which helped us analyze its performance bottlenecks and present our analysis results. Our experiments indicate that depending on the workload and hardware used, performance degradation caused by FUSE can be completely imperceptible or as high as –83% even when optimized; and relative CPU utilization can increase by 31%.

1 Introduction

File systems offer a common interface for applications to access data. Although micro-kernels implement file systems in user space [1, 16], most file systems are part of monolithic kernels [6, 22, 34]. Kernel implementations avoid the high message-passing overheads of micro-kernels and user-space daemons [7, 14].

In recent years, however, user-space file systems rose in popularity for four reasons. (1) Several stackable file systems add specialized functionality over existing file systems (e.g., deduplication and compression [19, 31]). (2) In academia and R&D settings, this framework enabled quick experimentation and prototyping of new approaches [3, 9, 15, 21, 40]. (3) Several existing kernel-level file systems were ported to user space (e.g., ZFS [45], NTFS [25]). (4) More companies rely on user-space implementations: IBM'S GPFS [30] and LTFS [26], Nimble Storage's CASL [24], Apache's HDFS [2], Google File System [13], RedHat's GlusterFS [29], Data Domain's DDFS [46], etc.

Increased file systems complexity is a contributing factor to user-space file systems' growing popularity (e.g., Btrfs is over 85 KLoC). User space code is easier to develop, port, and maintain. Kernel bugs can crash whole systems, whereas user-space bugs' impact is more contained. Many libraries and programming languages are available in user-space in multiple platforms.

Although user-space file systems are not expected to displace kernel file systems entirely, they undoubtedly occupy a growing niche, as some of the more heated debates between proponents and opponents indicate [20, 39, 41]. The debates center around two trade-off factors: (1) how large is the performance overhead caused by a user-space implementations and (2) how much easier is it to develop in user space. Ease of development is highly subjective, hard to formalize and therefore evaluate; but performance is easier to evaluate empirically. Oddly, little has been published on the performance of user-space file system frameworks.

In this paper we use a popular user-space file system framework, *FUSE*, and characterize its performance. We start with a detailed explanation of FUSE's design and implementation for four reasons: (1) the architecture is somewhat complex; (2) little information on internals is available publicly; (3) FUSE's source code can be difficult to analyze, with complex asynchrony and user-kernel communications; and (4) as FUSE's popularity grows, a detailed analysis of its implementation becomes of high value to many.

We developed a simple pass-through stackable file system in FUSE and then evaluated its performance when layered on top of Ext4 compared to native Ext4. We used a wide variety of micro- and macro-workloads, and different hardware using basic and optimized configurations of FUSE. We found that depending on the workload and hardware, FUSE can perform as well as Ext4, but in the worst cases can be 3× slower. Next, we designed and built a rich instrumentation system for FUSE to gather detailed performance metrics. The statistics extracted are applicable to any FUSE-based systems. We then used this instrumentation to identify bottlenecks in FUSE, and to explain why, for example, its performance varied greatly for different workloads.

2 FUSE Design

FUSE—Filesystem in Userspace—is the most widely used user-space file system framework [35]. According to the most modest estimates, at least 100 FUSE-based file systems are readily available on the Web [36]. Although other, specialized implementations of user-space file systems exist [30, 32, 42], we selected FUSE for this study because of its high popularity.

Although many file systems were implemented using

FUSE—thanks mainly to the simple API it provides—little work was done on understanding its internal architecture, implementation, and performance [27]. For our evaluation it was essential to understand not only FUSE's high-level design but also some details of its implementation. In this section we first describe FUSE's basics and then we explain certain important implementation details. FUSE is available for several OSes: we selected Linux due to its wide-spread use. We analyzed the code of and ran experiments on the latest stable version of the Linux kernel available at the beginning of the project—v4.1.13. We also used FUSE library commit #386b1b; on top of FUSE v2.9.4, this commit contains several important patches which we did not want exclude from our evaluation. We manually examined all new commits up to the time of this writing and confirmed that no new major features or improvements were added to FUSE since the release of the selected versions.

2.1 High-Level Architecture

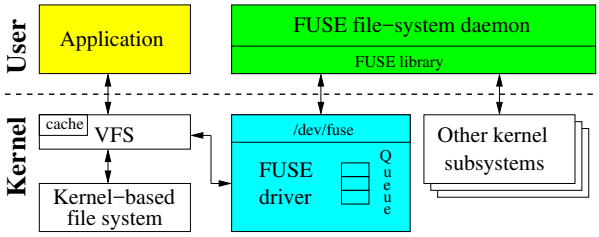


Figure 1: FUSE high-level architecture.

FUSE consists of a kernel part and a user-level daemon. The kernel part is implemented as a Linux kernel module that, when loaded, registers a *fuse* file-system driver with Linux's VFS. This *Fuse* driver acts as a proxy for various specific file systems implemented by different user-level daemons.

In addition to registering a new file system, FUSE's kernel module also registers a `/dev/fuse` block device. This device serves as an interface between user-space FUSE daemons and the kernel. In general, daemon reads FUSE requests from `/dev/fuse`, processes them, and then writes replies back to `/dev/fuse`.

Figure 1 shows FUSE's high-level architecture. When a user application performs some operation on a mounted FUSE file system, the VFS routes the operation to FUSE's kernel driver. The driver allocates a FUSE request structure and puts it in a FUSE queue. At this point, the process that submitted the operation is usually put in a wait state. FUSE's user-level daemon then picks the request from the kernel queue by reading from `/dev/fuse` and processes the request. Processing the request might require re-entering the kernel again: for example, in case of a stackable FUSE file system, the daemon submits operations to the underlying file system (e.g., Ext4); or in case of a block-based FUSE file system, the daemon reads or writes from the block device. When done with processing the request, the FUSE daemon writes the response back to `/dev/fuse`; FUSE's kernel driver then marks the request as completed and wakes up the original user process.

Some file system operations invoked by an application can complete without communicating with the user-level FUSE daemon. For example, reads from a file whose pages are cached in the kernel page cache, do not need to be forwarded to the FUSE driver.

2.2 Implementation Details

We now discuss several important FUSE implementation details: the user-kernel protocol, library and API levels, in-kernel FUSE queues, splicing, multi-threading, and write-back cache.

Group (#)	Request Types
Special (3)	**INIT**, **DESTROY**, **INTERRUPT**
Metadata (14)	**LOOKUP**, **FORGET**, **BATCH_FORGET**, CREATE, UNLINK, LINK, RENAME, RENAME2, **OPEN**, **RELEASE**, STATFS, FSYNC, **FLUSH**, **ACCESS**
Data (2)	READ, WRITE
Attributes (2)	GETATTR, SETATTR
Extended Attributes (4)	SETXATTR, GETXATTR, LISTXATTR, REMOVEXATTR
Symlinks (2)	SYMLINK, READLINK
Directory (7)	MKDIR, RMDIR, **OPENDIR**, **RELEASEDIR**, READDIR, **READDIRPLUS**, FSYNCDIR
Locking (3)	GETLK, SETLK, SETLKW
Misc (6)	BMAP, FALLOCATE, MKNOD, IOCTL, POLL, NOTIFY_REPLY

Table 1: FUSE request types, by group (whose size is in parenthesis). Requests we discuss in the text are in bold.

User-kernel protocol. When FUSE's kernel driver communicates to the user-space daemon, it forms a *FUSE request* structure. Requests have different types depending on the operation they convey. Table 1 lists all 43 FUSE request types, grouped by their semantics. As seen, most requests have a direct mapping to traditional VFS operations: we omit discussion of obvious requests (e.g., READ, CREATE) and instead next focus on those less intuitive request types (marked bold in Table 1).

The INIT request is produced by the kernel when a file system is mounted. At this point user space and kernel negotiate (1) the protocol version they will operate on (7.23 at the time of this writing), (2) the set of mutually supported capabilities (e.g., READDIRPLUS or FLOCK support), and (3) various parameter settings (e.g., FUSE read-ahead size, time granularity). Conversely, the DESTROY request is sent by the kernel during the file system's unmounting process. When getting a DESTROY, the daemon is expected to perform all necessary

cleanups. No more requests will come from the kernel for this session and subsequent reads from `/dev/fuse` will return 0, causing the daemon to exit gracefully.

The INTERRUPT request is emitted by the kernel if any previously sent requests are no longer needed (e.g., when a user process blocked on a READ is terminated). Each request has a unique *sequence#* which INTERRUPT uses to identify victim requests. Sequence numbers are assigned by the kernel and are also used to locate completed requests when the user space replies.

Every request also contains a *node ID*—an unsigned 64-bit integer identifying the inode both in kernel and user spaces. The path-to-inode translation is performed by the LOOKUP request. Every time an existing inode is looked up (or a new one is created), the kernel keeps the inode in the inode cache. When removing an inode from the dcache, the kernel passes the FORGET request to the user-space daemon. At this point the daemon might decide to deallocate any corresponding data structures. BATCH_FORGET allows kernel to forget multiple inodes with a single request.

An OPEN request is generated, not surprisingly, when a user application opens a file. When replying to this request, a FUSE daemon has a chance to optionally assign a 64-bit *file handle* to the opened file. This file handle is then returned by the kernel along with every request associated with the opened file. The user-space daemon can use the handle to store per-opened-file information. E.g., a stackable file system can store the descriptor of the file opened in the underlying file system as part of FUSE's file handle. FLUSH is generated every time an opened file is closed; and RELEASE is sent when there are no more references to a previously opened file.

OPENDIR and RELEASEDIR requests have the same semantics as OPEN and RELEASE, respectively, but for directories. The READDIRPLUS request returns one or more directory entries like READDIR, but it also includes metadata information for each entry. This allows the kernel to pre-fill its inode cache (similar to NFSv3's READDIRPLUS procedure [4]).

When the kernel evaluates if a user process has permissions to access a file, it generates an ACCESS request. By handling this request, the FUSE daemon can implement custom permission logic. However, typically users mount FUSE with the *default_permissions* option that allows kernel to grant or deny access to a file based on its standard Unix attributes (ownership and permission bits). In this case no ACCESS requests are generated.

Library and API levels. Conceptually, the FUSE library consists of two levels. The lower level takes care of (1) receiving and parsing requests from the kernel, (2) sending properly formatted replies, (3) facilitating file system configuration and mounting, and (4) hiding potential version differences between kernel and user space. This part exports the *low-level FUSE API*.

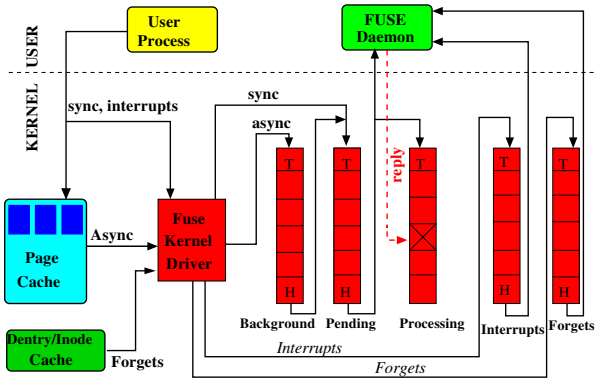


Figure 2: The organization of FUSE queues marked with their Head and Tail. The processing queue does not have a tail because the daemon replies in an arbitrary order.

The *High-level FUSE API* builds on top of the low-level API and allows developers to skip the implementation of the *path-to-inode* mapping. Therefore, neither inodes nor lookup operations exist in the high-level API, easing the code development. Instead, all high-level API methods operate directly on file paths. The high-level API also handles request interrupts and provides other convenient features: e.g., developers can use the more common `chown()`, `chmod()`, and `truncate()` methods, instead of the lower-level `setattr()`. File system developers must decide which API to use, by balancing flexibility vs. development ease.

Queues. In Section 2.1 we mentioned that FUSE's kernel has a request queue. FUSE actually maintains five queues as seen in Figure 2: (1) *interrupts*, (2) *forgets*, (3) *pending*, (4) *processing*, and (5) *background*. A request belongs to only one queue at any time. FUSE puts INTERRUPT requests in the interrupts queue, FORGET requests in the forgets queue, and synchronous requests (e.g., metadata) in the pending queue. When a file-system daemon reads from `/dev/fuse`, requests are transferred to the user daemon as follows: (1) Priority is given to requests in the interrupts queue; they are transferred to the user space before any other request. (2) FORGET and non-FORGET requests are selected fairly: for each 8 non-FORGET requests, 16 FORGET requests are transferred. This reduces the burstiness of FORGET requests, while allowing other requests to proceed. The oldest request in the pending queue is transferred to the user space and simultaneously moved to the processing queue. Thus, processing queue requests are currently processed by the daemon. If the pending queue is empty then the FUSE daemon is blocked on the read call. When the daemon replies to a request (by writing to `/dev/fuse`), the corresponding request is removed from the processing queue.

The background queue is for staging asynchronous requests. In a typical setup, only read requests go to

the background queue; writes go to the background queue too but only if the writeback cache is enabled. In such configurations, writes from the user processes are first accumulated in the page cache and later `bdflush` threads wake up to flush dirty pages [8]. While flushing the pages FUSE forms asynchronous write requests and puts them in the background queue. Requests from the background queue gradually trickle to the pending queue. FUSE limits the number of asynchronous requests simultaneously residing in the pending queue to the configurable `max_background` parameter (12 by default). When fewer than 12 asynchronous requests are in the pending queue, requests from the background queue are moved to the pending queue. The intention is to limit the delay caused to important synchronous requests by bursts of background requests.

The queues' lengths are not explicitly limited. However, when the number of asynchronous requests in the pending and processing queues reaches the value of the tunable `congestion_threshold` parameter (75% of `max_background`, 9 by default), FUSE informs the Linux VFS that it is congested; the VFS then throttles the user processes that write to this file system.

Splicing and FUSE buffers. In its basic setup, the FUSE daemon has to `read()` requests from and `write()` replies to `/dev/fuse`. Every such call requires a memory copy between the kernel and user space. It is especially harmful for WRITE requests and READ replies because they often process a lot of data. To alleviate this problem, FUSE can use *splicing* functionality provided by the Linux kernel [38]. Splicing allows the user space to transfer data between two in-kernel memory buffers without copying the data to user space. This is useful, e.g., for stackable file systems that pass data directly to the underlying file system.

To seamlessly support splicing, FUSE represents its buffers in one of two forms: (1) a regular memory region identified by a pointer in the user daemon's address space, or (2) a kernel-space memory pointed by a file descriptor. If a user-space file system implements the `write_buf()` method, then FUSE splices the data from `/dev/fuse` and passes the data directly to this method in a form of the buffer containing a file descriptor. FUSE splices WRITE requests that contain more than a single page of data. Similar logic applies to replies to READ requests with more than two pages of data.

Multithreading. FUSE added multithreading support as parallelism got more popular. In multi-threaded mode, FUSE's daemon starts with one thread. If there are two or more requests available in the pending queue, FUSE automatically spawns additional threads. Every thread processes one request at a time. After processing the request, each thread checks if there are more than 10 threads; if so, that thread exits. There is no explicit upper limit on the number of threads created by the FUSE library. An implicit limit exists for two reasons: (1) by default, only 12 asynchronous requests (`max_background` parameter) can be in the pending queue at one time; and (2) the number of synchronous requests in the pending queue depends on the total amount of I/O activity generated by user processes. In addition, for every INTERRUPT and FORGET requests, a new thread is invoked. In a typical system where there is no interrupts support and few FORGETs are generated, the total number of FUSE daemon threads is at most $(12 + number\ of\ requests\ in\ pending\ queue)$.

Write back cache and max writes. The basic write behavior of FUSE is synchronous and only 4KB of data is sent to the user daemon for writing. This results in performance problems on certain workloads; when copying a large file into a FUSE file system, `/bin/cp` indirectly causes every 4KB of data to be sent to userspace synchronously. The solution FUSE implemented was to make FUSE's page cache support a write-back policy and then make writes asynchronous. With that change, file data can be pushed to the user daemon in larger chunks of `max_write` size (limited to 32 pages).

3 Instrumentation

To study FUSE's performance, we developed a simple stackable passthrough file system—called *Stackfs*—and instrumented FUSE's kernel module and user-space library to collect useful statistics and traces. We believe that the instrumentation presented here is useful for anyone who develops a FUSE-based file system.

3.1 Stackfs

Stackfs is a file system that passes FUSE requests unmodified directly to the underlying file system. The reason for Stackfs was twofold. (1) After examining the code of all publicly available [28, 43] FUSE-based file systems, we found that most of them are stackable (i.e., deployed on top of other, often in-kernel file systems). (2) We wanted to add as little overhead as possible, to isolate the overhead of FUSE's kernel and library.

Complex production file systems often need a high degree of flexibility, and thus use FUSE's low-level API. As such file systems are our primary focus, we implemented Stackfs using FUSE's low-level API. This also avoided the overheads added by the high-level API. Below we describe several important data structures and procedures that Stackfs uses.

Inode. Stackfs stores per-file metadata in an inode. Stackfs's inode is not persistent and exists in memory only while the file system is mounted. Apart from bookkeeping information, the inode stores the path to the underlying file, its inode number, and a reference counter.

The path is used, e.g., to open the underlying file when an OPEN request for a Stackfs file arrives.

Lookup. During lookup, Stackfs uses `stat(2)` to check if the underlying file exists. Every time a file is found, Stackfs allocates a new inode and returns the required information to the kernel. Stackfs assigns its inode the number equal to the address of the inode structure in memory (by typecasting), which is guaranteed to be unique. This allows Stackfs to quickly find the inode structure for any operations following the lookup (e.g., open or stat). The same inode can be looked up several times (e.g., due to hardlinks) and therefore Stackfs stores inodes in a hash table indexed by the underlying inode number. When handling LOOKUP, Stackfs checks the hash table to see whether the inode was previously allocated and, if found, increases its reference counter by one. When a FORGET request arrives for an inode, Stackfs decreases inode's reference count and deallocates the inode when the count drops to zero.

File create and open. During file creation, Stackfs adds a new inode to the hash table after the corresponding file was successfully created in the underlying file system. While processing OPEN requests, Stackfs saves the file descriptor of the underlying file in the file handle. The file descriptor is then used during read and write operations and deallocated when the file is closed.

3.2 Performance Statistics and Traces

The existing FUSE instrumentation was insufficient for in-depth FUSE performance analysis. We therefore instrumented FUSE to export important runtime statistics. Specifically, we were interested in recording the duration of time that FUSE spends in various stages of request processing, both in kernel and user space.

We introduced a two-dimensional array where a row index (0–42) represents the request type and the column index (0–31) represents the time. Every cell in the array stores the number of requests of a corresponding type that were processed within the 2^{N+1}–2^{N+2} nanoseconds where N is the column index. The time dimension therefore covers the interval of up to 8 seconds which is enough in typical FUSE setups. (This technique efficiently records a log_2 latency histogram [18].) We then added four such arrays to FUSE: the first three arrays are in the kernel, capturing the time spent by the request inside the background, pending, and processing queues. For the processing queue, the captured time also includes the time spent by requests in user space. The fourth array is in user space and tracks the time the daemon needs to process a request. The total memory size of all four arrays is only 48KiB and only few instructions are necessary to update values in the array.

FUSE includes a special `fusectl` file system to allow users to control several aspects of FUSE's behavior. This file system is usually mounted at `/sys/fs/fuse/connections/` and creates a directory for every mounted FUSE instance. Every directory contains control files to abort a connection, check the total number of requests being processed, and adjust the upper limit and the threshold on the number of background requests (see Section 2.2). We added 3 new files to these directories to export statistics from the in-kernel arrays. To export user-level array we added `SIGUSR1` signal handler to the daemon. When triggered, the handler prints the array to a log file specified during the daemon's start. The statistics captured have no measurable overhead on FUSE's performance and are the primary source of information about FUSE's performance.

Tracing. To understand FUSE's behavior in more detail we sometimes needed more information and had to resort to tracing. FUSE's library already performs tracing when the daemon runs in debug mode but there is no tracing support for FUSE's kernel module. We used Linux's static tracepoint mechanism [10] to add over 30 tracepoints mainly to monitor the formation of requests during the complex writeback logic, reads, and some metadata operations. Tracing helped us learn how fast queues grow during our experiments, how much data is put into a single request, and why.

Both FUSE's statistics and tracing can be used by any existing and future FUSE-based file systems. The instrumentation is completely transparent and requires no changes to file-system-specific code.

4 Methodology

FUSE has evolved significantly over the years and added several useful optimizations: writeback cache, zero-copy via splicing, and multi-threading. In our personal experience, some in the storage community tend to pre-judge FUSE's performance—assuming it is poor—mainly due to not having information about the improvements FUSE made over the years. We therefore designed our methodology to evaluate and demonstrate how FUSE's performance advanced from its basic configurations to ones that include all of the latest optimizations. We now detail our methodology, starting from the description of FUSE configurations, proceed to the list of workloads, and finally present our testbed.

FUSE configurations. To demonstrate the evolution of FUSE's performance, we picked two configurations on opposite ends of the spectrum: the *basic* configuration (called *StackfsBase*) with no major FUSE optimizations and the *optimized* configuration (called *StackfsOpt*) that enables all FUSE improvements available as of this writing. Compared to StackfsBase, the StackfsOpt configuration adds the following features: (1) writeback cache is turned on; (2) maximum size of a single FUSE request is increased from 4KiB to

128KiB (max_write parameter); (3) user daemon runs in the multi-threaded mode; (4) splicing is activated for all operations (splice_read, splice_write, and splice_move parameters). We left all other parameters at their default values in both configurations.

Workloads. To stress different modes of FUSE operation and conduct an thorough performance characterization, we selected a broad set of workloads: micro and macro, metadata- and data-intensive, and also experimented with a wide range of I/O sizes and parallelism levels. Table 2 describes all workloads that we employed. To simplify the identification of workloads in the text we use the following mnemonics: rnd stands for random, seq for sequential, rd for reads, wr for writes, cr for creates, and del for deletes. The presence of *N*th and *M*f substrings in a workload name means that the workload contains *N* threads and *M* files, respectively. We fixed the amount of work (e.g., the number of reads in rd workloads) rather than the amount of time in every experiment. We find it easier to analyze performance in experiments with a fixed amount of work. We picked a sufficient amount of work so that the performance stabilized. Resulting runtimes varied between 8 and 20 minutes across the experiments. Because SSDs are orders of magnitude faster than HDDs, for some workloads we selected a larger amount of work for our SSD-based experiments. We used Filebench [12, 37] to generate all workloads.

Experimental setup. FUSE performance depends heavily on the speed of the underlying storage: faster devices expose FUSE's own overheads. We therefore experimented with two common storage devices of different speed: an HDD (Seagate Savvio 15K.2, 15KRPM, 146GB) and an SSD (Intel X25-M SSD, 200GB). Both devices were installed in three identical Dell PowerEdge R710 machines with 4-core Intel Xeon E5530 2.40GHz CPU each. The amount of RAM available to the OS was set to 4GB to accelerate cache warmup in our experiments. The machines ran CentOS 7 with Linux kernel upgraded to v4.1.13 and FUSE library commit #386b1b.

We used Ext4 [11] as the underlying file system because it is common, stable, and has a well documented design which facilitates performance analysis. Before every experiment we reformatted the storage devices with Ext4 and remounted the file systems. To lower the variability in our experiments we disabled Ext4's lazy inode initialization [5]. In either case, standard deviations in our experiments were less than 2% for all workloads except for three: seq-rd-1th-1f (6%), files-rd-32th (7%), and mail-server (7%).

5 Evaluation

For many, FUSE is just a practical tool to build real products or prototypes, but not a research focus. To present our results more effectively, we split the evaluation in two. Section 5.1 overviews our extensive evaluation results—most useful information for many practitioners. Detailed performance analysis follows in Section 5.2.

5.1 Performance Overview

To evaluate FUSE's performance degradation, we first measured the throughput (in ops/sec) achieved by native Ext4 and then measured the same for Stackfs deployed over Ext4. As detailed in Section 4 we used two configurations of Stackfs: a basic (*StackfsBase*) and optimized (*StackfsOpt*) one. From here on, we use Stackfs to refer to both of these configurations. We then calculated the relative performance degradation (or improvement) of Stackfs vs. Ext4 for each workload. Table 3 shows absolute throughputs for Ext4 and relative performance for two Stackfs configurations for both HDD and SSD.

For better clarity we categorized the results by Stackfs's performance difference into four classes: (1) The *Green* class (marked with $^{+}$) indicates that the performance either degraded by less than 5% or actually improved; (2) The *Yellow* class (*) includes results with the performance degradation in the 5–25% range; (3) The *Orange* class ($^{\#}$) indicates that the performance degradation is between 25–50%; And finally, (4) the *Red* class ($^{!}$) is for when performance decreased by more than 50%. Although the ranges for acceptable performance degradation depend on the specific deployment and the value of other benefits provided by FUSE, our classification gives a broad overview of FUSE's performance. Below we list our main observations that characterize the results. We start from the general trends and move to more specific results towards the end of the list.

Observation 1. The relative difference varied across workloads, devices, and FUSE configurations from –83.1% for files-cr-1th [row #37] to +6.2% for web-server [row #45].

Observation 2. For many workloads, FUSE's optimizations improve performance significantly. E.g., for the web-server workload, StackfsOpt improves performance by 6.2% while StackfsBase degrades it by more than 50% [row #45].

Observation 3. Although optimizations increase the performance of some workloads, they can degrade the performance of others. E.g., StackfsOpt decreases performance by 35% more than StackfsBase for the files-rd-1th workload on SSD [row #39].

Observation 4. In the best performing configuration of Stackfs (among StackfsOpt and StackfsBase) only two file-create workloads (out of a total 45 workloads) fell into the red class: files-cr-1th [row #37] and files-cr-32th [row #38].

Workload Name	Description	Results
`seq-rd-Nth-1f`	*N* threads (1, 32) sequentially read from a single preallocated 60GB file.	[rows #1–8]
`seq-rd-32th-32f`	32 threads sequentially read 32 preallocated 2GB files. Each thread reads its own file.	[rows #9–12]
`rnd-rd-Nth-1f`	*N* threads (1, 32) randomly read from a single preallocated 60GB file.	[rows #13–20]
`seq-wr-1th-1f`	Single thread creates and sequentially writes a new 60GB file.	[rows #21–24]
`seq-wr-32th-32f`	32 threads sequentially write 32 new 2GB files. Each thread writes its own file.	[rows #25–28]
`rnd-wr-Nth-1f`	*N* threads (1, 32) randomly write to a single preallocated 60GB file.	[rows #29–36]
`files-cr-Nth`	*N* threads (1, 32) create 4 million 4KB files over many directories.	[rows #37–38]
`files-rd-Nth`	*N* threads (1, 32) read from 1 million preallocated 4KB files over many directories.	[rows #39–40]
`files-del-Nth`	*N* threads (1, 32) delete 4 million of preallocated 4KB files over many directories.	[rows #41–42]
`file-server`	File-server workload emulated by Filebench. Scaled up to 200,000 files.	[row #43]
`mail-server`	Mail-server workload emulated by Filebench. Scaled up to 1.5 million files.	[row #44]
`web-server`	Web-server workload emulated by Filebench. Scaled up to 1.25 million files.	[row #45]

Table 2: Description of workloads and their corresponding result rows. For data-intensive workloads, we experimented with 4KB, 32KB, 128KB, and 1MB I/O sizes. We picked dataset sizes so that both cached and non-cached data are exercised. The Results column correlates these descriptions with results in Table 3.

Observation 5. Stackfs's performance depends significantly on the underlying device. E.g., for sequential read workloads [rows #1–12], Stackfs shows no performance degradation for SSD and a 26–42% degradation for HDD. The situation is reversed, e.g., when a `mail-server` [row #44] workload is used.

Observation 6. At least in one Stackfs configuration, all write workloads (sequential and random) [rows #21–36] are within the *Green* class for both HDD and SSD.

Observation 7. The performance of sequential read [rows #1–12] are well within the *Green* class for both HDD and SSD; however, for the `seq-rd-32th-32f` workload [rows #5–8] on HDD, they are in *Orange* class. Random read workload results [rows #13–20] span all four classes. Furthermore, the performance grows as I/O sizes increase for both HDD and SSD.

Observation 8. In general, Stackfs performs visibly worse for metadata-intensive and macro workloads [rows #37–45] than for data-intensive workloads [rows #1–36]. The performance is especially low for SSDs.

Observation 9. The relative CPU utilization of Stackfs (not shown in the Table) is higher than that of Ext4 and varies in the range of +0.13% to +31.2%; similarly, CPU cycles per operation increased by 1.2× to 10× times between Ext4 and Stackfs (in both configurations). This behavior is seen in both HDD and SSD.

Observation 10. CPU cycles per operation are higher for StackfsOpt than for StackfsBase for the majority of workloads. But for the workloads `seq-wr-32th-32f` [rows #25–28] and `rnd-wr-1th-1f` [rows #30–32], StackfsOpt consumes fewer CPU cycles per operation.

5.2 Analysis

We analyzed FUSE performance results and present main findings here, following the order in Table 3.

5.2.1 Read Workloads

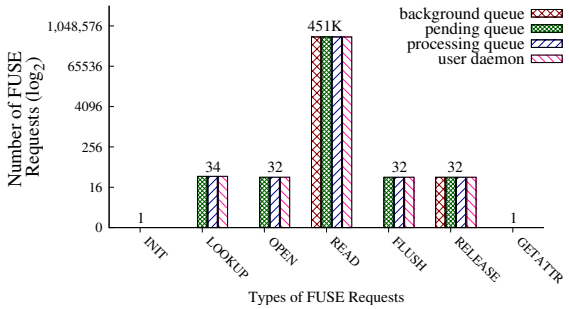


Figure 3: Different types and number of requests generated by StackfsBase on SSD during the `seq-rd-32th-32f` workload, from left to right, in their order of generation.

Figure 3 demonstrates the types of requests that were generated with the `seq-rd-32th-32f` workload. We use `seq-rd-32th-32f` as a reference for the figure because this workload has more requests per operation type compared to other workloads. Bars are ordered from left to right by the appearance of requests in the experiment. The same request types, but in different quantities, were generated by the other read-intensive workloads [rows #1–20]. For the single threaded read workloads, only one request per LOOKUP, OPEN, FLUSH, and RELEASE type was generated. The number of READ requests depended on the I/O size and the amount of data read; INIT request is produced at mount time so its count remained the same across all workloads; and finally GETATTR is invoked before unmount for the root directory and was the same for all the workloads.

Figure 3 also shows the breakdown of requests by queues. By default, READ, RELEASE, and INIT are asynchronous; they are added to the background queue first. All other requests are synchronous and are added to pending queue directly. In read workloads, only READ requests are generated in large numbers. Thus, we discuss in detail only READ requests for these workloads.

#	Workload	I/O Size (KB)	HDD Results			SSD Results		
			EXT4 (ops/sec)	StackfsBase (%Diff)	StackfsOpt (%Diff)	EXT4 (ops/sec)	StackfsBase (%Diff)	StackfsOpt (%Diff)
1	seq-rd-1th-1f	4	38382	- 2.45^{+}	+ 1.7^{+}	30694	- 0.5^{+}	- 0.9^{+}
2		32	4805	- 0.2^{+}	- 2.2^{+}	3811	+ 0.8^{+}	+ 0.3^{+}
3		128	1199	- 0.86^{+}	- 2.1^{+}	950	+ 0.4^{+}	+ 1.7^{+}
4		1024	150	- 0.9^{+}	- 2.2^{+}	119	+ 0.2^{+}	- 0.3^{+}
5	seq-rd-32th-1f	4	1228400	- 2.4^{+}	- 3.0^{+}	973450	+ 0.02^{+}	+ 2.1^{+}
6		32	153480	- 2.4^{+}	- 4.1^{+}	121410	+ 0.7^{+}	+ 2.2^{+}
7		128	38443	- 2.6^{+}	- 4.4^{+}	30338	+ 1.5^{+}	+ 1.97^{+}
8		1024	4805	- 2.5^{+}	- 4.0^{+}	3814.50	- 0.1^{+}	- 0.4^{+}
9	seq-rd-32th-32f	4	11141	- 36.9$^{\#}$	- 26.9$^{\#}$	32855	- 0.1^{+}	- 0.16^{+}
10		32	1491	- 41.5$^{\#}$	- 30.3$^{\#}$	4202	- 0.1^{+}	- 1.8^{+}
11		128	371	- 41.3$^{\#}$	- 29.8$^{\#}$	1051	- 0.1^{+}	- 0.2^{+}
12		1024	46	- 41.0$^{\#}$	- 28.3$^{\#}$	131	- 0.03^{+}	- 2.1^{+}
13	rnd-rd-1th-1f	4	243	- 9.96*	- 9.95*	4712	- 32.1$^{\#}$	- 39.8$^{\#}$
14		32	232	- 7.4*	- 7.5*	2032	- 18.8*	- 25.2$^{\#}$
15		128	191	- 7.4*	- 5.5*	852	- 14.7*	- 12.4*
16		1024	88	- 9.0*	-3.1^{+}	114	- 15.3*	-1.5^{+}
17	rnd-rd-32th-1f	4	572	- 60.4$^{!}$	-23.2*	24998	- 82.5$^{!}$	-27.6$^{\#}$
18		32	504	- 56.2$^{!}$	-17.2*	4273	- 55.7$^{!}$	-1.9^{+}
19		128	278	- 34.4$^{\#}$	-11.4*	1123	- 29.1$^{\#}$	-2.6^{+}
20		1024	41	- 37.0$^{\#}$	-15.0*	126	- 12.2*	-1.9^{+}
21	seq-wr-1th-1f	4	36919	-26.2$^{\#}$	- 0.1^{+}	32959	- 9.0*	+ 0.1^{+}
22		32	4615	- 17.8*	- 0.16^{+}	4119	- 2.5^{+}	+ 0.12^{+}
23		128	1153	- 16.6*	- 0.15^{+}	1030	- 2.1^{+}	+ 0.1^{+}
24		1024	144	- 17.7*	-0.31^{+}	129	- 2.3^{+}	- 0.08^{+}
25	seq-wr-32th-32f	4	34370	- 2.5^{+}	+ 0.1^{+}	32921	+ 0.05^{+}	+ 0.2^{+}
26		32	4296	- 2.7^{+}	+ 0.0^{+}	4115	+ 0.1^{+}	+ 0.1^{+}
27		128	1075	- 2.6^{+}	- 0.02^{+}	1029	- 0.04^{+}	+ 0.2^{+}
28		1024	134	- 2.4^{+}	- 0.18^{+}	129	- 0.1^{+}	+ 0.2^{+}
29	rnd-wr-1th-1f	4	1074	- 0.7^{+}	- 1.3^{+}	16066	+ 0.9^{+}	- 27.0$^{\#}$
30		32	708	- 0.1^{+}	- 1.3^{+}	4102	- 2.2^{+}	- 13.0*
31		128	359	- 0.1^{+}	- 1.3^{+}	1045	- 1.7^{+}	- 0.7^{+}
32		1024	79	- 0.01^{+}	- 0.8^{+}	129	- 0.02^{+}	- 0.3^{+}
33	rnd-wr-32th-1f	4	1073	- 0.9^{+}	- 1.8^{+}	16213	- 0.7^{+}	- 26.6$^{\#}$
34		32	705	+ 0.1^{+}	- 0.7^{+}	4103	- 2.2^{+}	- 13.0*
35		128	358	+ 0.3^{+}	- 1.1^{+}	1031	- 0.1^{+}	+ 0.03^{+}
36		1024	79	+ 0.1^{+}	- 0.3^{+}	128	+ 0.9^{+}	- 0.3^{+}
37	files-cr-1th	4	30211	- 57$^{!}$	- 81.0$^{!}$	35361	- 62.2$^{!}$	- 83.3$^{!}$
38	files-cr-32th	4	36590	- 50.2$^{!}$	- 54.9$^{!}$	46688	- 57.6$^{!}$	- 62.6$^{!}$
39	files-rd-1th	4	645	+ 0.0^{+}	- 10.6*	8055	- 25.0*	- 60.3$^{!}$
40	files-rd-32th	4	1263	- 50.5$^{!}$	-4.5^{+}	25341	- 74.1$^{!}$	-33.0$^{\#}$
41	files-del-1th	-	1105	- 4.0^{+}	- 10.2*	7391	- 31.6$^{\#}$	- 60.7$^{!}$
42	files-del-32th	-	1109	- 2.8^{+}	- 6.9*	8563	- 42.9$^{\#}$	- 52.6$^{!}$
43	file-server	-	1705	- 26.3$^{\#}$	-1.4^{+}	5201	- 41.2$^{\#}$	-1.5^{+}
44	mail-server	-	1547	- 45.0$^{\#}$	-4.6^{+}	11806	- 70.5$^{!}$	-32.5$^{\#}$
45	web-server	-	1704	- 51.8$^{!}$	+6.2^{+}	19437	- 72.9$^{!}$	-17.3*

Table 3: List of workloads and corresponding performance results. Green *class (marked with $^{+}$) indicates that the performance either degraded by less than 5% or actually improved;* Yellow *class (*) includes results with the performance degradation in the 5–25% range;* Orange *class ($^{\#}$) indicates that the performance degradation is between 25–50%; And finally, the* Red *class ($^{!}$) is for when performance decreased by more than 50%.*

Sequential Read using 1 thread on 1 file. The total number of READ requests that StackfsBase generated during the whole experiment for different I/O sizes for HDD and SSD remained approximately the same and equal to 491K. Our analysis revealed that this happens because of FUSE's default 128KB-size readahead which effectively levels FUSE request sizes no matter what is the user application I/O size. Thanks to readahead, sequential read performance of StackfsBase and StackfsOpt was as good as Ext4 for both HDD and SSD.

Sequential Read using 32 threads on 32 files. Due to readahead, the total number of READ requests generated here was also approximately same for different I/O sizes. At any given time, 32 threads are requesting data and continuously add requests to queues. StackfsBase and StackfsOpt show significantly larger performance degradation on HDD compared to SSD. For StackfsBase, the user daemon is single threaded and the device is slower, so requests do not move quickly through the queues. On the faster SSD, however, even though the user daemon is single threaded, requests move faster in the queues. Hence performance of StackfsBase is as close to that of Ext4. With StackfsOpt, the user daemon is multi-threaded and can fill the HDD's queue faster so performance improved for HDD compared to SSD.

Investigating further, we found that for HDD and StackfsOpt, FUSE's daemon was bound by the *max_background* value (default is 12): at most, only 12 user deamons (threads) were spawned. We increased that limit to 100 and reran the experiments: now StackfsOpt was within 2% of Ext4's performance.

Sequential Read using 32 threads on 1 file. This workload exhibits similar performance trends to `seq-rd-1th-1f`. However, because all 32 user threads read from the same file, they benefit from the shared page cache. As a result, instead of 32× more FUSE requests, we saw only up to a 37% increase in number of requests. This modest increase is because, in the beginning of the experiment, every thread tries to read the data separately; but after a certain point in time, only a single thread's requests are propagated to the user daemon while all other threads' requests are available in the page cache. Also, having 32 user threads running left less CPU time available for FUSE's threads to execute, thus causing a slight (up to 4.4%) decrease in performance compared to Ext4.

Random Read using 1 thread on 1 file. Unlike the case of small sequential reads, small random reads did not benefit from FUSE's readahead. Thus, every application read call was forwarded to the user daemon which resulted in an overhead of up to 10% for HDD and 40% for SSD. The absolute Ext4 throughput is about 20× higher for SSD than for HDD which explains the higher penalty on FUSE's relative performance on SSD.

The smaller the I/O size is, the more READ requests are generated and the higher FUSE's overhead tended to be. This is seen for StackfsOpt where performance for HDD gradually grows from –10.0% for 4KB to –3% for 1MB I/O sizes. A similar situation is seen for SSD. Thanks to splice, StackfsOpt performs better than StackfsBase for large I/O sizes. For 1MB I/O size, the improvement is 6% on HDD and 14% on SSD. Interestingly, 4KB I/O sizes have the highest overhead because FUSE splices requests only if they are larger than 4KB.

Random Read using 32 threads on 1 file. Similar to the previous experiment (single thread random read), readahead does not help smaller I/O sizes here: every user read call is sent to the user daemon and causes high performance degradation: up to –83% for StackfsBase and –28% for StackfsOpt. The overhead caused by StackfsBase is high in these experiments (up to –60% for HDD and –83% for SSD), for both HDD and SSD, and especially for smaller I/O sizes. This is because when 32 user threads submit a READ request, 31 of those threads need to wait while the single-threaded user daemon processes one request at a time. StackfsOpt reduced performance degradation compared to StackfsBase, but not as much for 4KB I/Os because splice is not used for request that are smaller or equal to 4KB.

5.2.2 Write Workloads

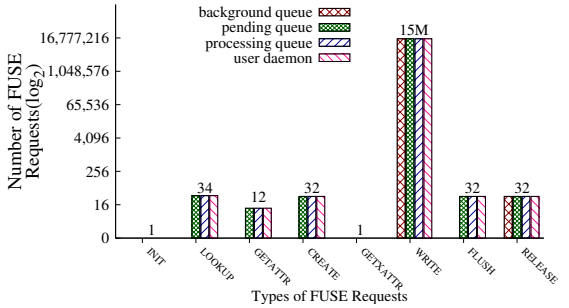


Figure 4: Different types of requests that were generated by StackfsBase on SSD for the `seq-wr-32th-32f` workload, from left to right in their order of generation.

We now discuss the behavior of StackfsBase and StackfsOpt in all write workloads listed in Table 3 [rows #21–36]. Figure 4 shows the different types of requests that got generated during all write workloads, from left to right in their order of generation (`seq-wr-32th-32f` is used as a reference). In case of `rnd-wr` workloads, CREATE requests are replaced by OPEN requests, as random writes operate on pre-allocated files. For all the `seq-wr` workloads, due to the creation of files, a GETATTR request was generated to check permissions of the single directory where the files were created. Linux VFS caches attributes and therefore there were fewer than 32 GETATTRs. For single-threaded

Stages of `write()` call processing	Time (μs)	Time (%)
Processing by VFS before passing execution to FUSE kernel code	1.4	2.4
FUSE request allocation and initialization	3.4	6.0
Waiting in queues and copying to user space	10.7	18.9
Processing by Stackfs daemon, includes Ext4 execution	24.6	43.4
Processing reply by FUSE kernel code	13.3	23.5
Processing by VFS after FUSE kernel code	3.3	5.8
Total	56.7	100.0

Table 4: Average latencies of a single write request generated by StackfsBase during seq-wr-4KB-1th-1f workload across multiple profile points on HDD.

workloads, five operations generated only one request: LOOKUP, OPEN, CREATE, FLUSH, and RELEASE; however, the number of WRITE requests was orders of magnitude higher and depended on the amount of data written. Therefore, we consider only WRITE requests when we discuss each workload in detail.

Usually the Linux VFS generates GETXATTR before every write operation. But in our case StackfsBase and StackfsOpt did not support extended attributes and the kernel cached this knowledge after FUSE returned ENOSUPPORT for the first GETXATTR.

Sequential Write using 1 thread on 1 file. The total number of WRITE requests that StackfsBase generated during this experiment was 15.7M for all I/O sizes. This is because in StackfsBase each user write call is split into several 4KB-size FUSE requests which are sent to the user daemon. As a result StackfsBase degraded performance ranged from –26% to –9%. Compared to StackfsBase, StackfsOpt generated significantly fewer FUSE requests: between 500K and 563K depending on the I/O size. The reason is the writeback cache that allows FUSE's kernel part to pack several dirty pages (up to 128KB in total) into a single WRITE request. Approximately $\frac{1}{32}$ of requests were generated in StackfsOpt compared to StackfsBase. This suggests indeed that each WRITE request transferred about 128KB of data (or 32× more than 4KB).

Table 4 shows the breakdown of time spent (latencies) by a single write request across various stages, during the `seq-wr-4KB-1th-1f` workload on HDD. Taking only major latencies, the write request spends 19% of its time in request creation and waiting in the kernel queues; 43% of its time in user space, which includes time taken by the underlying Ext4 to serve the write; and then 23% of time during copy of the response from user space to kernel. The relative CPU utilization caused by StackfsBase and StackfsOpt in `seq-wr-4KB-1th-1f` on HDD is 6.8% and 11.1% more than native Ext4, respectively; CPU cycles per operation were the same for StackfsBase and StackfsOpt—4× that of native Ext4.

Sequential Write using 32 threads on 32 files. Performance trends are similar to `seq-wr-1th-1f` but even the unoptimized StackfsBase performed much better (up to –2.7% and –0.1% degradation for HDD and SSD, respectively). This is because without the writeback cache, 32 user threads put more requests into FUSE's queues (compared to 1 thread) and therefore kept the user daemon constantly busy.

Random Write using 1 thread on 1 file. Performance degradation caused by StackfsBase and StackfsOpt was low on HDD for all I/O sizes (max –1.3%) because the random write performance of Ext4 on HDD is low—between 79 and 1,074 Filebench ops/sec, depending on the I/O size (compare to over 16,000 ops/sec for SSD). The performance bottleneck, therefore, was in the HDD I/O time and FUSE overhead was invisible.

Interestingly, on SSD, StackfsOpt performance degradation was high (–27% for 4KB I/O) and more than the StackfsBase for 4KB and 32KB I/O sizes. The reason for this is that currently FUSE's writeback cache batches only *sequential* writes into a single WRITE. Therefore, in the case of *random* writes there is no reduction in the number of WRITE requests compared to StackfsBase. These numerous requests are processed asynchronously (i.e., added to the background queue). And because of FUSE's congestion threshold on the background queue the application that is writing the data becomes throttled.

For I/O size of 32KB, StackfsOpt can pack the entire 32KB into a single WRITE request. Compared to StackfsBase, this reduces the number of WRITE requests by 8× and results in 15% better performance.

Random Write using 32 threads on 1 file. This workload performs similarly to `rnd-wr-1th-1f` and the same analysis applies.

5.2.3 Metadata Workloads

We now discuss the behavior of Stackfs in all metadata micro-workloads as listed in Table 3 [rows #37–42].

File creates. Different types of requests that got generated during the `files-cr-Nth` runs are GETATTR, LOOKUP, CREATE, WRITE, FLUSH, RELEASE, and FORGET. The total number of each request type generated was exactly 4 million. Many GETATTR requests were generated due to Filebench calling a `fstat` on the file to check whether it exists or not before creating it. `Files-cr-Nth` workloads demonstrated the worst performance among all workloads for both StackfsBase and StackfsOpt and for both HDD and SSD. The reason is twofold. First, for every single file create, five operations happened serially: GETATTR, LOOKUP, CREATE, WRITE, and FLUSH; and as there were many files accessed, they all could not be cached, so we saw many FORGET requests to remove cached items—which added further overhead. Second, file creates are fairly fast in

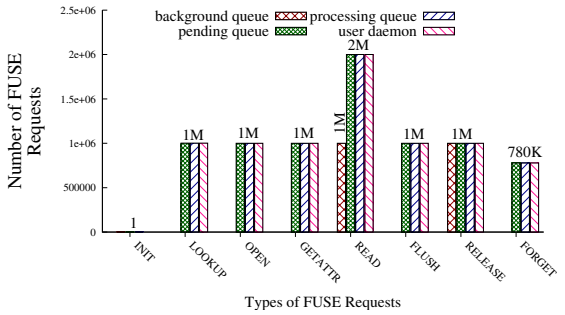


Figure 5: Different types of requests that were generated by StackfsBase on SSD for the `files-rd-1th` workload, from left to right in their order of generation.

Ext4 (30–46 thousand creates/sec) because small newly created inodes can be effectively cached in RAM. Thus, overheads caused by the FUSE's user-kernel communications explain the performance degradation.

File Reads. Figure 5 shows different types of requests that got generated during the `files-rd-1th` workload. This workload is metadata-intensive because it contains many small files (one million 4KB files) that are repeatedly opened and closed. Figure 5 shows that half of the READ requests went to the background queue and the rest directly to the pending queue. The reason is that when reading a whole file, and the application requests reads beyond the EOF, FUSE generates a synchronous READ request which goes to the pending queue (not the background queue). Reads past the EOF also generate a GETATTR request to confirm the file's size.

The performance degradation for `files-rd-1th` in StackfsBase on HDD is negligible; on SSD, however, the relative degradation is high (–25%) because the SSD is 12.5× faster than HDD (see Ext4 absolute throughput in Table 3). Interestingly, StackfsOpt's performance degradation is more than that of StackfsBase (by 10% and 35% for HDD and SSD, respectively). The reason is that in StackfsOpt, *different* FUSE threads process requests for the *same* file, which requires additional synchronization and context switches. Conversely, but as expected, for `files-rd-32th` workload, StackfsOpt performed 40–45% better than StackfsBase because multiple threads are needed to effectively process parallel READ requests.

File Deletes. The different types of operations that got generated during the `files-del-1th` workloads are LOOKUP, UNLINK, FORGET (exactly 4 million each). Every UNLINK request is followed by FORGET. Therefore, for every incoming delete request that the application (Filebench) submits, StackfsBase and StackfsOpt generates three requests (LOOKUP, UNLINK, and FORGET) in series, which depend on each other.

Deletes translate to small random writes at the block layer and therefore Ext4 benefited from using an SSD (7–8× higher throughput than the HDD). This negatively impacted Stackfs in terms of relative numbers: its performance degradation was 25–50% higher on SSD than on HDD. In all cases StackfsOpt's performance degradation is more than StackfsBase's because neither splice nor the writeback cache helped `files-del-Nth` workloads and only added additional overhead for managing extra threads.

5.2.4 Macro Server Workloads

We now discuss the behavior of Stackfs for macro-workloads [rows #43–45].

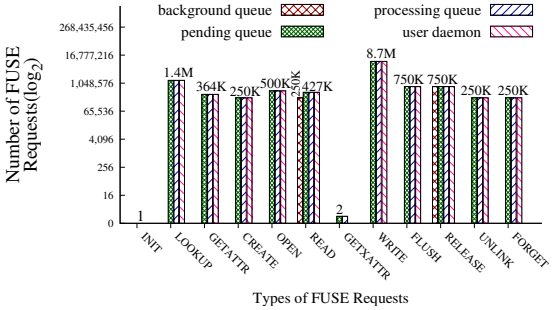


Figure 6: Different types of requests that were generated by StackfsBase on SSD for the `file-server` workload.

File Server. Figure 6 shows different types of operations that got generated during the `file-server` workload. Macro workloads are expected to have a more diverse request profile than micro workloads, and `file-server` confirms this: many different requests got generated, with WRITEs being the majority.

The performance improved by 25–40% (depending on storage device) with StackfsOpt compared to StackfsBase, and got close to Ext4's native performance for three reasons: (1) with a writeback cache and 128KB requests, the number of WRITEs decreased by a factor of 17× for both HDD and SSD, (2) with splice, READ and WRITE requests took advantage of zero copy, and (3) the user daemon is multi-threaded, as the workload is.

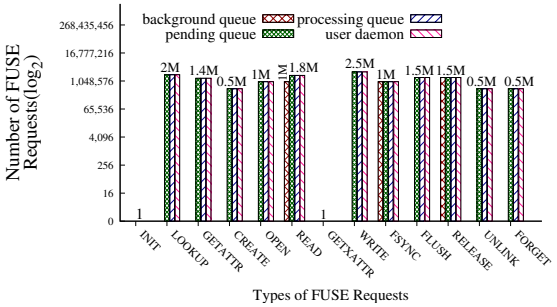


Figure 7: Different types of requests that were generated by StackfsBase on SSD for the `mail-server` workload.

Mail Server. Figure 7 shows different types of operations that got generated during the `mail-server` workload. As with the `file-server` workload, many

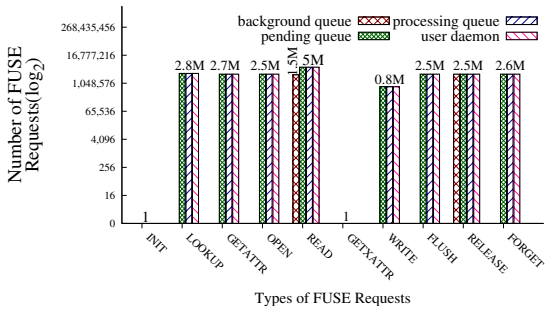


Figure 8: Different types of requests that were generated by StackfsBase on SSD for the `web-server` *workload.*

different requests got generated, with WRITEs being the majority. Performance trends are also similar between these two workloads. However, in the SSD setup, even the optimized StackfsOpt still did not perform close to Ext4 in this `mail-server` workload, compared to `file-server`. The reason is twofold. First, compared to file server, mail server has almost double the metadata operations, which increases FUSE overhead. Second, I/O sizes are smaller in mail-server which improves the underlying Ext4 SSD performance and therefore shifts the bottleneck to FUSE.

Web Server. Figure 8 shows different types of requests generated during the `web-server` workload. This workload is highly read-intensive as expected from a Web-server that services static Web-pages. The performance degradation caused by StackfsBase falls into the *Red* class in both HDD and SSD. The major bottleneck was due to the FUSE daemon being single-threaded, while the workload itself contained 100 user threads. Performance improved with StackfsOpt significantly on both HDD and SSD, mainly thanks to using multiple threads. In fact, StackfsOpt performance on HDD is even 6% *higher* than of native Ext4. We believe this minor improvement is caused by the Linux VFS treating Stackfs and Ext4 as two independent file systems and allowing them together to cache more data compared to when Ext4 is used alone, without Stackfs. This does not help SSD setup as much due to the high speed of SSD.

6 Related Work

Many researchers used FUSE to implement file systems [3, 9, 15, 40] but little attention was given to understanding FUSE's underlying design and performance. To the best of our knowledge, only two papers studied some aspects of FUSE. First, Rajgarhia and Gehani evaluated FUSE performance with Java bindings [27]. Compared to this work, they focused on evaluating Java library wrappers, used only three workloads, and ran experiments with FUSE v2.8.0-pre1 (released in 2008). The version they used did not support zero-copying via splice, writeback caching, and other important features. The authors also presented only limited information about FUSE design at the time.

Second, in a position paper, Tarasov et al. characterized FUSE performance for a variety of workloads but did not analyze the results [36]. Furthermore, they evaluated only default FUSE configuration and discussed only FUSE's high-level architecture. In this paper we evaluated and analyzed several FUSE configurations in detail, and described FUSE's low-level architecture.

Several researchers designed and implemented useful extensions to FUSE. Re-FUSE automatically restarts FUSE file systems that crash [33]. To improve FUSE performance, Narayan et al. proposed to marry in-kernel stackable file systems [44] with FUSE [23]. Shun et al. modified FUSE's kernel module to allow applications to access storage devices directly [17]. These improvements were in research prototypes and were never included in the mainline.

7 Conclusion

User-space file systems are popular for prototyping new ideas and developing complex production file systems that are difficult to maintain in kernel. Although many researchers and companies rely on user-space file systems, little attention was given to understanding the performance implications of moving file systems to user space. In this paper we first presented the detailed design of FUSE, the most popular user-space file system framework. We then conducted a broad performance characterization of FUSE and we present an in-depth analysis of FUSE performance patterns. We found that for many workloads, an optimized FUSE can perform within 5% of native Ext4. However, some workloads are unfriendly to FUSE and even if optimized, FUSE degrades their performance by up to 83%. Also, in terms of the CPU utilization, the relative increase seen is 31%.

All of our code and Filebench workloads files are available from *http://filesystems.org/fuse/*.

Future work. There is a large room for improvement in FUSE performance. We plan to add support for compound FUSE requests and investigate the possibility of shared memory between kernel and user spaces for faster communications.

Acknowledgments

We thank the anonymous FAST reviewers and our shepherd Tudor Marian for their valuable comments. This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; NSF awards CNS-1251137, CNS-1302246, CNS-1305360, and CNS-1622832; and ONR award 12055763.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer USENIX Technical Conference*, pages 93–112, Atlanta, GA, June 1986. USENIX Association.

[2] The Apache Foundation. Hadoop, January 2010. *http://hadoop.apache.org*.

[3] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. Plfs: A checkpoint filesystem for parallel applications. Technical Report LA-UR 09-02117, LANL, April 2009. *http://institute.lanl.gov/plfs/*.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813, Network Working Group, June 1995.

[5] Z. Cao, V. Tarasov, H. Raman, D. Hildebrand, and E. Zadok. On the performance variation in modern storage stacks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February/March 2017. USENIX Association. to appear.

[6] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings to the First Dutch International Symposium on Linux*, Seattle, WA, December 1994.

[7] Michael Condict, Don Bolinger, Dave Mitchell, and Eamonn McManus. Microkernel Modularity with Integrated Kernel Performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI 1994)*, Monterey, CA, November 1994.

[8] Jonathan Corbet. In defense of per-bdi writeback, September 2009. *http://lwn.net/Articles/354851/*.

[9] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: A User-level Versioning File System for Linux. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 19–28, Boston, MA, June 2004. USENIX Association.

[10] Mathieu Desnoyers. Using the Linux kernel tracepoints, 2016. *https://www.kernel.org/doc/Documentation/trace/tracepoints.txt*.

[11] Ext4 Documentation. *https://www.kernel.org/doc/Documentation/filesystems/ext4.txt*.

[12] Filebench, 2016. *https://github.com/filebench/filebench/wiki*.

[13] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, NY, October 2003. ACM SIGOPS.

[14] Hermann Hartig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schonberg. The performance of Microkernel-based systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP '97)*, Saint Malo, France, October 1997. ACM.

[15] V. Henson, A. Ven, A. Gud, and Z. Brown. Chunkfs: Using Divide-and-Conquer to Improve File System Reliability and Repair. In *Proceedings of the Second Workshop on Hot Topics in System Dependability (HotDep 2006)*, Seattle, WA, November 2006. ACM SIGOPS.

[16] GNU Hurd. *www.gnu.org/software/hurd/hurd.html*.

[17] Shun Ishiguro, Jun Murakami, Yoshihiro Oyama, and Osamu Tatebe. Optimizing local file accesses for FUSE-based distributed storage. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*, pages 760–765. IEEE, 2012.

[18] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating System Profiling via Latency Analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 89–102, Seattle, WA, November 2006. ACM SIGOPS.

[19] Lessfs, January 2012. *www.lessfs.com*.

[20] Linus Torvalds doesn't understand user-space filesystems. *http://redhatstorage.redhat.com/2011/06/28/linus-torvalds-doesnt-understand-user-space-storage/*.

[21] David Mazieres. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001.

[22] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[23] Sumit Narayan, Rohit K Mehta, and John A Chandy. User space storage system stack modules with file level control. In *Proceedings of the 12th Annual Linux Symposium in Ottawa*, pages 189–196, 2010.

[24] Nimble's Hybrid Storage Architecture. *http://info.nimblestorage.com/rs/nimblestorage/images/nimblestorage_technology_overview.pdf*.

[25] NTFS-3G. *www.tuxera.com*.

[26] David Pease, Arnon Amir, Lucas Villa Real, Brian Biskeborn, Michael Richmond, and Atsushi Abe. The linear tape file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[27] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *25th Symposium On Applied Computing*. ACM, March 2010.

[28] Nikolaus Rath. List of fuse based file systems (git page), 2011. *https://github.com/libfuse/libfuse/wiki/Filesystems*.

[29] Glusterfs. *http://www.gluster.org/*.

[30] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, pages 231–244, Monterey, CA, January 2002. USENIX Association.

[31] Opendedup, January 2012. *www.opendedup.org*.

[32] D. Steere, J. Kistler, and M. Satyanarayanan. Efficient user-level file cache management on the sun vnode interface. In *Proceedings of the Summer USENIX Technical Conference*, Anaheim, CA, June 1990. IEEE.

[33] Swaminathan Sundararaman, Laxman Visampalli, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Refuse to crash with Re-FUSE. In *Proceedings of the sixth conference on Computer systems*, pages 77–90. ACM, 2011.

[34] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 1–14, San Diego, CA, January 1996.

[35] M. Szeredi. Filesystem in Userspace. *http://fuse.sourceforge.net*, February 2005.

[36] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok. Terra incognita: On the practicality of user-space file systems. In *HotStorage '15: Proceedings of the 7th USENIX Workshop on Hot Topics in Storage*, Santa Clara, CA, July 2015.

[37] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[38] L. Torvalds. *splice()*. Kernel Trap, 2007. *http://kerneltrap.org/node/6505*.

[39] Linux Torwalds. Re: [patch 0/7] overlay filesystem: request for inclusion. *https://lkml.org/lkml/2011/6/9/462*.

[40] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: a High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *Proceedings of the FAST Conference*, 2010.

[41] Sage Weil. Linus vs fuse. *http://ceph.com/dev-notes/linus-vs-fuse/*.

[42] Assar Westerlund and Johan Danielsson. Arla-a free AFS client. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, New Orleans, LA, June 1998. USENIX Association.

[43] Felix Wiemann. List of fuse based file systems (wiki page), 2006. *https://en.wikipedia.org/wiki/Filesystem_in_Userspace#Example_uses*.

[44] E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000. USENIX Association.

[45] ZFS for Linux, January 2016. *www.zfs-fuse.net*.

[46] 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)*, San Jose, California, USA, 2008.

Knockoff: Cheap versions in the cloud

Xianzheng Dou, Peter M. Chen, and Jason Flinn
University of Michigan

Abstract

Cloud-based storage provides reliability and ease-of-management. Unfortunately, it can also incur significant costs for both storing and communicating data, even after using techniques such as chunk-based deduplication and delta compression. The current trend of providing access to past versions of data exacerbates both costs.

In this paper, we show that deterministic recomputation of data can substantially reduce the cost of cloud storage. Borrowing a well-known dualism from the fault-tolerance community, we note that any data can be equivalently represented by a log of the nondeterministic inputs needed to produce that data. We design a file system, called Knockoff, that selectively substitutes nondeterministic inputs for file data to reduce communication and storage costs. Knockoff compresses both data and computation logs: it uses chunk-based deduplication for file data and delta compression for logs of nondeterminism. In two studies, Knockoff reduces the average cost of sending files to the cloud without versioning by 21% and 24%; the relative benefit increases as versions are retained more frequently.

1 Introduction

Two trends in storage systems are conspiring to increase the cost of storing and retrieving data. First, due to compelling ease-of-management, cost-effectiveness, and reliability benefits, businesses and consumers are storing more of their data at cloud-based storage providers. However, writing and reading data from remote sites can incur significant costs for network communication.

Second, customers are increasingly expecting and depending on the ability to access multiple versions of their data. Local storage solutions such as Apple's Time Machine [3] retain multiple versions of users' data and make it easy to access this data. Cloud storage providers have followed suit; for example, Google Drive [18], Microsoft OneDrive [25], and DropBox [14] all store and allow users to access old versions of their files.

Past versions have many uses; e.g., recovery of lost or overwritten data, reproduction of the process by which data was created, auditing, and forensic troubleshooting. The benefit of versioning increases as more versions are retained. For instance, if versions are retained every time a file is closed, the user is usually guaranteed a snapshot of file data with each save operation or when the application terminates. However, many applications save data only on termination; in such cases, all intermediate data created during application usage are unavailable for recovery and analysis. Saving data on every file system modification produces more frequent checkpoints, but cannot recover transient state in memory that never is written to the file system and, importantly, does not capture modifications to memory-mapped files. In the extreme, a user should be able to reproduce any past state in the file system or in application memory, a property we call *eidetic* versioning.

The cost of versioning also depends on the frequency at which versions are retained. For instance, retaining a version on every file modification incurs greater storage cost than retaining a version on every close, and the client will consume more bandwidth by sending a greater number of versions to cloud storage. Versioning policies must balance these benefits and costs. Many current systems choose infrequent versioning as a result.

In this paper, we seek to substantially reduce the cost of communicating file data between clients and servers; we also seek to reduce the cost of keeping multiple versions of data. Our work reduces resource usage and user costs for existing versioning policies. It also enables finer-grained versioning, e.g. eidetic versioning, that is infeasible in current distributed storage architectures.

To accomplish these goals, we leverage an unconventional method for communicating and storing file data. In lieu of the actual file data, we selectively represent a file as a log of the nondeterminism needed to recom-

pute the data (e.g., system call results, thread scheduling, and external data read by a process). With such a log, a file server can deterministically replay the computation to recreate the data in the file. Representing state as a log of nondeterminism is well known in the fault-tolerance community [16]; however, logs of nondeterminism are often quite large, and applying this idea requires that the logs for a computation be smaller than the output files produced. To address this problem, we apply recent ideas for reducing the size of individual logs [13], and we also use delta compression to reduce the collective size of logs of similar executions.

Representing data as a log of nondeterminism leads to several benefits in a distributed file system. First, it substitutes (re-)computation for communication and storage, and this can reduce total cost because computation in cloud systems is less costly than communication and storage. Second, it can reduce the number of bytes sent over the network when the log of nondeterminism is smaller than the data produced by the recorded computation. For the same reason, it can reduce the number of bytes stored by the cloud storage provider. Finally, representing data as a log of nondeterminism can support a wider range of versioning frequencies than prior methods.

This paper describes the design and implementation of a distributed file system, Knockoff, that selectively replaces file data with a log of the nondeterminism needed to produce that data for both communication with cloud servers and storage in the cloud. Knockoff supports several frequencies of versioning: no versioning at all, version on file close, version on every write system call, and version on every store instruction (for mapped files).

The contributions of this paper are:

- We provide the first general-purpose solution for operation shipping in a distributed file system by leveraging deterministic record and replay.
- We show how compression can be applied to computation as well as to storage by using delta compression to reduce the size of the logs of nondeterminism that represent such computation.
- We quantify the costs and benefits of general-purpose operation shipping in a distributed file system over actual file system usage.

We evaluate Knockoff by performing a multi-user study for a software development scenario and a 20-day, single-user longitudinal study. Without versioning, Knockoff reduced the average cost of sending files to the cloud in these studies by 24% and 21%, respectively. The benefit of using Knockoff increases as versions are retained with greater frequency. The cost of this approach is the performance overhead of recording executions (7-8% in our evaluation) and a greater delay in retrieving past versions (up to 60 seconds for our default settings).

2 Background and related work

Knockoff is based on the principle that one can represent data generated by computation either by value or by the log of inputs needed to reproduce the computation. We call this the principle of equivalence (between values and computation); it has been observed and used in many settings; e.g., fault tolerance [16], state machine replication [37], data center storage management [20], and state synchronization [19].

The projects most related to ours use the principle of equivalence for the same purpose, namely to reduce communication overhead between clients and servers in a distributed file system. Lee et al. first applied this principle in the Coda File System [22, 23] and coined the term *operation shipping*. Clients log and send user operations (e.g., shell commands) to a server surrogate that replays the operations to regenerate the data. Chang et al. extend this idea to log and send user activity, such as keyboard and mouse inputs [10].

Although the basic idea of operation shipping is powerful, prior system logged and shipped very restricted types of nondeterminism and thus could not guarantee that the state received through the log matched the original state. Neither a log of shell commands nor a log of user activity are sufficient to reproduce the computation of general-purpose programs. Researchers recognized this shortcoming and mitigated it by supplementing the replayed computation with forward error correction and compensating actions, using hashes to detect remaining differences and revert back to value shipping. Unfortunately, the shift to multiprocessors and multithreaded programs means that many programs are nondeterministic in ways not handled in these prior systems. Further, because these prior systems handled a very limited set of nondeterministic inputs, they required identical environments on the recording and replaying side, which is unrealistic in many client-server settings.

Knockoff applies the same basic principle of equivalence, but it uses a comprehensive log of nondeterminism to provide equivalence for all race-free programs (and many programs with occasional data races). This enables Knockoff to use operation shipping in more settings, and it also makes possible the first realistic evaluation of operation shipping for such settings (earlier studies unrealistically assumed that programs were deterministic). Knockoff also applies operation shipping to versioning file systems. We find that the gains of operation shipping are larger when multiple versions are saved, and indispensable when versions are saved at eidetic granularity.

Adams et al. identify recomputation as a way to reduce storage [1], but do not implement or evaluate any system based on this observation. Other systems use recomputation to reduce storage in restricted environments

	Log entry	Values
1	open	rc=3
2	mmap	file=<id,version>
3	pthread_lock	
4	open	rc=4
5	read	rc=<size>, file=<id,version>
6	gettimeofday	rc=0, time=<timestamp>
7	open	rc=5
8	write	rc=<size>
9	pthread_unlock	

Figure 1: Sample log of nondeterminism

in which the computation is guaranteed to be deterministic. Nectar [20] applies this idea to DryadLINQ applications, which are both deterministic and functional. BAD-FS [6] uses re-computation in lieu of replicating data; users must specify explicit dependencies and the computation must be deterministic to produce the same data.

Besides reproducing data, logging has been used in file systems to track the provenance of files [29, 41] and guide when new versions should be saved [28]. More generally, redo logging [26] provides transactional properties in the presence of failures.

Many prior systems deduplicate file data to reduce communication and storage [12, 21, 31, 38, 40, 43]. LBFS [31] uses chunk-based deduplication in which Rabin fingerprinting divides a file into chunks, a hash value is computed for each chunk, and a client and server use the hash values to avoid communicating chunks already seen by the other party. Knockoff uses LBFS-style deduplication when transferring data by value.

Versioning file systems [24, 30, 36, 39, 44] retain past state at a specific granularity such as on every file close or on every modification. Cloud storage providers such as DropBox [14] and Google Drive [18] currently allow users to retain past versions of file data. Knockoff makes versioning file systems more efficient by reducing storage and computation costs. It also supports versioning at finer granularities than these prior systems.

3 Motivating example

We start with a motivating example that illustrates why a log of nondeterminism for an execution may require significantly less storage than the data produced by the execution. Consider a simple application that reads in a data file, computes a statistical transformation over that data, and writes a timestamped summary to an output file. The output data may be many megabytes in size. However, the program itself can be reproduced given a small log of determinism, as shown in Figure 1 (for clarity, the log has been simplified).

The log records the results of system calls (e.g., `open`) and synchronization operation (e.g., `pthread_lock`). The first entry in Figure 1 records the file descriptor chosen by the operating system during the original execution. Parameters to the open call do not need to be logged since they will be reproduced during a deterministic re-execution. The second entry records the mapping of the executable; replaying this entry will cause the exact version used during recording to be mapped to the same place in the replaying process address space. Lines 4 and 5 read data from the input file, line 6 records the original timestamp, and lines 7 and 8 write the transformation to the output file. Note that data read from the file system is not in the log since Knockoff can reproduce the desired version on demand. Also, the data written to the output file need not be logged since it will be reproduced exactly as a result of replaying the execution.

With compression, a log for this sample application can be only a few hundred bytes in size, as contrasted with the megabytes of data that the execution produces. The output data is reproduced by starting from the same initial state, re-executing the computation, and supplying values from the log for each nondeterministic operation.

4 Design considerations

Recent work on deterministic replay [2, 13, 15, 32, 35, 42] now makes it possible to use operation shipping to build a general-purpose distributed file system for realistic environments and workloads. Our goals are to build such a system, identify synergies between operation shipping and versioning file systems, and demonstrate how operation shipping can reduce communication and storage costs for realistic workloads.

4.1 Deterministic record and replay

To use operation shipping for realistic workloads and environments, we need a general-purpose deterministic record and replay system. The record/replay system should support unmodified applications and work for multithreaded programs. To work in realistic client/server configurations, the record/replay system should allow recorded executions to be replayed in environments that differ from the one on which they were recorded. Finally, to enable operation shipping to be used for some (but not all) processes, the system should record each application individually and allow each to be replayed individually on the server.

Knockoff uses the Arnold system [13], which meets these requirements. Arnold uses a modified Linux kernel to record the execution of Linux processes. It records all nondeterministic data that enters a process, including the results of system calls (such as user and network input), the timing of signals, and real-time clock queries. Because it supplies recorded values on replay rather than

re-executing system calls that interact with external dependencies, Arnold can trivially record an application on one computer and replay it on another. The only requirements are that both computers run the Arnold kernel and have the same processor architecture (x86).

Arnold enables deterministic replay of multi-threaded programs by recording all synchronization operations (e.g., `pthread_lock` and atomic hardware instructions). Arnold can detect programs with data races, but it does not guarantee that the replay of such programs will match their recorded execution. Arnold does guarantee that a replay is always repeatable (i.e., deterministic with respect to other replays), even for racy programs.

4.2 Files: values or operations?

Knockoff can represent a file in one of two ways: as normal file data (by value) or as a log of the nondeterminism needed to recreate the file (by operation). Which of these representations is more cost-effective depends on the characteristics of the program that generated the file data, as well as the relative costs of computation, communication, and storage. Files that are large and generated by programs that are mostly deterministic (e.g., photo-editing software) are best represented by operation. In contrast, files that are small and generated by programs that use a lot of nondeterministic data (e.g,. cryptographic key generation) are best represented by value.

At any time, Knockoff can use deterministic replay to convert a file that is represented by operation into a representation by value (but not vice versa). To do so, Knockoff loads and re-executes the original program, then feeds in the log of nondeterminism that was recorded in the original execution. Note that file data read by system calls from Knockoff are *not* included in the log. Instead, these log entries refer to the file and version that was read, and Arnold's replay system reads this data from Knockoff. Usually, application binaries, dynamic libraries, configuration files, and the like are stored in Knockoff, so the server replay sees the same application files as existed on the client during the original recording. If a binary or library is not stored in Knockoff, the file data are included by value in the log of nondeterminism and provided on replay. Replay of previously recorded applications may require retention of past file versions. Alternatively, we can regenerate these past versions through additional recursive replays of the applications that produced the data.

Whenever Knockoff represents a file by operation, it must first verify that Arnold's replay faithfully reconstructs the file data because Arnold does not guarantee that programs with data races replay exactly as recorded. Knockoff uses a SHA-512 hash for each file to verify that the replay correctly generated the original data. Because replay in Arnold is repeatable, a run that produces matching data in the first replay is guaranteed to produce matching data in all subsequent replays. If the replay does not produce matching data, Knockoff switches to representing the file by value.

Knockoff chooses between these two representations when it ships files between clients and servers and when it stores files on the server. To guide its choice, Knockoff measures the computation time used to create each file and the size of each file.

5 Implementation

Knockoff is a client-server distributed file system in which the server is hosted in the cloud. The server stores the current version of all files, and it optionally stores past versions of all files according to a user-selected versioning policy. Knockoff clients have a local disk cache that stores current and (optionally) past file versions.

Knockoff implements Coda-style weak file consistency [27]. Clients propagate file system updates asynchronously to the server. Clients register a callback with the server when they cache the current version of a file, and the server breaks the callback by sending a client a message when another client modifies the file.

Knockoff associates a version vector [33] with each file to identify specific versions and detect conflicting updates. Knockoff assigns clients a unique identifier; every time a client performs a system call that modifies a file (e.g., `write`), it increments an integer in the version vector associated with its identifier. Thus, every past version of a file has a unique version vector that can be used to name and retrieve that version. The server detects conflicting updates by comparing the version vector for each update and determining that neither one dominates the other. If a conflict occurs, the server retains both versions, and the user manually resolves the conflict.

Knockoff clients record almost all user-level process executions (excluding some servers such as the X server and sshd) and the kernel generates a log of nondeterminism for each such execution. Logs of nondeterminism are stored in a log cache on the client and may also be sent to the server and stored in a database there. The server has a replay engine that allows it to regenerate file data from such logs.

5.1 Versioning

Knockoff supports versioning policies on a per-file-system basis. Users select one of the following:

- **No versioning.** Knockoff retains only the current version of all files. For durability, a client sends a modified file to the server on `close`. After the first close, Knockoff waits up to 10 seconds to send the file modifications to the server (this delay allows coalescing of multiple updates that occur closely

together in time [27]). Upon receiving file modifications, the server overwrites the previous version of the file and breaks any callbacks held by other clients. The server retains multiple versions only in the case of conflicting updates.

- **Version on close.** Knockoff retains all past versions at close granularity; for past versions, Knockoff may store the actual data or the logs required to regenerate the data. On receiving a file modification, the server retains the previous version instead of overwriting it. Clients may ask for a version by specifying its unique version vector.
- **Version on write.** Knockoff retains all past versions at write granularity. Every system call that modifies a file creates a new version, and Knockoff can reproduce all such versions.
- **Eidetic.** Knockoff retains all past versions at instruction granularity. It can reproduce any computation or file data and determine the provenance of data via Arnold. The server stores all application logs. Clients may ask for a specific version of a file by specifying a version vector and an instruction count that specifies when to stop the replay (so as to recover a specific state for a mapped file).

5.2 Architecture

Figure 2 shows Knockoff's storage architecture. The client runs a FUSE[17] user-level file system daemon.

5.2.1 Clients

The Knockoff client stores file system data in four persistent caches to hide network latency. Whole file versions are stored in the *version cache*; this cache may hold multiple versions of the same file simultaneously. Each file in Knockoff is given a unique integer *fileid*, so a particular version of a file can be retrieved from the cache by specifying both a fileid and a version vector. The version cache tracks which versions it stores are the current version of a file. It sets callbacks for such entries; if the server breaks a callback (because another client has updated the file), the version is retained in the cache, but its designation as the current version is removed.

The *chunk cache* stores the chunks generated by chunk-based deduplication for each file in the version cache; thus the version cache contains only pointers to chunks in the chunk cache. Knockoff divides each entry in the database into chunks using the LBFS chunk-based deduplication algorithm [31] and calculates the SHA-512 hash of each such chunk. The chunk database is indexed by these hash values.

Directory data is stored in a separate Berkeley DB [7] *directory cache*. Knockoff clients implement an in-memory index over this data to speed up path lookups. The *log cache* stores logs of nondeterminism generated by recording application execution.

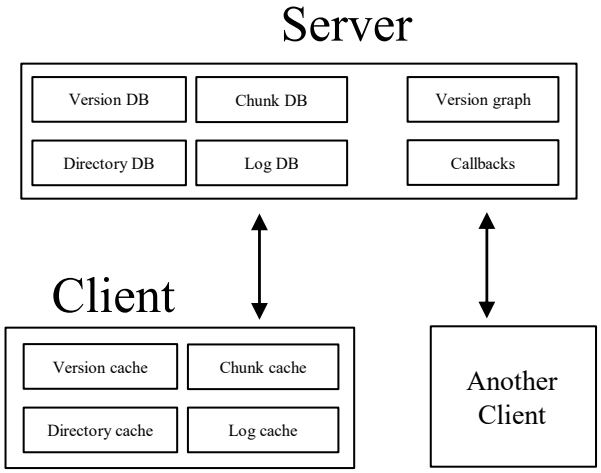


Figure 2: Architecture overview

All client caches are managed via LRU eviction, with preference given to retaining current versions over past versions. Chunks are removed from the chunk cache when they are no longer used by any version in the version cache. Modified values are pinned in the caches until they have been persisted to the server.

5.2.2 Server

The server maintains analogs of these client stores. The server's *version DB* stores the current version of every file and, depending on the the versioning policy, past file versions. The *chunk DB* stores chunk data for all files in the version DB, indexed by the chunk SHA-512 hash values. The *directory DB* stores all directory information for Knockoff, and the *log DB* stores logs of nondeterminism recorded by clients.

If the versioning policy is `eidetic`, the log DB stores every application log recorded by any Knockoff client. If the versioning policy is `version on write`, every past file version is either stored in the version DB, or the logs necessary to reproduce that version are stored in the log DB. If the versioning policy is `version on close`, this invariant holds only for versions that correspond to file closes. If the versioning policy is `no versioning`, only the current versions of file and directory data are stored.

The server also maintains callbacks set by clients for each file. If a client updates a file, the server uses these to determine which clients to notify of the update.

Finally, the server maintains the *version graph* which relates file versions with the computation that produced the data. Nodes in the graph are either recorded logs of nondeterminism (representing a particular execution) or file versions. Each edge represents a range of file data that was written by one recording and either read by another recording or part of a file version. An index allows Knockoff to quickly find a particular file version by fileid and version vector. If a version is not present in the ver-

sion DB, the version graph allows Knockoff to determine which logs need to be replayed to regenerate the data.

5.3 Writing data

Knockoff is designed to reduce cloud storage costs. A large part of these costs is communication. For example, AT&T [4] and Comcast [11] currently charge up to $10 for every 50 GB of data communication, whereas Amazon currently charges $0.052 per hour for an instance with 2 vCPUs and 4 GB of memory [5]. This means that Knockoff can reduce costs if it can use 1 second of cloud computation to save 76 KB of communication.

We first describe how Knockoff handles writes for its `no versioning` policy, and then generalize to other policies. Like Arnold, Knockoff records application execution in a *replay group* that logs the nondeterminism of related processes. When an application closes a file, the Knockoff client checks the file's version vector to determine whether it was modified. If so, Knockoff starts a 10-second timer that allows the application to make further modifications. When the timer expires, Knockoff starts a transaction in which all outstanding modifications to any file by that application are sent to the server. If the application terminates, Knockoff starts the transaction immediately.

To propagate modifications to the server, Knockoff first calculates the cost of sending and replaying the log of nondeterminism given a pre-defined cost of communication ($cost_{comm}$) and computation ($cost_{comp}$):

$$cost_{log} = size_{log} * cost_{comm} + time_{replay} * cost_{comp} \quad (1)$$

$size_{log}$ is determined by compressing the log of nondeterminism for the application that wrote the file and measuring its size directly. Because Knockoff does not currently support checkpointing, each log must be replayed from application start.

To estimate $time_{replay}$, Knockoff modifies Arnold to store the user CPU time consumed so far by the recorded application with each log entry that modifies file data. This is a very good estimate for the time needed to replay the log on the client [34]. To estimate server replay time, Arnold multiplies this value by a conversion factor to reflect the relative CPU speeds of the client and server.

Knockoff calculates the cost of sending file data as:

$$cost_{data} = size_{chunks} * cost_{comm} \quad (2)$$

Knockoff implements the chunk-based deduplication algorithm used by LBFS to reduce the cost of transmitting file data. It breaks all modified files into chunks, hashes each chunk, and sends the hashes to the server. The server responds with the set of hashes it has stored. $size_{chunks}$ is the size of any chunks unknown to the server that would need to be transmitted; Knockoff uses gzip compression to reduce bytes transmitted for such chunks.

If $cost_{log} < cost_{data}$, Knockoff sends the log to the server. The server makes a copy of the modified files and assigns them new version vectors. It then spawns a replay process that consumes the log and replays the application. When the replay process executes a system call that modifies a target file, it updates the new version in the cache directly. Once the replay reaches a file close operation, the new cache version is complete and marked as available to be read. The server deletes the old version for the `no versioning` policy. The replay process is paused at this point rather than terminated because the client may ship more log data to the server to regenerate additional file modifications made by the same application. The server only terminates a replay process when the client notifies it that the application that is being replayed has terminated.

In rare cases, more than one application may write to a file simultaneously. The Knockoff server replays these logs concurrently and ensures that writes are ordered correctly according to the version vectors.

Replay is guaranteed to produce the same data if the application being replayed is free of data races. Data-race freedom can be guaranteed for some programs (e.g., single-threaded ones) but not for complex applications. Knockoff therefore ships a SHA-512 hash of each modified file to the server with the log. The Knockoff server verifies this hash on close. If verification fails, it asks the client to ship the file data. Note that such races are rare with Arnold since the replay system itself acts as an efficient data-race detector [13].

If $cost_{data} < cost_{log}$, then Knockoff could reduce the cost of the current transaction by sending the unique chunks to the server. However, for long running applications, it may be the case that sending and replaying the log collected so far would help reduce the cost of future file modifications that have yet to be seen (because the cost of replaying from this point is less than replaying from the beginning of the program). Knockoff predicts this by looking at a history of $cost_{data}/cost_{log}$ ratios for the application. If sending logs has been historically beneficial and current application behavior is similar (the ratios differ by less than 40%) to past executions, it sends the log. Otherwise, it sends the unique data chunks.

For long running applications, Knockoff may use multiple transactions to send modified files to the server. If the client has previously chosen to ship the log to the server, $size_{log}$ is the portion of the log that excludes what has already been sent to the server, and $time_{replay}$ is the additional user CPU time consumed by the recorded application after the log prefix, scaled for the difference in client and server CPU speed.

Other versioning policies work similarly. For the `version on close` policy, Knockoff sends not only the current version of a modified file but also all versions that

existed at any file close during the period covered by the transaction. The cost of sending the log is identical to the `no versioning` case, but the previous versions may introduce new data chunks not yet seen by the server and thereby increase the cost of sending the file data.

For the `version on write` policy, the transaction includes all versions created by any system call that modified a file. Since chunk-based deduplication works better across larger modification sizes, Knockoff tries to coalesce modifications to the same file within a transaction as long as one modification does not overwrite data from a previous one. The transaction metadata describes how to reconstruct individual versions.

For the `eidetic` policy, Knockoff always sends the application log to the server since it is needed to reproduce past computations and mapped file state. The server usually updates the current version of files by replaying the log. However, if the computation cost of log replay is greater than the communication cost of fetching the modified file chunks from the server, the server asks the client for the file data instead of replaying the log.

5.4 Storing data

Knockoff may store file data on the server either by value (as normal file data) or by operation (as the log of nondeterminism required to recompute that data). If the log of nondeterminism is smaller than the file data it produces, then storing the file by operation saves space and money. However, storing files by operation delays future reads of that data, since Knockoff will need to replay the original computation that produced the data. In general, this implies that Knockoff should only store file data by operation if the data is very cold; i.e., if the probability of reading the data in the future is low.

Knockoff currently implements a simple policy to decide how to store data at the server. It always stores the current version of every file by value so that its read performance for current file data is the same as that of a traditional file system. Knockoff may store past versions by operation if the storage requirements for storing the data by log are less than those of storing the data by value. However, Knockoff also has a configuration parameter that sets a maximum *materialization delay*, which is the time to reconstruct any version stored by operation. The default materialization delay is 60 seconds.

For the `eidetic` policy, the materialization delay applies to all versions created via a file system call such as `write`. We could also apply this bound to intermediate file states for mapped files, but this would require us to implement checkpoints of application state so as to limit the time needed to produce intermediate states.

With the `eidetic` policy, Knockoff retains all logs of nondeterminism since they are needed to reproduce past computation and transient process state. This is sufficient to recompute any file version. However, the recomputation time is unbounded due to recursive dependencies. For instance, producing a past file version may require replaying a recorded computation. That computation may have read file data from Knockoff, so Knockoff must also reproduce those file versions via replay of other applications. This continues recursively until Knockoff encounters no more past versions to reproduce.

To limit the time to reproduce past versions, Knockoff selectively stores some past versions by value. It uses the server's version graph to decide which versions to store by value. The version graph shows the relationships between file versions and logs of nondeterminism. File versions and replay logs form the vertexes of the graph.

A version node in the graph contains a list of all the file byte ranges in that version that were written by distinct system calls. For each range, the node stores the log and the specific system call within the log that wrote the data. Replaying all such logs up to the specified system calls would be sufficient to recompute that particular version. However, recomputation is not necessary for byte ranges that already exist in the version DB. All byte ranges are present if the version represented by the node is itself in the version DB. Otherwise, a particular byte range may still be in the version DB because another version of the same file is stored by value and the range was not overwritten between the two versions.

Knockoff inserts an edge from a version node to a log node if the log contains a system call that wrote a byte range not in the version DB. Each edge has weight equal to the time to recompute all such byte ranges.

A log node contains similar information for each of its system calls that read data from a Knockoff file. For each read, it lists all file byte ranges that were written by distinct system calls; this contains the log and system call of the writing application. Replaying these logs would be sufficient to recompute all file system data read by the log node in question. Knockoff inserts an edge from one log node to another if the latter log wrote at least one byte range read by the former log that is not currently available in the version DB. As above, the weight of each edge is the predicted time to recompute all such byte ranges. If there is a cycle in the graph, two or more logs must be replayed concurrently to regenerate file data; Knockoff coalesces these cycles into a single node.

The time to recompute a version is given by the longest path rooted at the graph node for that version. Calculating the longest path for each version requires visiting each node at most once. If any path exceeds the specified materialization delay, Knockoff replays the latest log in the path, regenerates its file versions, and stores them by value. It repeats this process until no paths exceed the materialization delay. This greedy algorithm works well in our evaluation; if warranted in the future,

we could draw on more complex algorithms [8] to minimize storage costs while not exceeding the maximum materialization delay.

Currently, Knockoff recalculates the version graph and runs the above algorithm nightly. Note that file modifications and queries of past state between batch updates may have created new versions of past files in the version DB. These new versions are temporarily excluded from the batch computation. If Knockoff determines that they are not needed, they are removed from the version DB to save space. Otherwise, they are retained instead of recomputing them from logs of nondeterminism.

The `version on write` and `version on close` policies store data in similar fashion. The major difference is that these policies can discard logs to save storage space. Thus, for any log, if the size of the data produced by that log is less than the size of the log, Knockoff replays the log (if necessary) to recompute the data, then deletes the log. Discarded logs are removed from the version graph and the file versions produced by those logs are pinned in the version DB (they can never be deleted without violating the versioning policy since it is no longer possible to recompute their data).

5.5 Reading data

By default, any application that reads data from Knockoff receives the current version of the file. The client first checks its version cache to see if it has stored the current version of the file locally. If the version is present, Knockoff reads the requested data from the cache. It also appends a record to the reading application's log of nondeterminism that specifies the fileid, the version vector, and the logid and system call of the application that wrote the data it just read. The latter two values are obtained from Arnold's filemap [13].

If the version is not present, Knockoff fetches it from the server and caches it. Knockoff caches whole file versions, and clients fetch versions from the server by value. A client sends a request that specifies the fileid. The server responds with the current version vector and a list of hashes for each chunk comprising the file. The server also sends the filemap for the version. The client specifies which chunks it does not have cached, and the server sends that data to the client. The server sets a callback on the file. The client inserts the version into its version cache, marks it as the current version, and places the version's chunks into its chunk cache.

Applications may also read past versions of files by specifying a version vector. If a requested version is in the server's version DB, it is shipped by value as above. If it is not present, it must be recomputed by replaying one or more logs. We next describe this process.

In its version graph, the server maintains an index over all versions; this allows it to quickly find the particular version node being requested. The version node reveals the distinct byte ranges that were written by different system calls. If a range is in the version DB, it is used directly. Otherwise, the server must replay the log of the application that wrote the range to regenerate the data. For each such log, it determines the longest prefix that has to be replayed; this is the last system call in the log that wrote any range being read. Knockoff examines each such log prefix to determine if replaying the log requires file data that is not in the version cache. If so, it recursively visits the log(s) that wrote the needed data. Note that Knockoff's materialization delay bounds the amount of computation needed to produce any version. Knockoff then replays the visited logs to regenerate the desired version. It places this version in its version database and ships it to the client as described above.

5.6 Optimization: Log compression

While implementing Knockoff, we saw the effectiveness of chunk-based deduplication in reducing communication and storage costs. This led us to wonder: can we apply the same compression techniques to logs of nondeterminism that current file systems apply to file data?

Intuitively, log compression should be able to identify similar regions of nondeterministic data across executions of the same application. For example, application startup regions should be very similar because the application will open and load the same libraries, send similar messages to the X server to initialize windows, open similar configuration files, and so on.

We first attempted to apply chunk-based deduplication directly to log data. This worked reasonably well. However, after examining the logs we generated in more detail, we realized that the similarities between logs are often different from similarities between files. Similar files tend to have large contiguous chunks that are the same, whereas similar logs often lack such regions. Instead, most of the bytes within two log regions might be the same, but there exist in each region a smattering of values such as timestamps that differ. So, even very similar log chunks hash to different values.

Therefore, we turned to delta encoding. Knockoff first identifies a *reference log* that it expects to be similar to the current log. It then generates a binary patch via xdelta [45] that encodes the difference between the current log and the reference log. Given both the reference log and the patch, Knockoff can reconstruct the original values in the log.

When an application generates a log, the client and server identify a reference log. The client queries the log cache to find all prior logs for the same executable that it has stored locally. For each log, the log cache stores the arguments to the application, the size of the nondeterministic data, the running time of the application, and the

user-level CPU time. The client orders the cached logs by similarity across these metrics; if the application has not yet completed execution by the time the log is shipped, only the arguments are used to determine similarity since the other parameters are not yet known. Arguments are compared using cosine string similarity.

The client orders logs by similarity and sends the list to the server. The server responds with the most similar log that it has stored in its log DB. This step is omitted for the `eidetic` policy since the server stores all logs. The client then generates an xdelta patch and uses its size as $size_{log}$ in the algorithm described in Section 5.3.

When the server receives a compressed log, it stores it in compressed form in the log DB. It also adds a dependency on the reference log. Before a reference log can be pruned for the `version on close` or `version on write` policies, the server must first uncompress any log that depends on that log. The server uses the delta size when deciding whether to retain the log or the data in these policies. It currently does not take into account the cost of uncompressing logs when a reference log is purged because it assumes that logs can be recompressed effectively using different reference logs.

6 Evaluation

Our evaluation answers the following questions:

- How much does Knockoff reduce bandwidth usage compared to current cloud storage solutions?
- How much does Knockoff reduce communication and storage costs?
- What is Knockoff's performance overhead?
- How effective is log compression?

6.1 Experimental setup

All experiments for measuring communication and storage costs were run on two virtual machines (one for the client and one for the server). Both virtual machines were hosted on computers with a 4 core Intel i7-3770 CPU, 16GB memory, and two 7200 RPM hard drives. For accuracy, performance results were measured with a physical machine as the client with a 4 core Xeon E5620 processor, 6 GB memory, and a 7200 RPM hard drive. All platforms run the Ubuntu 12.04LTS Linux kernel.

Due to a lack of representative file system benchmarks that also include the computation to generate the data, we use two methods to evaluate Knockoff. First, we study users performing a software development task to measure how Knockoff benefits different people. Second, we measure Knockoff while an author of this paper runs the system on the author's primary computer for 20 days. This allows us to study the storage costs of multiple versions generated over a longer time period.

	20-day study	User study
Disk read (MB)	5473	2583
Disk write (MB)	6706	4339
File open count	261523	418594
Number of executions	3803	1146
Number of programs	75	63

Table 1: Workload characteristics

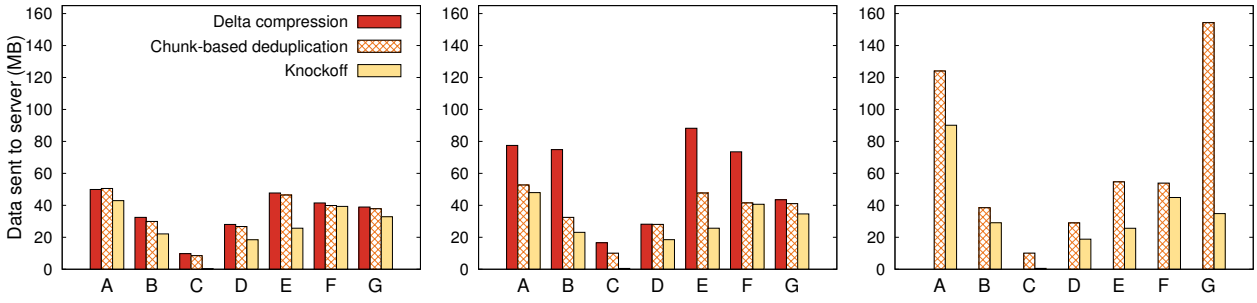
During these studies, we use Knockoff's `eidetic` policy, which allows us to regenerate all file system reads and writes by replaying Arnold logs. We use these logs to measure the bandwidth and storage costs of running Knockoff over the same workload with other policies.

We implement two baseline file systems for comparison. The first uses the LBFS algorithm for chunk-based deduplication to implement all versioning policies except `eidetic`; this is representative of current cloud storage solutions such as DropBox. The second uses delta compression to implement `no versioning` and `version on close`; this is representative of git [9] and other version control systems. Delta compression performed poorly for `version on write` because our implementation did not detect when bytes were inserted in the middle of a file; we therefore omit these results.

6.2 User study

We recruited 8 graduate students to study Knockoff for a software development workload. We asked them to write software to perform several simple tasks, e.g., converting a CSV file to a JSON file; each participant could spend up to an hour solving the problem. We did not dictate how the problem should be solved. Participants used various Linux utilities, text editors, IDEs, and programming languages. They used Web browsers to visit different Web sites such as Google and StackOverflow, as well as sites unrelated to the assignment (e.g., Facebook and CNN News). One of the 8 participants was unable to complete the programming assignment and quit right away. We show results for the 7 participants who attempted the tasks; 4 of these finished successfully within the hour. The second column of Table 1 shows aggregate characteristics of this recorded workload.

Figure 4 summarizes the results by aggregating the bytes sent to the server by Knockoff and the baseline file systems across all 7 users; this represents approximately 7 hours of software development activity. Although we are targeting versioning file systems, Knockoff is surprisingly effective in reducing bytes sent over the network for non-versioning file systems. Compared to chunk-based deduplication, Knockoff reduces communication by 24%. Compared to delta compression, it reduces communication by 32%. Note that the baselines are already very effective in reducing bandwidth; without compression, this workload requires 1.9 GB of commu-



(a) No versioning (b) Version on close (c) Version on write

Figure 3: Bytes sent to the server for each individual user study participant (A-G). We compare Knockoff with two baselines across all relevant versioning policies.

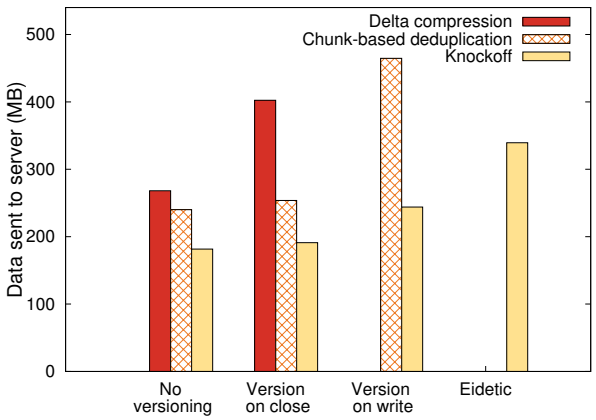


Figure 4: Total bytes sent to the server across all user study participants. We compare Knockoff with two baselines across all relevant versioning policies.

nication, so delta compression is achieving a 86% reduction in network bandwidth, and chunk-based deduplication is achieving a 87% reduction.

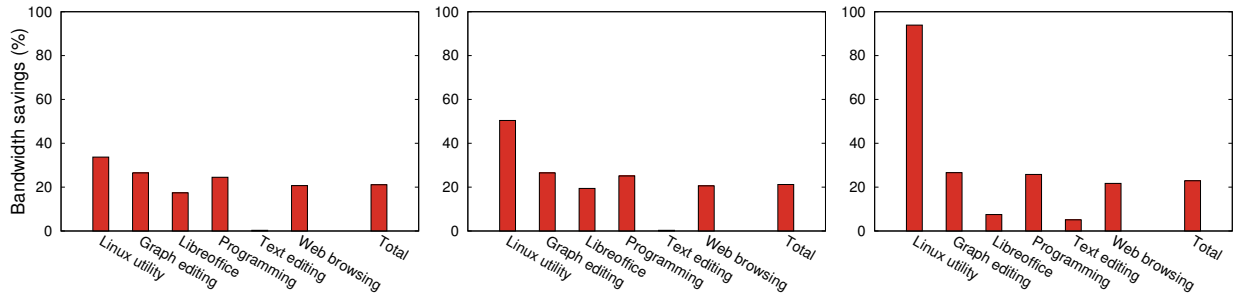
Results for `version on close` are similar to `no versioning` for two reasons: (1) the 10-second delay in transmitting data limits the amount of file closes that can be coalesced, and (2) file closes that occur within a few seconds of one another often save very similar data, so deduplication is very effective in reducing communication for both the baseline and Knockoff. For the `version on write` policy, Knockoff reduces bytes sent by 47% compared to chunk-based deduplication. Knockoff is very effective in reducing the additional cost of retaining fine-grained versions in this study; in fact, `version on write` with Knockoff use less bandwidth than `no versioning` with the baselines.

Figure 3 shows results for each individual study participant (labeled A-G in each graph). The most noticeable result is that the effectiveness of Knockoff varies tremendously across users. For participant C, Knockoff achieves a 97% reduction in bandwidth for the `no versioning` policy and a 95% reduction for the `version on write` policy compared to chunk-based deduplication. On the other hand, for participant F, the corresponding reductions are 2% and 17%. Participant C used more command line tools and repeated tasks than other participants. Participant F browsed Web sites more often. Unfortunately, Knockoff mispredicted whether to ship by log or value for browsing sessions and missed several opportunities for bandwidth reduction. Running a longer study might have allowed Knockoff to better model behavior and make better predictions for this user.

6.3 Bandwidth savings

To assess longer-term impacts of running Knockoff, one author ran Knockoff on his primary computer for 20 days. The usage was not continuous, as the author was simultaneously developing the system and fixing bugs. When in use, all user-level applications were recorded, and almost all data was stored in Knockoff. There were a few exceptions that included system directories, maintenance operations, and the execution of user-level daemons like the X server. Knockoff was initially populated by mirroring the current data in the computer's file system at the beginning of the trial; copying this data into Knockoff is excluded from our results. The first column of Table 1 shows aggregate characteristics of this recorded workload.

Figure 6 compares the bytes sent to the server by Knockoff with those sent by the baseline file systems. For the `no versioning` policy, Knockoff reduced bytes sent by 21% compared to chunk-based deduplication and by 39% compared to delta compression. Note that these compression techniques already reduce bytes sent by 84% and 79%, respectively, when compared to using no compression at all. For the `version on write` policy, Knockoff reduced bytes sent by 21% compared to chunk-based deduplication. In this experiment, Knockoff's implementation of fine-grained versioning policies is competitive with chunk-based deduplication without versioning, sending 21% more bytes to the cloud for `version on write` and 96% more for `eidetic`. This is a very



(a) No versioning (b) Version on close (c) Version on write

Figure 5: Relative reduction in bytes sent to the server for the 20-day study

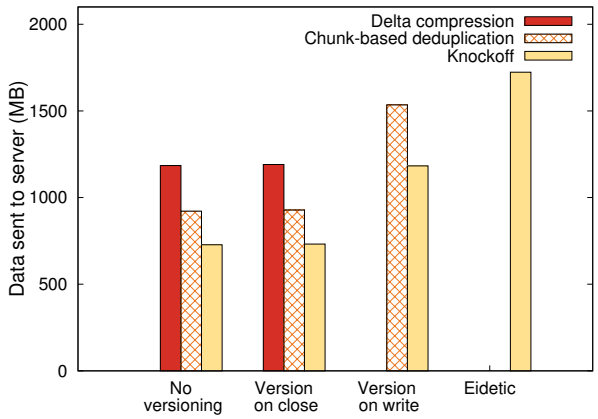


Figure 6: Bytes sent to the server for the 20-day study. We compare Knockoff with two baselines across all relevant versioning policies.

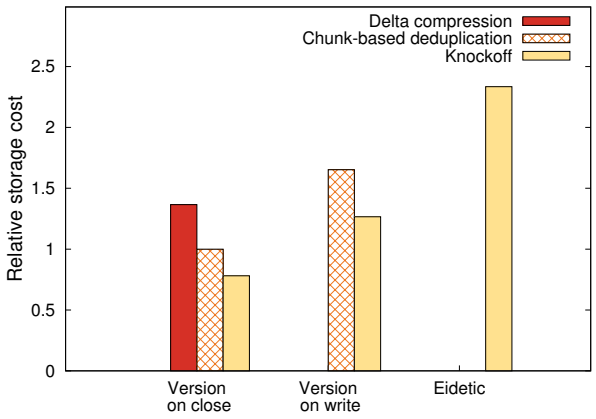


Figure 7: Relative storage costs for different versioning policies for the 20-day study.

encouraging result as it argues that retention of past state at fine granularity can be economically feasible.

To further explore these results, we manually classified the logs collected during the trial by application type. Figure 5 shows the reduction in bytes sent to the cloud for each type relative to chunk-based deduplication. Knockoff helps little for text editing because the log of nondeterminism is almost always larger than the data file produced by the editor. All other application types show robust reductions in bytes sent, with the savings being the greatest for Linux command line utilities.

6.4 Storage savings

We next examine how Knockoff impacts storage costs for the 20-day study. Storage costs typically depend on the amount of data stored; e.g., AWS currently charges $0.045 per GB-month [5]. Since Knockoff stores all current versions by value, we compare and report the amount of storage consumed by all file systems to store past versions. The Knockoff materialization delay limit for past versions is set to its default of 60 seconds.

Figure 7 shows the cost of storing versions of past state normalized to the chunk-based deduplication baseline. Compared to this baseline, Knockoff reduces storage utilization by 19% and 23% for the `version on close` and `version on write` policies, respectively.

Storage utilization increases as the version granularity gets smaller. However, with Knockoff, storing every write is only 21% more costly than the baseline system versioning on close, and eidetic storage is only 134% more costly. Thus, even versioning at eidetic granularity can be economically feasible if the storage system stores some past versions by operation rather than by value.

Figure 8 shows how changing the materialization delay impacts the relative storage cost. The values to the far right represent an infinite materialization delay, and thus show the minimum cost possible through varying this parameter.

6.5 Communication cost savings

We next measure the cost savings achieved by Knockoff. Sending file data to the server by operation reduces network usage, but it requires server-side computation to regenerate file data. We assess this tradeoff using current rates charged by popular Internet and cloud service providers. For network, we use a range of possible values. Popular broadband ISPs (AT&T [4] and Com-

	Price($ per GB)	Knockoff savings					
		No version		Version on close		Version on write	
		20-day study	User study	20-day study	User study	20-day study	User study
4G network	4.50	21.0%	21.8%	21.2%	21.7%	22.9%	46.3%
Expensive ISP	0.20	20.3%	18.4%	20.5%	18.5%	22.0%	43.3%
Cheap ISP	0.05	18.1%	13.8%	18.2%	14.5%	19.2%	34.9%
Hypothetical ISP	0.005	8.2%	4.9%	8.4%	5.8%	8.2%	11.5%

Table 2: Relative cost savings from using Knockoff for different versioning policies. We show costs for a typical 4G cellular network, an expensive current ISP, a cheap current ISP, and a hypothetical ISP that is an order of magnitude cheaper than the cheap current ISP.

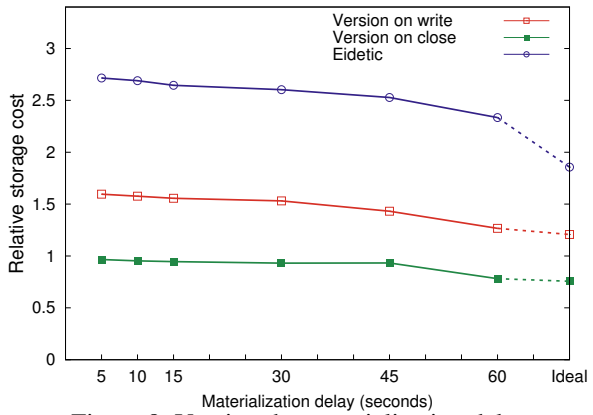


Figure 8: Varying the materialization delay

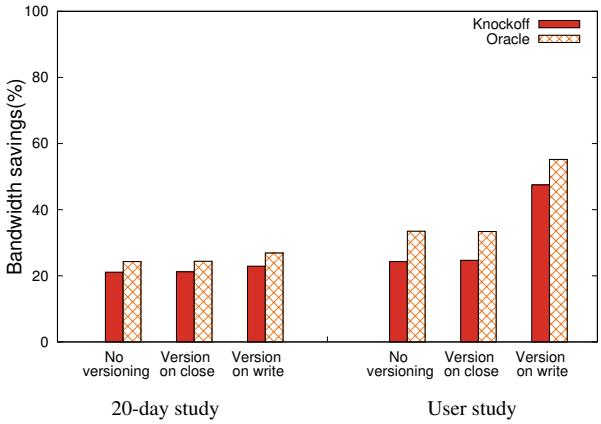


Figure 9: We compare Knockoff's actual bandwidth savings with those it could achieve with an oracle that perfectly predicts whether to ship by value or by operation.

cast [11]) currently charge a base ($50 per month) and an incremental charge ($10 for every 50 GB of data beyond a monthly data cap between 150GB and 1TB). Thus, we consider 2 price points: $0.05/GB and $0.20/GB. We also consider a typical 4G network, which has much higher cost ($4.50/GB or more), and a hypothetical cheap ISP ($0.005/GB) that is an order of magnitude lower than current providers. Cloud computation cost depends on the capabilities of the instance. AWS currently charges $0.052 per hour for an instance with 2 vCPUs and 4 GB of memory [5]. This is the cheapest instance sufficient to replay all of our logs, so we use this cost for our study.

Table 2 compares the monetary cost of sending data to the server for Knockoff and chunk-based deduplication (the best of the two baselines). For high network cost (4G), the cost savings of using Knockoff are essentially identical to the bandwidth savings. As network cost decreases, Knockoff's cost benefit diminishes. However, even for the hypothetical cheap ISP, Knockoff still achieves a healthy 4.9-11.5% reduction in dollar cost.

The reason why monetary cost savings aligns closely with network bandwidth savings for most network types is that the current tradeoff between communication and network costs is very favorable for operation shipping. Replaying applications is proportional to user-level CPU time because it eliminates user think-time and most I/O delays. When applications do not fit this profile, e.g., they have a lot of computation or large logs of nondeterminism, Knockoff usually ships the data by value.

6.6 Effectiveness of prediction

For long-running programs, Knockoff must predict whether it will be better to ship the output of that program by value or by operation. Mispredictions increase the bytes sent to the server. To measure this cost, we calculated the bytes that would be sent for our studies if an oracle were to perfectly predict which method Knockoff should use. As the results in Figure 9 show, better predictions could reduce network communication, but the potential improvement is not especially large.

6.7 Performance

We next examine Knockoff's performance overhead as perceived by its user. We measured this overhead by compiling libelf-0.8.9 with all source files, executables, and compilation outputs stored in Knockoff. We report the mean time to compile across 8 trials. Note that Knockoff sends data to the server asynchronously, and the server also replays logs asynchronously. As a baseline, we use a FUSE file system that simply forwards all file system operations to a local ext4 file system.

Figure 10 shows the results of our experiment. The first bar shows the baseline file system. The next bar

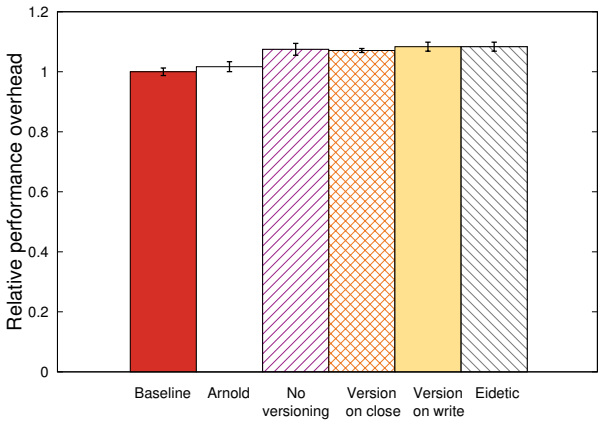


Figure 10: Performance overhead building libelf-0.8.9

shows the relative performance when we use Arnold to record the compilation with data stored in the baseline file system. This shows that the isolated cost of using deterministic record and replay is 2%. We then show relative performance when using Knockoff with its different versioning policies. The average performance overhead for using Knockoff ranges from 7% to 8%; the relative costs of different policies are equivalent within experimental error as shown by the overlapping 95% confidence intervals.

6.8 Log compression

Finally, we examine the benefit of using delta compression on logs of nondeterminism. Across all logs, delta compression reduces the bytes needed to store those logs by 42%. In comparison, chunk-based deduplication reduces the size of the logs by only 33%.

We find it interesting that chunk-based deduplication is more effective for compressing file data, whereas delta compression is more effective for compressing nondeterminism in the computation that produced that data. It is possible that restructuring the logs to make them more amenable to either delta compression or chunk-based deduplication could lead to further savings.

7 Conclusion

Operation shipping has long been recognized as a promising technique for reducing the cost of distributed storage. However, using operation shipping in practice has required onerous restrictions about application determinism or standardization of computing platforms, and these assumptions make operation shipping unsuitable for general-purpose file systems. Knockoff leverages recent advances in deterministic record and replay to lift those restrictions. It can represent, communicate, and store file data as logs of nondeterminism. This saves network communication and reduces storage utilization, leading to cost savings. In the future, we hope to extend the ideas in Knockoff to other uses; one promising target is reducing cross-data-center communication.

Acknowledgments

We thank the anonymous reviewers and our shepherd An-I Wang for their thoughtful comments. This work has been supported by the National Science Foundation under grants CNS-1513718 and CNS-1421441 and by a gift from NetApp. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, NetApp, or the University of Michigan.

References

[1] ADAMS, I. F., LONG, D. D. E., MILLER, E. L., PASUPATHY, S., AND STORER, M. W. Maximizing efficiency by trading storage for computation. In *Proceedings of the Workshop on Hot Topics in Cloud Computing* (June 2009).

[2] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 193–206.

[3] Apple time machine. `https://support.apple.com/en-us/HT201250`.

[4] AT&T Broadband monthly data allowance. `https://www.att.com/support/internet/usage.html`.

[5] Amazon Web Services (AWS) pricing. `https://aws.amazon.com/ec2/pricing/`.

[6] BENT, J., THAIN, D., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LIVNY, M. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation* (March 2004).

[7] Oracle Berkeley DB. `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb`.

[8] BHATTACHERJEE, S., CHAVAN, A., HUANG, S., DESHPANDE, A., AND PARAMESWARAN, A. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *The Proceedings of the VLDB Endowment* (August 2015), 1346–1357.

[9] CHACON, S., AND STRAUB, B. *Pro Git (2nd Edition)*. Apress, November 2014.

[10] CHANG, T.-Y., VELAYUTHAM, A., AND SIVAKUMAR, R. Mimic: Raw activity shipping for file synchronization in mobile file systems. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services* (Boston, MA, June 2004), pp. 165–176.

[11] Comcast Broadband monthly data allowance. https://customer.xfinity.com/help-and-support/internet/data-usage-plan.

[12] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 285–298.

[13] DEVECSERY, D., CHOW, M., DOU, X., FLINN, J., AND CHEN, P. M. Eidetic systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (Broomfield, CO, October 2014).

[14] Dropbox. http://www.dropbox.com.

[15] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M., AND CHEN, P. M. Execution replay on multiprocessor virtual machines. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2008), pp. 121–130.

[16] ELNOZAHY, E. N., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (September 2002), 375–408.

[17] Filesystem in Userspace. http://fuse.sourceforge.net/.

[18] Google drive. https://www.google.com/drive/.

[19] GORDON, M., HONG, D. K., CHEN, P. M., FLINN, J., MAHLKE, S., AND MAO, Z. M. Accelerating mobile applications through flip-flop replication. In *Proceedings of the 13th International Conference on Mobile Systems, Applications and Services* (2015).

[20] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[21] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (2010).

[22] LEE, C., LEHOCZKY, J., RAJKUMAR, R., AND SIEWIOREK, D. On quality of service optimization with discrete QoS options. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium* (June 1999).

[23] LEE, Y.-W., LEUNG, K.-S., AND SATYANARAYANAN, M. Operation shipping for mobile file systems. *IEEE Transactions on Computers 51*, 12 (December 2002), 1410–1422.

[24] MASHTIZADEH, A. J., BITTAU, A., HUANG, Y. F., AND MAZIÈRES, D. Replication, history, and grafting in the ori file system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farminton, PA, October 2013), pp. 151–166.

[25] Microsoft onedrive. https://onedrive.live.com/about/en-us/.

[26] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS) 17*, 1 (March 1992), 94–162.

[27] MUMMERT, L., EBLING, M., AND SATYANARAYANAN, M. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995).

[28] MUNISWAMY-REDDY, K.-K., AND HOLLAND, D. A. Causality-based versioning. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (San Francisco, CA, February 2009).

[29] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference* (Boston, MA, May/June 2006), pp. 43–56.

[30] MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A., AND ZADOK, E. A versatile and user-oriented versioning file system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (San Francisco, CA, March/April 2004), pp. 115–128.

[31] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Banff, Canada, October 2001), pp. 174–187.

[32] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 177–191.

[33] PARKER, D. S., POPEK, G. J., RUDISIN, G., STOUGHTON, A., WALKER, B. J., WALTON, E., CHOW, J. M., EDWARDS, D., KISER, S., AND KLINE, C. Detection of mutual inconsistencies in distributed systems. *IEEE Transactions on Software Engineering SE-9*, 3 (May 1983), 240–247.

[34] QUINN, A., DEVECSERY, D., CHEN, P. M., AND FLINN, J. JetStream: Cluster-scale parallelization of information flow queries. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation* (Savannah, GA, November 2016).

[35] rr: lightweight recording and deterministic debugging. http://www.rr-project.org.

[36] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. *SIGOPS Operating Systems Review 33*, 5 (1999), 110–123.

[37] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22*, 4 (December 1990), 299–319.

[38] SHILANE, P., HUANG, M., WALLACE, G., AND HSU, W. WAN optimized replication of backup datasets using stream-informed delta compression. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012).

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

[40] TRIDGELL, A., AND MACKERRAS, P. The rsync algorithm. Tech. Rep. TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra, Australia, 1996.

[41] VAHDAT, A., AND ANDERSON, T. Transparent result caching. In *Proceedings of the 1998 USENIX Annual Technical Conference* (June 1998).

[42] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Long Beach, CA, March 2011).

[43] VRABLE, M., SAVAGE, S., AND VOELKER, G. M. Cumulus: Filesystem backup to the cloud. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (San Francisco, CA, February 2009), pp. 225–238.

[44] Wayback: User-level versioning file system for linux. `http://wayback.sourceforge.net/`.

[45] Xdelta. `http://xdelta.org/`.

HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases

Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling
KTH - Royal Institute of Technology
{smkniazi, maism, haridi, jdowling}@kth.se

Steffen Grohsschmiedt
Spotify AB
steffeng@spotify.com

Mikael Ronström
Oracle
mikael.ronstrom@oracle.com

Abstract

Recent improvements in both the performance and scalability of shared-nothing, transactional, in-memory NewSQL databases have reopened the research question of whether distributed metadata for hierarchical file systems can be managed using commodity databases. In this paper, we introduce HopsFS, a next generation distribution of the Hadoop Distributed File System (HDFS) that replaces HDFS' single node in-memory metadata service, with a distributed metadata service built on a NewSQL database. By removing the metadata bottleneck, HopsFS enables an order of magnitude larger and higher throughput clusters compared to HDFS. Metadata capacity has been increased to at least 37 times HDFS' capacity, and in experiments based on a workload trace from Spotify, we show that HopsFS supports 16 to 37 times the throughput of Apache HDFS. HopsFS also has lower latency for many concurrent clients, and no downtime during failover. Finally, as metadata is now stored in a commodity database, it can be safely extended and easily exported to external systems for online analysis and free-text search.

1 Introduction

Distributed file systems are an important infrastructure component of many large scale data-parallel processing systems, such as MapReduce [13], Dryad [27], Flink [5] and Spark [77]. By the end of this decade, data centers storing multiple exabytes of data will not be uncommon [12, 47]. For large distributed hierarchical file systems, the metadata management service is the scalability bottleneck [62]. Many existing distributed file systems store their metadata on either a single node or a shared-disk file systems, such as storage-area network (SAN), both of which have limited scalability. Well known examples include GFS [17], HDFS [61], QFS [41], Farsite [3], Ursa Minor [2], GPFS [58], Frangipani [67], GlobalFS [50], and Panasas [73]. Other systems scale out their metadata by statically sharding the namespace and storing the shards on different hosts, such as NFS [44], AFS [36], MapR [64], Locus [49], Coda [57], Sprite [40] and XtreemFS [26]. However, statically sharding the namespace negatively affects file system operations that cross different shards, in particular *move* operation. Also, it complicates the management of the file system, as administrators have to map metadata servers to namespace shards that change in size over time.

Recent improvements in both the performance and scalability of shared-nothing, transactional, in-memory NewSQL [42] databases have reopened the possibility of storing distributed file system metadata in a commodity database. To date, the conventional wisdom has been that it is too expensive (in terms of throughput and latency) to store hierarchical file system metadata fully normalized in a distributed database [59, 33].

In this paper we show how to build a high throughput and low operational latency distributed file system using a NewSQL database. We present HopsFS, a new distribution of the Hadoop Distributed File System (HDFS) [61], which decouples file system metadata storage and management services. HopsFS stores all metadata normalized in a highly available, in-memory, distributed, relational database called Network Database (NDB), a NewSQL storage engine for MySQL Cluster [38, 54]. HopsFS provides redundant stateless servers (namenodes) that in parallel, read and update metadata stored in the database.

HopsFS encapsulates file system operations in distributed transactions. To improve the performance of file system operations, we leverage both classical database techniques such as *batching* (bulk operations) and *write-ahead* caches within transactions, as well as distribution aware techniques commonly found in NewSQL databases. These distribution aware NewSQL techniques include *application defined partitioning* (we partition the namespace such that the metadata for all immediate descendants of a directory (child files/directories) reside on the same database shard for efficient directory listing), and *distribution aware transactions* (we start a transaction on the database shard that stores all/most of the metadata required for the file system operation), and *partition pruned index scans* (scan operations are localized to a single database shard [78]). We also introduce an inode hints cache for faster resolution of file paths. Cache hits when resolving a path of depth N can reduce the number of database round trips from N to 1.

However, some file system operations on large directory subtrees (such as *move*, and *delete*) may be too large to fit in a single database transaction. For example, deleting a folder containing millions of files cannot be performed using a single database transaction due to the limitations imposed by the database management system

on the number of operations that can be included in a single transaction. For these *subtree operations*, we introduce a novel protocol that uses an application level distributed locking mechanism to isolate large subtrees to perform file system operations. After isolating the subtrees large file system operations are broken down into smaller transactions that execute in parallel for performance. The subtree operations protocol ensures that the consistency of the namespace is not violated if the namenode executing the operation fails.

HopsFS is a drop-in replacement for HDFS. HopsFS has been running in production since April 2016, providing Hadoop-as-a-Service for researchers at a data center in Luleå, Sweden [63]. In experiments, using a real-world workload generated by Hadoop/Spark applications from Spotify, we show that HopsFS delivers 16 times higher throughput than HDFS, and HopsFS has no downtime during failover. For a more write-intensive workload, HopsFS delivers 37 times the throughput of HDFS. To the best of our knowledge HopsFS is the first open-source distributed file system that stores fully normalized metadata in a distributed relational database.

2 Background

This section describes Hadoop Distributed File System (HDFS) and MySQL Cluster Network Database (NDB) storage engine.

2.1 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) [61] is an open source implementation of the Google File System [17]. HDFS' metadata is stored on the heap of single Java process called the Active NameNode (ANN), see Figure 1. The files are split into small (typically 128 MB) blocks that are by default triple replicated across the datanodes. For high availability of the metadata management service, the Active namenode logs changes to the metadata to journal servers using quorum based replication. The metadata change log is replicated asynchronously to a Standby NameNode (SbNN), which also performs checkpointing functionality. In HDFS, the ZooKeeper coordination service [25] enables both agreement on which machine is running the active namenode (preventing a split-brain scenario) as well as coordinating failover from the active to the standby namenode.

The namenode serves requests from potentially thousands of datanodes and clients, and keeps the metadata strongly consistent by executing the file system operations atomically. The namenode implements atomic operations using a single global lock on the entire file system metadata, providing single-writer, multiple-readers concurrency semantics. Some large file system operations are not atomic, as they would hold the global lock for too long, starving clients. For example, deleting large directories is performed in batches, with inodes first being deleted, then the blocks are deleted in later phases. Moreover, as writing namespace changes to the quorum of journal nodes can take long time, the global file system lock is released before the operation is logged to prevent other clients from starving. Concurrent clients can acquire the file system lock before the previous operations are logged, preventing starvation, at the cost of inconsistent file system operations during namenode failover. For example, when the active namenode fails all the changes that are not logged to the journal nodes will be lost.

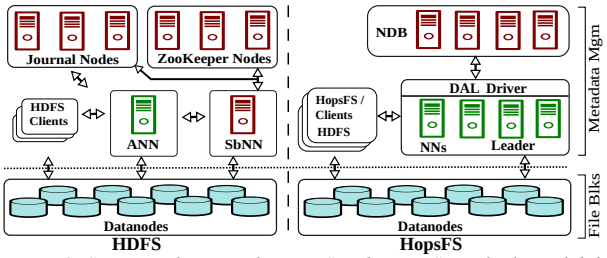


Figure 1: System architecture for HDFS and HopsFS. For high availability, HDFS requires an Active NameNode (ANN), at least one Standby NameNode (SbNN), at least three Journal Nodes for quorum-based replication of the write ahead log of metadata changes, and at least three ZooKeeper instances for quorum based coordination. HopsFS supports multiple stateless namenodes that access the metadata stored in NDB database nodes.

The datanodes are connected to both active and standby namenodes. All the datanodes periodically generate a block report containing information about its own stored blocks. The namenode processes the block report to validate the consistency of the namenode's blocks map with the blocks actually stored at the datanode.

In HDFS the amount of metadata is quite low relative to file data. There is approximately 1 gigabyte of metadata for every petabyte of file system data [62]. Spotify's HDFS cluster has 1600+ nodes, storing 60 petabytes of data, but its metadata fits in 140 gigabytes Java Virtual Machine (JVM) heap. The extra heap space is taken by temporary objects, RPC request queues and secondary metadata required for the maintenance of the file system. However, current trends are towards even larger HDFS clusters (Facebook has HDFS clusters with more than 100 petabytes of data [48]), but current JVM garbage collection technology does not permit very large heap sizes, as the application pauses caused by the JVM garbage collector affects the operations of HDFS [22]. As such, JVM garbage collection technology and the monolithic architecture of the HDFS namenode are now the scalability bottlenecks for Hadoop [62]. Another limitation with this architecture is that data structures are optimized to reduce their memory footprint with the result that metadata is difficult to modify or export to external systems.

2.2 Network Database (NDB)

MySQL Cluster is a shared-nothing, replicated, in-memory, auto-sharding, *consistent*, NewSQL relational database [38]. Network DataBase (NDB) is the storage

engine for MySQL Cluster. NDB supports both datanode-level and cluster-level failure recovery. The datanode-level failure recovery is performed using transaction redo and undo logs. NDB datanodes also asynchronously snapshot their state to disk to bound the size of logs and to improve datanode recovery time. Cluster-level recovery is supported using a global checkpointing protocol that increments a global epoch-ID, by default every 2 seconds. On cluster-level recovery, datanodes recover all transactions to the latest epoch-ID.

NDB horizontally partitions the tables among storage nodes called NDB datanodes. NDB also supports application defined partitioning (ADP) for the tables. Transaction coordinators are located at all NDB datanodes, enabling high performance transactions between data shards, that is, multi-partition transactions. Distribution aware transactions (DAT) are possible by providing a *hint*, based on the application defined partitioning scheme, to start a transaction on the NDB datanode containing the data read/updated by the transaction. In particular, single row read operations and partition pruned index scans (scan operations in which a single data shard participates) benefit from distribution aware transactions as they can read all their data locally [78]. Incorrect hints result in additional network traffic being incurred but otherwise correct system operation.

2.2.1 NDB Data Replication and Failure Handling

NDB datanodes are organized into node groups, where the data replication factor, R, determines the number of datanodes in a node group. Given a cluster size N, there are N/R node groups. NDB partitions tables (hash partitioning by default) into a fixed set of partitions distributed across the node groups. New node groups can be added online, and existing data is automatically rebalanced to the new node group. A partition is a fragment of data stored and replicated by a node group. Each datanode stores a copy (replica) of the partition assigned to its node group. In NDB, the default replication degree is two, which means that each node group can tolerate one NDB datanode failure as the other NDB datanode in the node group contains a full copy of the data. So, a twelve node NDB cluster has six node groups can tolerate six NDB datanode failures as long as there is one surviving NDB datanode in each of the node groups. To tolerate multiple failures within a node group, the replication degree can be increased at the cost of lower throughput.

2.2.2 Transaction Isolation

NDB only supports *read-committed* transaction isolation, which guarantees that any data read is committed at the moment it is read. The *read-committed* isolation level does not allow *dirty* reads but *phantom* and *fuzzy* (non-repeatable) reads can happen in a transaction [7]. However, NDB supports row level locks, such as, *exclusive* (write) locks, *shared* (read) locks, and *read-committed* locks that can be used to isolate conflicting transactions.

3 HopsFS Overview

HopsFS is a fork of HDFS v2.0.4. Unlike HDFS, HopsFS provides a scale-out metadata layer by decoupling the metadata storage and manipulation services. HopsFS supports multiple stateless namenodes, written in Java, to handle clients' requests and process the metadata stored in an external distributed database, see Figure 1. Each namenode has a Data Access Layer (DAL) driver that, similar to JDBC, encapsulates all database operations allowing HopsFS to store the metadata in a variety of NewSQL databases. The internal management (housekeeping) operations, such as datanode failure handling, must be coordinated amongst the namenodes. HopsFS solves this problem by electing a leader namenode that is responsible for the housekeeping. HopsFS uses the database as shared memory to implement a leader election and membership management service. The leader election protocol assigns a unique ID to each namenode, and the ID of the namenode changes when the namenode restarts. The leader election protocol defines an alive namenode as one that can write to the database in bounded time, details for which can be found in [56].

Clients can choose between *random*, *round-robin*, and *sticky* policies for selecting a namenode on which to execute file system operations. HopsFS clients periodically refresh the namenode list, enabling new namenodes to join an operational cluster. HDFS v2.x clients are fully compatible with HopsFS, although they do not distribute operations over namenodes, as they assume there is a single active namenode. Like HDFS, the datanodes are connected to all the namenodes, however, the datanodes send the block reports to only one namenode. The leader namenode load balances block reports over all alive namenodes.

In section 4, we discuss how HopsFS' auto sharding scheme enables common file system operations to read metadata using low cost database access queries. Section 5 discusses how the consistency of the file system metadata is maintained by converting file system operations into distributed transactions, and how the latency of the distributed transactions is reduced using per-transaction and namenode level caches. Then, in section 6, a protocol is introduced to handle file system operations that are too large to fit in a single database transaction.

4 HopsFS Distributed Metadata

Metadata for hierarchical distributed file systems typically contains information on inodes, blocks, replicas, quotas, leases and mappings (directories to files, files to blocks, and blocks to replicas). When metadata is distributed, an application defined partitioning scheme is needed to

Op Name	Percentage	Op Name	Percentage
append file	0.0%	content summary	0.01%
mkdirs	0.02%	set permissions	0.03% [26.3%*]
set replication	0.14%	set owner	0.32 % [100%*]
delete	0.75% [3.5%*]	create file	1.2%
move	1.3% [0.03%*]	add blocks	1.5%
list (*listStatus*)	**9% [94.5%*]**	**stat (*fileInfo*)**	**17% [23.3%*]**
read (*getBlkLoc*)	**68.73%**	**Total Read Ops**	**94.74%**

Table 1: Relative frequency of file system operations for Spotify's HDFS cluster. List, read, and stat operations account for ≈ 95% of the metadata operations in the cluster.
**Of which, the relative percentage is on directories*

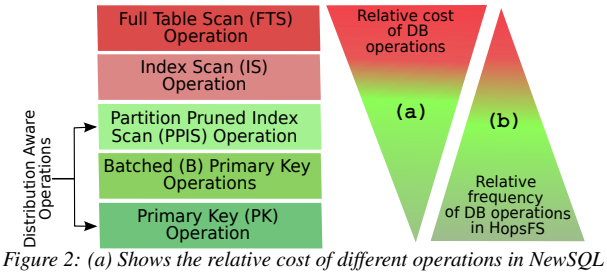


Figure 2: (a) Shows the relative cost of different operations in NewSQL database. (b) HopsFS avoids FTS and IS operations as the cost these operation is relatively higher than PPIS, B, and PK operations.

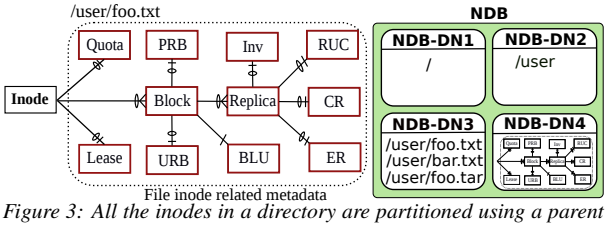


Figure 3: All the inodes in a directory are partitioned using a parent inode ID, therefore, all the immediate children of /user directory are stored on NDB-DN-3 for efficient directory listing, for example, ls /user. The file inode related metadata for /user/foo.txt is stored on NDB-DN-4 for efficient file reading operations, for example, cat /user/foo.txt.

shard the metadata and a consensus protocol is required to ensure metadata integrity for operations that cross shards. Quorum-based consensus protocols, such as Paxos, provide high performance within a single shard, but are typically combined with transactions, implemented using the two-phase commit protocol, for operations that cross shards, as in Megastore [6] and Spanner [10]. File system operations in HopsFS are implemented primarily using multi-partition transactions and row-level locks in MySQL Cluster to provide serializability [23] for metadata operations.

The choice of partitioning scheme for the hierarchical namespace is a key design decision for distributed metadata architectures. We base our partitioning scheme on the expected relative frequency of HDFS operations in production deployments and the cost of different database operations that can be used to implement the file system operations. Table 1 shows the relative frequency of selected HDFS operations in a workload generated by Hadoop applications, such as, *Pig*, *Hive*, *HBase*, *MapReduce*, *Tez*, *Spark*, and *Giraph* at Spotify. *List*, *stat* and *file read* operations alone account for ≈ 95% of the operations in the HDFS cluster. These statistics are similar to the published workloads for Hadoop clusters at Yahoo [1], LinkedIn [52], and Facebook [65]. Figure 2a shows the relative cost of different database operations. We can see that the cost of a full table scan or an index scan, in which all database shards participate, is much higher than a partition pruned index scan in which only a single database shard participates. HopsFS metadata design and metadata partitioning enables implementations of common file system operations using only the low cost database operations, that is, primary key operations, batched primary key operations and partition pruned index scans. For example, the read and directory listing operations, are implemented using only (batched) primary key lookups and partition pruned index scans. Index scans and full table scans were avoided, where possible, as they touch all database shards and scale poorly.

4.1 Entity Relation Model

In HopsFS, the file system metadata is stored in tables where a directory inode is represented by a single row in the *Inode* table. File inodes, however, have more associated metadata, such as a set of blocks, block locations, and checksums that are stored in separate tables.

Figure 3 shows the Entity Relational model depicting key entities in the HopsFS metadata model. Files and directories are represented by the *Inode* entity that contains a reference to its parent inode (parent inode ID) in the file system hierarchy. We store path individual components, not full paths, in inode entries. Each file contains multiple blocks stored in the *Block* entity. The location of each block replica is stored in the *Replica* entity. During its life-cycle a block goes through various phases. Blocks may be under-replicated if a datanode fails and such blocks are stored in the under-replicated blocks table (*URB*). The replication manager, located on the leader namenode, sends commands to datanodes to create more replicas of under-replicated blocks. Blocks undergoing replication are stored in the pending replication blocks table (*PRB*). Similarly, a replica of a block has various states during its life-cycle. When a replica gets corrupted, it is moved to the corrupted replicas (*CR*) table. Whenever a client writes to a new block's replica, this replica is moved to the replica under construction (*RUC*) table. If too many replicas of a block exist (for example, due to recovery of a datanode that contains blocks that were re-replicated), the extra copies are stored in the excess replicas (*ER*) table and replicas that are scheduled for deletion are stored in the invalidation (*Inv*) table. Note that the file inode related entities also contain the inode's foreign key (not shown in Figure 3) that is also the partition key, enabling HopsFS to read the file inode related metadata using partition pruned index scans.

4.2 Metadata Partitioning

With the exception of hotspots (see the following subsection), HopsFS partitions *inodes* by their *parents' inode IDs*, resulting in inodes with the same parent inode being stored on the same database shard. This has the effect of uniformly partitioning the metadata among all database shards and it enables the efficient implementation of the directory listing operation. When listing files in a directory, we use a hinting mechanism to start the transaction on a transaction coordinator located on the database shard that holds the child inodes for that directory. We can then use a pruned index scan to retrieve the contents of the directory locally. File inode related metadata, that is, blocks, replica mappings and checksums, is partitioned using the file's *inode ID*. This results in metadata for a given file all being stored in a single database shard, again enabling efficient file operations, see Figure 3.

4.2.1 Hotspots

A hotspot is an inode that receives a high proportion of file system operations. The maximum number of file system operations that can be performed on a 'hot' inode is limited by the throughput of the database shard that stores the inode. Currently, HopsFS does not have any built in mechanisms for identifying hotspots at run time.

All file system operations involve resolving the path components to check for user permissions and validity of the path. The *root* inode is shared among all file system valid paths. Naturally the *root* inode is read by all file system path resolution operations. The database shard that stores the *root* inode becomes a bottleneck as all file system operations will retrieve the *root* inode from the same database shard. HopsFS caches the *root* inode at all the namenodes. In HopsFS, the *root* inode is immutable, that is, we do not allow operations, such as, renaming, deleting or changing the permissions of the *root* inode. Making the *root* inode immutable prevents any inconsistencies that could result from its caching.

In HopsFS, all path resolution operations start from the second path component (that is, the top level directories). For the top-level directories, our partitioning scheme inadvertently introduced a hotspot – all top-level directories and files are children of the root directory, and, therefore, resided on the same database shard. Operations on those inodes were handled by a single shard in the database. To overcome this bottleneck, HopsFS uses a configurable directory partitioning scheme where the immediate children of the top level directories are pseudo-randomly partitioned by hashing the names of the children. By default, HopsFS pseudo-randomly partitions only the first two levels of the file system hierarchy, that is, the root directory and its immediate descendants. However, depending on the file system workloads it can be configured to pseudo-randomly partition additional levels at the cost of slowing down *move* and *ls* operations at the top levels of the file system hierarchy.

5 HopsFS Transactional Operations

Transactional metadata operations in HopsFS belong to one of the two categories: **Inode** operations that operate on single file, directory or block (for example, create/read file, mkdir, and block state change operations), and **subtree** operations that operate on an unknown number of inodes, potentially millions, (for example, recursive *delete*, *move*, *chmod*, and *chown* on non-empty directories).

This section describes how HopsFS efficiently encapsulates inode operations in transactions in NDB. The strongest transaction isolation level provided by NDB is *read-committed*, which is not strong enough to provide at least as strong consistency semantics as HDFS which uses single global lock to serialize all HDFS operations. To this end, we use row-level locking to serialize conflicting inode operations. That is, the operations execute in parallel as long as they do not take conflicting locks on the same inodes. However, taking multiple locks in a transaction could lead to extensive deadlocking and transaction timeouts. The reasons are:

Cyclic Deadlocks: In HDFS, not all inode operations follow the same order in locking the metadata which would lead to cyclic deadlocks in our case. To solve this problem, we have reimplemented all inode operations so that they acquire locks on the metadata in the same *total order*, traversing the file system tree from the root down to leave nodes using left-ordered depth-first search.

Lock Upgrades: In HDFS, many inode operations contain read operations followed by write operations on the same metadata. When translated into database operations within the same transaction, this results in deadlocking due to lock upgrades from read to exclusive locks. We have examined all locks acquired by the inode operations, and re-implemented them so that all data needed in a transaction is read only once at the start of the transaction (see Lock Phase, section 5.2.1) at the strongest lock level that could be needed during the transaction, thus preventing lock upgrades.

5.1 Inode Hint Cache

Resolving paths and checking permissions is by far the most common operation in most HDFS workloads, see Table 1. In HDFS, the full path is recursively resolved into individual components. In HopsFS for a path of depth N, it would require N roundtrips to the database to retrieve file path components, resulting in high latency for file system operations.

Similar to AFS [36] and Sprite [40], we use *hints* [30] to speed up the path lookups. Hints are mechanisms to quickly retrieve file path components in parallel (batched operations). In our partitioning scheme, inodes have a

composite primary key consisting of the parent inode's ID and the name of the inode (that is, file or directory name), with the parent inode's ID acting as the partition key. Each namenode caches only the primary keys of the inodes. Given a pathname and a hit for all path components directories, we can discover the primary keys for all the path components which are used to read the path components in parallel using a single database batch query containing only primary key lookups.

5.1.1 Cache Consistency

We use the inode hint cache entries to read the whole inodes in a single batch query at the start of a transaction for a file system operation. If a hint entry is invalid, a primary key read operation fails and path resolution falls back to recursive method for resolving file path components, followed by repairing the cache. Cache entries infrequently become stale as move operations, that update the primary key for an inode, are less than 2% of operations in typical Hadoop workloads, see Table 1. Moreover, typical file access patterns follow a heavy-tailed distribution (in Yahoo 3% of files account for 80% of accesses [1]), and using a *sticky* policy for HopsFS clients improves temporal locality and cache hit rates.

5.2 Inode Operations

HopsFS implements a pessimistic concurrency model that supports parallel read and write operations on the namespace, serializing conflicting inode and subtree operations. We chose a pessimistic scheme as, in contrast to optimistic concurrency control, it delivers good performance for medium to high levels of resource utilization [4], and many HDFS clusters, such as Spotify's, run at high load. Inode operations are encapsulated in a single transaction that consists of three distinct phases, which are, **lock**, **execute**, and **update** phases.

5.2.1 Lock Phase

In the lock phase, metadata is locked and read from the database with the strongest lock that will be required for the duration of the transaction. Locks are taken in the *total order*, defined earlier. Inode operations are path-based and if they are not read-only operations, they only modify the last component(s) of the path, for example, *rm /etc/conf* and *chmod +x /bin/script*. Thus, only the last component(s) of the file paths are locked for file system operations.

Figure 4 shows a transaction template for HopsFS inode operations. Using the inode hint cache the primary keys for the file path components are discovered, line 1. The transaction is started on the database shard that holds all or most of the desired data, line 2. A batched operation reads all the file path components up to the penultimate path component without locking (*read-committed*) the metadata, line 3. For a path of depth *N*, this removes *N-1* round trips to the database. If the inode hints are invalid

```
 1. Get hints from the inodes hint cache
 2. Set partition key hint for the transaction
BEGIN TRANSACTION
LOCK PHASE:
 3. Using the inode hints, batch read all inodes
    up to the penultimate inode in the path
 4. If (cache miss || invalid path component) then
      recursively resolve the path & update the cache
 5. Lock and read the last inode
 6. Read Lease, Quota, Blocks, Replica, URB, PRB, RUC,
     CR, ER, Inv using partition pruned index scans
EXECUTE PHASE:
 7. Process the data stored in the transaction cache
UPDATE PHASE:
 8. Transfer the changes to database in batches
COMMIT/ABORT TRANSACTION
```

Figure 4: Transaction template showing different optimization techniques, for example, setting a partition key hint to start a distribution aware transaction, inode hints to validate the file path components using a batch operation, and partition pruned index scans to read all file inode related metadata.

then the file path is recursively resolved and the inode hint cache is updated, line 4.

After the path is resolved, either a shared or an exclusive lock is taken on the last inode component in the path, line 5. Shared locks are taken for read-only inode operations, while exclusive locks are taken for inode operations that modify the namespace. Additionally, depending on the operation type and supplied operation parameters, inode related data, such as block, replica, and PRB, are read from the database in a predefined total order using partition pruned scans operations, line 6.

HopsFS uses *hierarchical locking* [19] for inode operations, that is, if data is arranged in tree like hierarchy and all data manipulation operations traverse the hierarchy from top to bottom, then taking a lock on the root of the tree/subtree implicitly locks the children of the tree/subtree. The entity relation diagram for file inode related data, see Figure 3, shows that the entities are arranged in a tree with an inode entity at the root. That is, taking a lock on an inode implicitly locks the tree of file inode related data. As in all operations, inodes are read first, followed by its related metadata. For some operations, such as creating files/directories and listing operations, the parent directory is also locked to prevent *phantom* and *fuzzy* reads for file system operations.

5.2.2 Per-Transaction Cache

All data that is read from the database is stored in a per-transaction cache (a snapshot) that withholds the propagation of the updated cache records to the database until the end of the transaction. The cache saves many round trips to the database as the metadata is often read and updated multiple times within the same transaction. Row-level locking of the metadata ensures the consistency of the cache, that is, no other transaction can update the metadata. Moreover, when the locks are released upon the completion of the transaction the cache is cleared.

5.2.3 Execute and Update Phases

The inode operation is performed by processing the metadata in the per-transaction cache. Updated and new metadata generated during the second phase is stored in the cache which is sent to the database in batches in the final *update* phase, after which the transaction is either committed or rolled back.

6 Handling Large Operations

Recursive operations on large directories, containing millions of inodes, are too large to fit in a single transaction, that is, locking millions of rows in a transaction is not supported in existing online transaction processing systems. These operations include *move*, *delete*, *change owner*, *change permissions*, and *set quota* operations. *Move* operation changes the absolute paths of all the descendant inodes, while *delete* removes all the descendant inodes, and the *set quota* operation affects how all the descendant inodes consume disk space or how many files/directories they can create. Similarly changing the permissions or owner of a directory may invalidate operations executing at the lower subtrees.

6.1 Subtree Operations Protocol

Our solution is a protocol that implements subtree operations incrementally in batches of transactions. Instead of row level database locks, our *subtree operations protocol* uses an application-level distributed locking mechanism to mark and isolate the subtrees. We serialize subtree operations by ensuring that all ongoing inode and subtree operations in a subtree complete before a newly requested subtree operation is executed. We implement this serialization property by enforcing the following invariants: (1) no new operations access the subtree until the operation completes, (2) the subtree is quiesced before the subtree operation starts, (3) no orphaned inodes or inconsistencies arise if failures occur.

Our subtree operations protocol provides the same consistency semantics as subtree operations in HDFS. For delete subtree operation HopsFS provides even stronger consistency semantics. Failed delete operations in HDFS can result in orphaned blocks that are eventually reclaimed by the block reporting subsystem (hours later). HopsFS improves the semantics of delete operation as failed operations does not cause any metadata inconsistencies, see section 6.2. Subtree operations have the following phases.

Phase 1: In the first phase, an exclusive lock is acquired on the root of the subtree and a *subtree lock* flag (which also contains the ID of the namenode that owns the lock) is set and persisted in the database. The flag is an indication that all the descendants of the subtree are locked with exclusive (write) lock.

Before setting the lock it is essential that there are no other *active subtree operations* at any lower level of the subtree. Setting the *subtree lock* could fail active subtree operations executing on a subset of the subtree. We store all active subtree operations in a table and query it to ensure that no subtree operations are executing at lower levels of the subtree. In a typical workload, this table does not grow too large as subtree operations are usually only a tiny fraction of all file system operations. It is important to note that during path resolution, inode and subtree operations that encounter an inode with a subtree lock turned on voluntarily abort the transaction and wait until the *subtree lock* is removed.

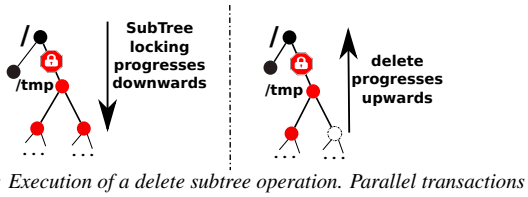


Figure 5: Execution of a delete subtree operation. Parallel transactions progress down (shown left) the subtree waiting for active operations to finish by taking and releasing write locks on all the descendant inodes. In the next phase (shown right), the delete operation is executed in batches using parallel transactions upwards from the leaf nodes.

Phase 2: To quiesce the subtree we wait for all ongoing inode operations to complete by taking and releasing database write locks on all inodes in the subtree in the same total order used to lock inodes. To do this efficiently, a pool of threads in parallel execute partition pruned index scans that write-lock child inodes. This is repeated down the subtree to the leaves, and, a tree data structure containing the inodes in the subtree is built in memory at the namenode, see Figure 5. The tree is later used by some subtree operations, such as, *move* and *delete* operations, to process the inodes. We reduce the overhead of reading all inodes in the subtree by using projections to only read the inode IDs. If the subtree operations protocol fails to quiesce the subtree due to concurrent file system operations on the subtree, it is retried with exponential backoff.

Phase 3: In the last phase the file system operation is broken down into smaller operations that execute in parallel. For improved performance, large batches of inodes are manipulated in each transaction.

6.2 Handling Failed Subtree Operations

HopsFS takes lazy approach to cleanup subtree locks left by the failed namenodes [45]. Each namenode maintains a list of the active namenodes provided by the leader election service. If an operation encounters an inode with a subtree lock set and the namenode ID of the *subtree lock* belongs to a dead namenode then the *subtree lock* is cleared. However, it is important that when a namenode that is executing a subtree operation fails then it should not leave the subtree in an inconsistent state. The in-memory tree built during the second phase plays an important role in keeping the namespace consistent if the namenode fails.

For example, in case of *delete* operations the subtree is deleted incrementally in *post-order* tree traversal manner using transactions. If half way through the operation the namenode fails then the inodes that were not deleted remain connected to the namespace tree. HopsFS clients will transparently resubmit the file system operation to another namenode to delete the remainder of the subtree.

Other subtree operations (*move*, *set quota*, *chmod* and *chown*) do not cause any inconsistencies as the actual operation where the metadata is modified is done in the third phase using a single transaction that only updates the root inodes of the subtrees and the inner inodes are left intact. In the case of a failure, the namenode might fail to unset the *subtree lock*, however, this is not a problem as other namenodes can easily remove the *subtree lock* when they find out that the *subtree lock* belongs to a dead namenode.

6.3 Inode and Subtree Lock Compatibility

Similar to the inode operation's locking mechanism (see section 5.2.1), subtree operations also implement hierarchical locking, that is, setting a subtree flag on a directory implicitly locks the contents of the directory. Both inode and subtree locking mechanisms are compatible with each other, respecting both of their corresponding locks. That is, a subtree flag cannot be set on a directory locked by an inode operation and an inode operation voluntarily aborts the transaction when it encounters a directory with a subtree lock set.

7 HopsFS Evaluation

As HopsFS addresses how to scale out the metadata layer of HDFS, all our experiments are designed to comparatively test the performance and scalability of the namenode(s) in HDFS and HopsFS in controlled conditions that approximate real-life file system load in big production clusters.

7.1 Experimental Setup

Benchmark: We have extended the benchmarking setup used to test the performance of Quantcast File System (QFS) [41], which is an open source C++ implementation of Google File System. The benchmarking utility is a distributed application that spawns tens of thousands of HDFS/HopsFS file system clients, distributed across many machines, which concurrently execute file system (metadata) operations on the namenode(s). The benchmark utility can test the performance of both individual file system operations and file system workloads based on industrial workload traces. HopsFS and the benchmark utility are open source and the readers are encouraged to perform their own experiments to verify our findings [21, 24].

HopsFS Setup: All the experiments were run on premise using Dell PowerEdge R730xd servers(Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz, 256 GB RAM, 4 TB 7200 RPM HDDs) connected using a single 10 GbE network adapter. Unless stated otherwise, NDB, version 7.5.3, was deployed on 12 nodes configured to run using 22 threads each and the data replication degree was 2.

HDFS Setup: In medium to large Hadoop clusters, 5 to 8 servers are required to provide high availability for HDFS metadata service, see Figure 1 and section section 2. The 5-server setup includes one active namenode, one standby namenode, at least three journal nodes collocated with at least three ZooKeeper nodes. In the 8-server setup, the ZooKeeper nodes are installed on separate servers to prevent multiple services from failing when a server fails. In our experiments Apache HDFS, version 2.7.2 was deployed on 5 servers. Based on Spotify's experience of running HDFS, we configured the HDFS namenodes with 240 client handler threads (*dfs.namenode.handler.count*).

None of the file system clients were co-located with the namenodes or the database nodes. As we are only evaluating metadata performance, all the tests created files of zero length (similar to the NNThroughputBenchmark [62]). Testing with non-empty files requires an order of magnitude more HDFS/HopsFS datanodes, and provides no further insight.

7.2 Industrial Workload Experiments

We benchmarked HopsFS using workloads based on operational traces from Spotify that operates a Hadoop cluster consisting of 1600+ nodes containing 60 petabytes of data. The namespace contains 13 million directories and 218 million files where each file on average contains 1.3 blocks. The Hadoop cluster at Spotify runs on average forty thousand jobs from different applications, such as, *Pig*, *Hive*, *HBase*, *MapReduce*, *Tez*, *Spark*, and *Giraph* every day. The file system workload generated by these application is summarized in Table 1, which shows the relative frequency of HDFS operations. At Spotify the average file path depth is 7 and average inode name length is 34 characters. On average each directory contains 16 files and 2 sub-directories. There are 289 million blocks stored on the datanodes. We use these statistics to generate file system workloads that approximate HDFS usage in production at Spotify.

Figure 6 shows that, for our industrial workload, using 60 namenodes and 12 NDB nodes, HopsFS can perform 1.25 million operations per second delivering **16 times** the throughput of HDFS. As discussed before in medium to large Hadoop clusters **5 to 8 servers** are required to provide high availability for HDFS. With equivalent hardware (2 NDB nodes and 3 namenodes), HopsFS delivers ≈10% higher throughput than HDFS. HopsFS performance increases linearly as more namenodes nodes are added to the system.

Table 2 shows the performance of HopsFS and HDFS

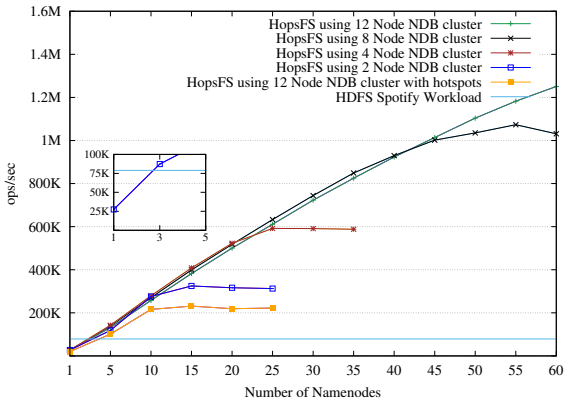


Figure 6: HopsFS and HDFS throughput for Spotify workload.

for write intensive synthetic workloads. These synthetic workloads are derived from the previously described workload, but here we increase the relative percentage of file create operations and reduce the percentage of file read operations. In this experiment, HopsFS is using 60 namenodes. As HopsFS only takes locks on inodes and subtrees, compared to HDFS' global lock, HopsFS outperforms HDFS by **37 times** for workloads where 20% of the file system operations are file create operations.

Workloads	HopsFS ops/sec	HDFS ops/sec	Scaling Factor
Spotify Workload (2.7% File Writes)	**1.25 M**	**78.9 K**	**16**
Synthetic Workload (5.0% File Writes)	1.19 M	53.6 K	**22**
Synthetic Workload (10% File Writes)	1.04 M	35.2 K	**30**
Synthetic Workload (20% File Writes)	0.748 M	19.9 K	**37**

Table 2: HDFS and HopsFS Scalability for Different Workloads.

7.2.1 Hotspots

It is not uncommon for big data applications to create millions of files in a single directory [51, 43]. As discussed in section 4.2.1 the performance of HopsFS is affected if the file system operations are not uniformly distributed among all the database shards. In this experiment, all the file system operation paths share a common ancestor, that is, */shared-dir/...*. All the file system operations manipulate files and directories with common ancestor and the file system operations are generated using the workload described in the previous section 7.2. The scalability of this workload is limited by the performance of the database shard that holds the */shared-dir*. Despite the fact that the current version of HopsFS does not yet provide a solution for scaling the performance of hotspots, the current solution outperforms HDFS by **3 times**, see Figure 6. We did not see any effect on the performance of HDFS in the presence of hotspots.

7.3 Metadata (Namespace) Scalability

In HDFS, as the entire namespace metadata must fit on the heap of single JVM, the data structures are highly optimized to reduce the memory footprint [60]. In HDFS, a file with two blocks that are replicated three ways requires ***448 + L*** bytes of metadata[1] where ***L*** represents the filename length. If the file names are 10 characters long, then a 1 GB JVM heap can store 2.3 million files. In reality the JVM heap size has to be significantly larger to accommodate secondary metadata, thousands of concurrent RPC requests, block reports that can each be tens of megabytes in size, as well as other temporary objects.

Memory	Number of Files HDFS	Number of Files HopsFS
1 GB	2.3 million	0.69 million
50 GB	115 million	34.5 million
100 GB	230 million	69 million
200 GB	**460 million**	138 million
500 GB	Does Not Scale	346 million
1 TB	Does Not Scale	708 million
24 TB	Does Not Scale	**17 billion**

Table 3: HDFS and HopsFS Metadata Scalability.

Migrating the metadata to a database causes an expansion in the amount of memory required to accommodate indexes, primary/foreign keys and padding. In HopsFS the same file described above takes **1552 bytes** if the metadata is replicated twice. For a highly available deployment with an active and standby namenodes for HDFS, you will need twice the amount of memory, thus, HopsFS requires $\approx$ **1.5** times more memory than HDFS to store metadata that is highly available. Table 3 shows the metadata scalability of HDFS and HopsFS.

NDB supports up to 48 datanodes, which allows it to scale up to 24 TB of data in a cluster with 512 GB RAM on each NDB datanode. HopsFS can store up to 17 billion files using 24 TB of metadata, which is ($\approx$**37 times**) higher than HDFS.

7.4 FS Operations' Raw Throughput

In this experiment, for each file system operation, the benchmark utility inundates the namenode(s) with the same file system operation. This test is particularly helpful in determining the maximum throughput and scalability of a particular file system operation. In real deployments, the namenode often receives a deluge of the same file system operation type, for example, a big job that reads large amounts of data will generate a huge number of requests to read files and list directories.

Figure 7 shows our results comparing the throughput for different file system operations. For each operation, HopsFS' results are displayed as a bar chart of stacked rectangles. Each rectangle represents an increase in the throughput when five new namenode are added. HopsFS outperforms HDFS for all file system operations and has significantly better performance than HDFS for the most common file system operations.

[1]These size estimates are for HDFS version 2.0.4 from which HopsFS was forked. Newer version of HDFS require additional memory for new features such as snapshots and extended attributes.

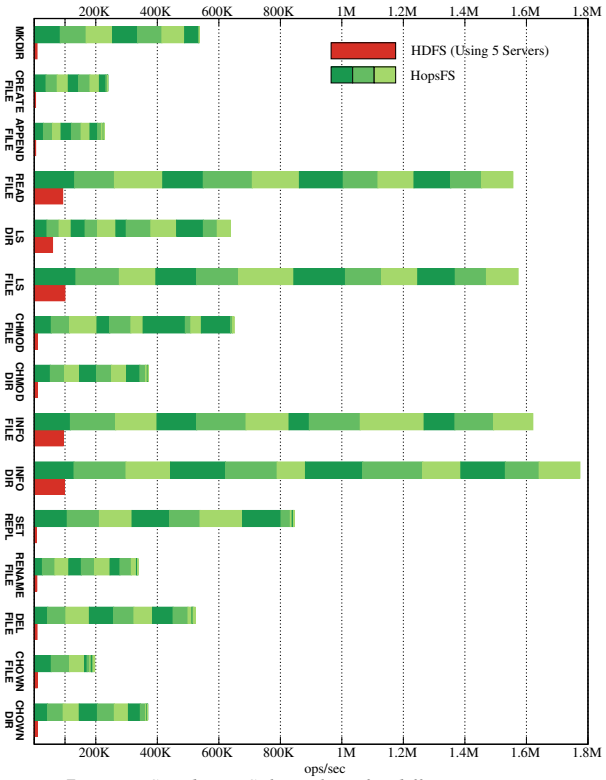


Figure 7: HopsFS and HDFS throughput for different operations. For HopsFS each shaded box represents an increase in the throughput of the file system operation when five namenodes are added. For HDFS, the shaded box represents the maximum throughput achieved using the 5-server HDFS namenode setup.

7.4.1 Subtree Operations

In Table 4, we show the latency for *move* and *delete* subtree operations on a directory containing a varying number of files, ranging from one quarter to one million files. In this experiment, the tests were performed on HopsFS and HDFS clusters under 50% load for the Spotify workload (50 % of the maximum throughput observed in figure 6).

In HopsFS, large amounts of data is read over the network and the operations are executed in many small transaction batches. The execution time of the move operation does not increase as rapidly because it does not update all the inner nodes or leaves of the subtree. HDFS outperforms HopsFS as all the data is readily available in the memory. However, due to the low frequency of such operations in typical industrial workloads (see Table 1), we think it is an acceptable trade-off for the higher performance of common file system operations in HopsFS.

7.5 Operational Latency

The latency for a single file system operation on an unloaded HDFS namenode will always be lower than in HopsFS, as all the metadata is readily available in main memory for the HDFS namenode, while it is remote for the namenodes in HopsFS. Figure 8 shows average file system operation latency observed by concurrent clients

	mv		rm -rf	
Dir Size	**HDFS**	**HopsFS**	**HDFS**	**HopsFS**
0.25 M	197 ms	1820 ms	256 ms	5027 ms
0.50 M	242 ms	3151 ms	314 ms	8589 ms
1.00 M	357 ms	5870 ms	606 ms	15941 ms

Table 4: Performance of move and delete operations on large directories.

while running the Spotify workload. For such a workload, HopsFS has lower operation latency than HDFS because in HDFS file system operations that update the namespace block all other file system operations. Large HDFS deployments, may have tens of thousands of clients [61] and the end-to-end latency observed by the clients increases as the file system operations wait in RPC call queues at the namenode [55]. In contrast, HopsFS can handle more concurrent clients while keeping operation latencies low.

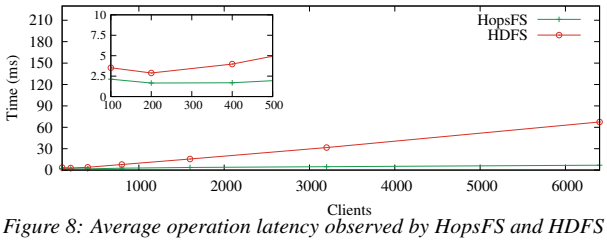


Figure 8: Average operation latency observed by HopsFS and HDFS for an increasing number of concurrent clients.

Figure 9 shows 99th percentile latencies for different file system operations in a non-overloaded cluster. In this experiment, we ran HopsFS and HDFS under 50% load for the Spotify workload (50 % of the maximum throughput observed in Figure 6). In HopsFS, 99th-percentiles for common file system operations such as touch file, read file, ls dir and stat dir are 100.8 ms, 8.6 ms, 11.4 ms and 8.5 ms, respectively. In a similar experiment for HDFS, running at 50% load, the 99th-percentile latency for touch file, read file, ls dir and stat dir are 101.8, 1.5, 0.9, and 1.5 ms respectively.

7.6 Failure Handling

Now we discuss how the performance of the HDFS and HopsFS is affected when the namenodes, NDB datanodes, and journal nodes fail.

7.6.1 Namenodes failure

Figure 10 shows how the performance of the file system metadata service is affected when a namenode fails at 50% of the load of the Spotify workload. The namenodes failures were simulated by killing and restarting all the file system processes on a namenode. For HDFS, the active namenode was periodically killed while for HopsFS, the namenodes were periodically killed in a round-robin manner. In the Figure 10, vertical lines indicate namenode failures. In HDFS, the standby namenode takes over when it detects that the active namenode has failed. In our experiments we have observed 8 to 10 seconds of downtime during failover in HDFS. During this time no file system

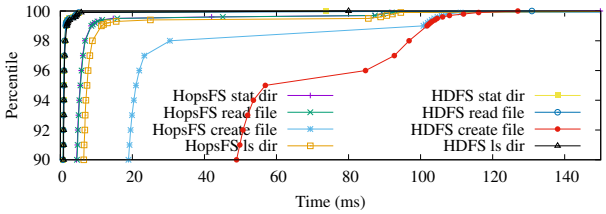


Figure 9: HopsFS and HDFS latency for common file system operations operating at 50% load.

metadata operation can be performed. Our failover tests were favorable to HDFS, as the amount of metadata stored by NNs in the experiment is minimal. At Spotify, with 140 gigabytes of metadata and 40 thousand jobs every day, failover takes at least 5 minutes, and often up to 15 minutes. Although we are unsure of the reason why, it may be due to the additional checkpointing role played by the Standby namenode. Moreover, starting a namenode takes tens of minutes to build the in-memory representation of the name space from the on disk name space image and applying outstanding redo logs. In contrast, in HopsFS when a namenode fails clients transparently re-execute failed file system operations on one of the remaining namenodes in the system. In these experiments the number of file system clients were fixed and no new clients were added during the experiment. For HopsFS the throughput gradually drops as more and more namenodes are restarted. This is due to the fact that after a namenode fails the clients switch to remaining namenodes. In the experiments, HopsFS uses *sticky* namenode selection policy and due to the fact that no new clients were started during the experiments the restarted namenodes do not receive as many file system operations requests as other namenodes.

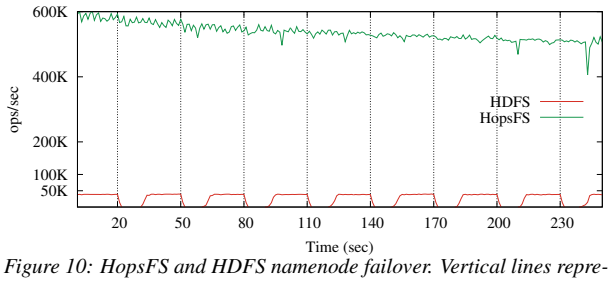


Figure 10: HopsFS and HDFS namenode failover. Vertical lines represent namenodes failures.

7.6.2 Failure of NDB Datanodes or Journal Nodes

For a HDFS cluster with N journal nodes, HDFS can tolerate failure of up to $\lceil N/2 \rceil - 1$ journal nodes. In our tests with a quorum of three journal nodes, HDFS can tolerate only one journal node failure. Increasing the size of the quorum to five enables HDFS to tolerate two journal nodes failure. We have tested HDFS with 3, 5, and 7 journal nodes, and the performance of the HDFS namenodes is not affected when the journal nodes fail provided that the quorum is not lost. When more journal nodes fail and the quorum is lost then the HDFS namenodes shutdown.

The number of NDB node failures that HopsFS can tolerate depends on the number of NDB datanodes and the replication degree. NDB is designed to provide 99.999% availability [37]. With a default NDB replication degree of 2, a 4 node NDB cluster can tolerate up to 2 NDB datanodes failures and a 12 node NDB cluster can tolerate up to 6 NDB datanodes failure in disjoint replica groups. We have tested HopsFS on 2, 4, 8 and 12 node NDB clusters and the performance of HopsFS is not affected when a NDB datanode fails as long as there is at least one remaining NDB datanode alive in each node group. If all the NDB datanodes in a node group fail, then the HopsFS namenodes shutdown.

A common complaint against the two-phase commit protocol is that it is blocking and failure of a transaction coordinator or participant will cause the system to block. NDB internally implements a transaction coordinator failover protocol that hands over transactions on a failed coordinator to a different coordinator, (the default 1500 ms heartbeat timeout gives an upper bound of 6 seconds for 3 missed heartbeats). Transaction participant failures are identified by very low transaction inactive timeouts, (the default is 1200 ms also used in our experiments and in production). In the event of a transaction participant failure, failed transactions are automatically retried by the namenode and will be handled by the surviving datanodes in that replication group.

7.7 Block Report Performance

In HDFS, each datanode periodically sends a block report to a namenode, containing IDs of all the blocks stored on the datanode. Block reports serve two purposes: (1) they help to rebuild the block location map when the namenode restarts since HDFS does not persist this information, (2) they serve as ground truth for available blocks in the system. We reimplemented the HDFS block reporting solution in HopsFS. Although the solution is fully functional it does not deliver as high throughput because a large amount of metadata is read over the network from the database by the namenodes to process a block report.

In an experiment with the same setup, 150 datanodes simultaneously submitted block report containing 100,000 blocks. With 30 namenodes, HopsFS manages to process 30 block reports per second while HDFS managed to process 60 block reports per second. However, full block-reports aren't needed as frequently in HopsFS as in HDFS, as we persist the block location mappings in the database. Even without further optimizations, with a 512 megabyte block size, and datanodes sending block reports every six hours, HopsFS can scale to handle block reporting in an exabyte cluster.

8 Related Work

The InversionFS [39] and Windows Future Storage (WinFS) [74] were some of the first monolithic file systems that stored the metadata in a relational database. Gunawi [20] showed that some file system operations, such as *fsck*, can be more efficient when implemented using a relational database.

Recently, high performance distributed databases such as HBase [16, 9], Cassandra [29], CalvinDB [69] have enabled the development of new distributed metadata management architectures in file systems such as CalvinFS [68], CassandraFS [8] and GiraffaFS [18]. All of these file systems store denormalized metadata, that is, they store the full file path with each inode which affects the subtree operations. GiraffaFS only supports file move operation in the same directory. CalvinFS relies on CalvinDB to perform large transactions. CalvinDB runs large transactions in two phases. In the first phase the lock set is identified, and in the second phase all the locks are acquired and the operation is performed, provided that the lock set has not changed. However, CalvinFS did not experimentally show that this is a viable technique for performing operations on a directory with millions of files. Production-grade online transaction processing systems have an upper bound on the number of operations that can be included in a transaction, where the upper bound is much lower than tens of millions.

IndexFS [52] and ShardFS [75] are file systems optimized for metadata workloads with a large number of small files. IndexFS and ShardFS are middleware file systems, that is, they are built on existing distributed file systems such as HDFS [61], Lustre [66], PanFS [73] and PVFS [31]. In IndexFS and ShardFS, the metadata servers handle metadata as well as user data for small files stored in local LevelDB [32] instances, and delegate the management of large files to an underlying distributed file system. For durability the LevelDB's SSTables are stored in the underlying distributed file system. IndexFS caches inode information at clients, while ShardFS caches it at metadata servers. Atomic file system operations that involves both the underlying distributed file system and IndexFS/ShardFS metadata servers are not supported. For example, atomically deleting large files whose metadata is stored in the IndexFS/ShardFS metadata server and the file data is stored by the underlying distributed file system is not supported. IndexFS [52] uses a caching mechanism to improve the performance of hot directories/files, while HopsFS' currently only load balances a user-configurable number of top-level directories. We are investigating more dynamic approaches for HopsFS.

PVFS2 [31], OrangeFS [76], Farsite [14], Lustre [66], Vesta [11], InterMezzo [46], zFS [53], and RAMA [35] shard inodes among multiple metadata servers by either (1) random partitioning or (2) partition based hashed file identifiers or hashed full/partial file paths. This partitioning scheme is typically combined with the caching of metadata at clients, which can cause cache invalidation storms for large subtree operations. Ceph dynamically partitions the file system tree, where hot-spot directories are hashed on multiple metadata servers [71, 72].

Finally, our architecture supports a pluggable NewSQL storage engine. MemSQL and SAP Hana are candidates, as they support high throughput cross-partition transactions, application defined partitioning, and partition pruned queries [34]. VoltDB is currently not a candidate as it serializes cross partition transactions [70].

9 External Metadata Implications

Administrators often resort to writing their own tools to analyze the HDFS namespace. HopsFS enables online ad hoc analytics on the metadata. With a NDB backend, HopsFS metadata can be selectively and asynchronously replicated to either a backup cluster or a MySQL slave server, enabling complex analytics without affecting the performance of the active cluster. HopsFS metadata is also easy to export to external systems and it is easy to safely extend the metadata. That is, additional tables can be created that contain a foreign key to the associated inode, thus ensuring the integrity of the extended metadata. Using this approach, we have already added new features to HopsFS, including extended attributes for inodes and erasure coding. Moreover, following similar ideas to [28], we developed an eventually consistent replication protocol that replicates (extended) HopsFS metadata to Elasticsearch [15] for free-text search. This enables us to search the entire namespace with sub-second latency. We believe that distributed metadata in a commodity database is a significant new enabling technology and it can become a reliable source of ground truth for metadata applications built on top of distributed file systems.

10 Summary

In this paper, we introduced HopsFS, that is, to the best of our knowledge, the first production-grade distributed hierarchical file system that stores its metadata in an external NewSQL database. HopsFS is an open-source, highly available file system that scales out in both capacity and throughput by adding new namenodes and database nodes. HopsFS can store 37 times more metadata than HDFS and for a workload from Spotify, HopsFS scales to handle 16 times the throughput of HDFS. HopsFS also has lower average latency for large number of concurrent clients, and no downtime during failover. Our architecture supports a pluggable database storage engine, and other NewSQL databases could be used. Finally, HopsFS makes metadata tinker friendly, opening it up for users and applications to extend and analyze in new and creative ways.

11 Acknowledgements

This work is funded by Swedish Foundation for Strategic Research project "E2E-Clouds", and by EU FP7 project "Scalable, Secure Storage and Analysis of Biobank Data" under Grant Agreement no. 317871.

References

[1] C. L. Abad. *Big Data Storage Workload Characterization, Modeling and Synthetic Generation*. PhD thesis, University of Illinois at Urbana-Champaign, 2014.

[2] 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 the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, Dec. 2002.

[4] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, Nov. 1987.

[5] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 23(6):939–964, Dec. 2014.

[6] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

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

[8] Cassandra File System Design. `http://www.datastax.com/dev/blog/cassandra-file-system-design`. [Online; accessed 1-January-2016].

[9] 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. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[10] 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 of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[11] P. Corbett, D. Feitelson, J.-P. Prost, and S. Baylor. Parallel access to files in the Vesta filesystem. In *Supercomputing '93. Proceedings*, pages 472–481, Nov 1993.

[12] E. Corporation. HADOOP IN THE LIFE SCIENCES:An Introduction. `https://www.emc.com/collateral/software/white-papers/h10574-wp-isilon-hadoop-in-lifesci.pdf`, 2012. [Online; accessed 30-Aug-2015].

[13] Dean, Jeffrey and Ghemawat, Sanjay. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[14] J. R. Douceur and J. Howell. Distributed Directory Service in the Farsite File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 321–334, Berkeley, CA, USA, 2006. USENIX Association.

[15] Elasticsearch. `https://www.elastic.co/products/elasticsearch`. [Online; accessed 1-January-2016].

[16] L. George. *HBase: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, Incorporated, 2011.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, Oct. 2003.

[18] GiraffaFS. `https://github.com/giraffafs/giraffa`. [Online; accessed 1-January-2016].

[19] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Database. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394. IFIP, 1976.

[20] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proc. of OSDI'08*, pages 131–146. USENIX Association, 2008.

[21] Hammer-Bench: Distributed Metadata Benchmark to HDFS. `https://github.com/smkniazi/hammer-bench`. [Online; accessed 1-January-2016].

[22] Hadoop JIRA: Add thread which detects JVM pauses. `https://issues.apache.org/jira/browse/HADOOP-9618`. [Online; accessed 1-January-2016].

[23] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[24] Hadoop Open Platform-as-a-Service (Hops) is a new distribution of Apache Hadoop with scalable, highly available, customizable metadata. `https://github.com/smkniazi/hammer-bench`. [Online; accessed 1-January-2016].

[25] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, 2010.

[26] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. The XtreemFS architecture—a case for object-based file systems in Grids. *Concurrency and computation: Practice and experience*, 20(17):2049–2060, 2008.

[27] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[28] C. Johnson, K. Keeton, C. B. Morrey, C. A. N. Soules, A. Veitch, S. Bacon, O. Batuner, M. Condotta, H. Coutinho, P. J. Doyle, R. Eichelberger, H. Kiehl, G. Magalhaes, J. McEvoy, P. Nagarajan, P. Osborne, J. Souza, A. Sparkes, M. Spitzer, S. Tandel, L. Thomas, and S. Zangaro. From research to practice: Experiences engineering a production metadata database for a scale out file system. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 191–198, Berkeley, CA, USA, 2014. USENIX Association.

[29] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[30] B. W. Lampson. Hints for computer system design. *SIGOPS Oper. Syst. Rev.*, 17(5):33–48, Oct. 1983.

[31] R. Latham, N. Miller, R. B. Ross, and P. H. Carns. A Next-Generation Parallel File System for Linux Clusters. *LinuxWorld Magazine*, January 2004.

[32] LevelDB. `http://leveldb.org/`. [Online; accessed 1-January-2016].

[33] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22:321–374, 1990.

[34] MemSQL, The World's Fastest Database, In Memory Database, Column Store Database. `http://www.memsql.com/`. [Online; accessed 30-June-2015].

[35] E. L. Miller and R. Katz. RAMA: An Easy-To-Use, High-Performance Parallel File System. *Parallel Computing*, 23(4):419–446, July 1997.

[36] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: A Distributed Personal Computing Environment. *Commun. ACM*, 29(3):184–201, Mar. 1986.

[37] MySQL Cluster: High Availability. `https://www.mysql.com/products/cluster/availability.html`. [Online; accessed 23-May-2016].

[38] MySQL Cluster CGE. `http://www.mysql.com/products/cluster/`. [Online; accessed 30-June-2015].

[39] M. A. Olson and M. A. The Design and Implementation of the Inversion File System, 1993.

[40] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *IEEE Computer*, 21:23–36, 1988.

[41] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. *Proc. VLDB Endow.*, 6(11):1092–1101, Aug. 2013.

[42] F. Özcan, N. Tatbul, D. J. Abadi, M. Kornacker, C. Mohan, K. Ramasamy, and J. Wiener. Are We Experiencing a Big Data Bubble? In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1407–1408, 2014.

[43] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte. GIGA+: Scalable Directories for Shared File Systems. In *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, pages 26–29, New York, NY, USA, 2007. ACM.

[44] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 - Design and Implementation. In *In Proceedings of the Summer USENIX Conference*, pages 137–152, 1994.

[45] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[46] Peter Braam Braam and Michael Callahan. The InterMezzo File System. In *In Proceedings of the 3rd of the Perl Conference, O'Reilly Open Source Convention*, Monterey, CA, USA, 1999.

[47] A. J. Peters and L. Janyst. Exabyte Scale Storage at CERN. *Journal of Physics: Conference Series*, 331(5):052015, 2011.

[48] I. Polato, R. Ré, A. Goldman, and F. Kon. A comprehensive view of Hadoop research – A systematic literature review. *Journal of Network and Computer Applications*, 46:1–25, 2014.

[49] G. Popek and B. J. Walker. *The LOCUS distributed system architecture*. MIT Press, 1985.

[50] K. W. Preslan, A. Barry, J. Brassow, R. Cattelan, A. Manthei, E. Nygaard, S. V. Oort, D. Teigland, M. Tilstra, and et al. Implementing Journaling in a Linux Shared Disk File System, 2000.

[51] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: an analysis of hadoop usage in scientific workloads. *Proceedings of the VLDB Endowment*, 6(10):853–864, 2013.

[52] K. Ren, Q. Zheng, S. Patil, and G. 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, pages 237–248, Piscataway, NJ, USA, 2014. IEEE Press.

[53] O. Rodeh and A. Teperman. zFS - a scalable distributed file system using object disks. In *Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 207–218, April 2003.

[54] M. Ronström and J. Oreland. Recovery Principles of MySQL Cluster 5.1. In *Proc. of VLDB'05*, pages 1108–1115. VLDB Endowment, 2005.

[55] RPC Congestion Control with FairCallQueue. `https://issues.apache.org/jira/browse/HADOOP-9640`. [Online; accessed 1-January-2016].

[56] G. B. Salman Niazi, Mahmoud Ismail and J. Dowling. Leader Election using NewSQL Systems. In *Proc. of DAIS 2015*, pages 158 –172. Springer, 2015.

[57] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, David, and C. Steere. Coda: A Highly available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39:447–459, 1990.

[58] Schmuck, Frank and Haskin, Roger. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[59] M. Seltzer and N. Murphy. Hierarchical File Systems Are Dead. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[60] K. Shvachko. Name-node memory size estimates and optimization proposal. `https://issues.apache.org/jira/browse/HADOOP-1687`, August 2007. [Online; accessed 11-Nov-2014].

[61] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[62] K. V. Shvachko. HDFS Scalability: The Limits to Growth. *login: The Magazine of USENIX*, 35(2):6–16, Apr. 2010.

[63] HOPS, Software-As-A-Service from SICS'S new datacenter. `https://www.swedishict.se/hops-software-as-a-service-from-sicss-new-datacenter`. [Online; accessed 23-May-2016].

[64] M. Srivas, P. Ravindra, U. Saradhi, A. Pande, C. Sanapala, L. Renu, S. Kavacheri, A. Hadke, and V. Vellanki. Map-Reduce Ready Distributed File System, 2011. US Patent App. 13/162,439.

[65] The Curse of the Singletons! The Vertical Scalability of Hadoop NameNode. `http://hadoopblog.blogspot.se/2010/04/curse-of-singletons-vertical.html`. [Online; accessed 30-Aug-2015].

[66] The Lustre Storage Architecture. `http://wiki.lustre.org/manual/LustreManual20_HTML/UnderstandingLustre.html`. [Online; accessed 30-Aug-2015].

[67] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 224–237, New York, NY, USA, 1997. ACM.

[68] A. Thomson and D. J. Abadi. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14, Santa Clara, CA, Feb. 2015. USENIX Association.

[69] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[70] VoltDB Documentation. `http://docs.voltdb.com/ReleaseNotes/`. [Online; accessed 30-June-2015].

[71] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[72] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 4–, Washington, DC, USA, 2004. IEEE Computer Society.

[73] Welch, Brent and Unangst, Marc and Abbasi, Zainul and Gibson, Garth and Mueller, Brian and Small, Jason and Zelenka, Jim and Zhou, Bin. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, Berkeley, CA, USA, 2008. USENIX Association.

[74] WinFS: Windows Future Storage. `https://en.wikipedia.org/wiki/WinFS`. [Online; accessed 30-June-2015].

[75] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson. ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 236–249, New York, NY, USA, 2015. ACM.

[76] S. Yang, W. B. Ligon III, and E. C. Quarles. Scalable distributed directory implementation on orange file system. *Proc. IEEE Intl. Wrkshp. Storage Network Architecture and Parallel I/Os (SNAPI)*, 2011.

[77] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[78] M. Zait and B. Dageville. Method and mechanism for database partitioning, Aug. 16 2005. US Patent 6,931,390.

Evolving Ext4 for Shingled Disks

Abutalib Aghayev, *Carnegie Mellon University*
Theodore Ts'o, *Google, Inc.*
Garth Gibson, *Carnegie Mellon University*
Peter Desnoyers, *Northeastern University*

Abstract

Drive-Managed SMR (Shingled Magnetic Recording) disks offer a plug-compatible higher-capacity replacement for conventional disks. For non-sequential workloads, these disks show bimodal behavior: After a short period of high throughput they enter a continuous period of low throughput.

We introduce *ext4-lazy*[1], a small change to the Linux ext4 file system that significantly improves the throughput in both modes. We present benchmarks on four different drive-managed SMR disks from two vendors, showing that ext4-lazy achieves 1.7-5.4× improvement over ext4 on a metadata-light file server benchmark. On metadata-heavy benchmarks it achieves 2-13× improvement over ext4 on drive-managed SMR disks as well as on conventional disks.

1 Introduction

Over 90% of all data in the world has been generated over the last two years [14]. To cope with the exponential growth of data, as well as to stay competitive with NAND flash-based solid state drives (SSDs), hard disk vendors are researching capacity-increasing technologies like Shingled Magnetic Recording (SMR) [20, 60], Heat Assisted Magnetic Recording (HAMR) [29], and Bit-Patterned Magnetic Recording (BPMR) [2, 13]. While HAMR and BPMR are still in the research stage, SMR allows disk manufacturers to increase areal density with existing fabrication methods. Unfortunately, this increase in density comes at the cost of increased complexity, resulting in a disk that has different behavior than Conventional Magnetic Recording (CMR) disks. Furthermore, since SMR can complement HAMR and BPMR to provide even higher growth in areal density, it is likely that all high-capacity disks in the near future will use SMR [42].

The industry has tried to address SMR adoption by introducing two kinds of SMR disks: Drive-Managed (DM-SMR) and Host-Managed (HM-SMR). DM-SMR disks are a drop-in replacement for conventional disks that offer higher capacity with the traditional block interface, but can suffer performance degradation when subjected to non-sequential write traffic. Unlike CMR disks that have a low but consistent throughput under random writes, DM-SMR disks offer high throughput for a short period followed by a precipitous drop, as shown in Figure 1. HM-SMR disks, on the other hand, offer a backward-incompatible interface that requires major changes to the I/O stacks to allow SMR-aware software to optimize their access pattern.

A new HM-SMR disk interface presents an interesting problem to storage researchers who have already proposed new file system designs based on it [10, 24, 32]. It also presents a challenge to the developers of existing file systems [12, 15, 16] who have been optimizing their code for CMR disks for years. There have been attempts to revamp mature Linux file systems like ext4 and XFS [11, 41, 42] to use the new interface, but these attempts have stalled due to the large amount of redesign required. The Log-Structured File System (LFS) [47], on the other hand, has an architecture that can be most easily adapted to an HM-SMR disk. However, although LFS has been influential, disk file systems based on it [28, 49] have not reached production quality in practice [34, 40, 48] .

We take an alternative approach to SMR adoption. Instead of redesigning for the HM-SMR disk interface, we make an incremental change to a mature, high performance file system, to optimize its performance on a DM-SMR disk. The systems community is no stranger to taking a revolutionary approach when faced with a new technology [5], only to discover that the existing system can be evolved to take the advantage of the new technology with a little effort [6]. Following a similar evolutionary approach, we take the first step to optimize ext4 file system for DM-SMR disks, observing that random writes are even more expensive on these disks, and that metadata writeback is a key generator of it.

We introduce ext4-lazy, a small change to ext4 that eliminates most metadata writeback. Like other journaling file systems [45], ext4 writes metadata twice; as Figure 2 (a) shows,

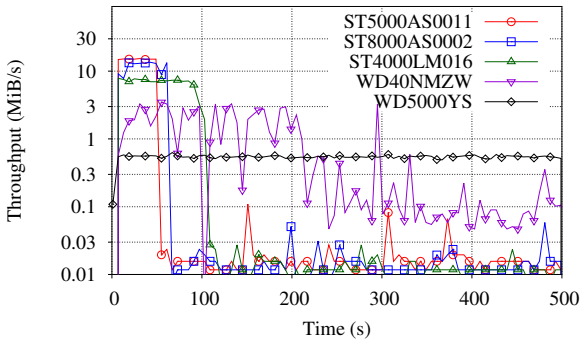


Figure 1: Throughput of CMR and DM-SMR disks from Table 1 under 4 KiB random write traffic. CMR disk has a stable but low throughput under random writes. DM-SMR disks, on the other hand, have a short period of high throughput followed by a continuous period of ultra-low throughput.

Type	Vendor	Model	Capacity	Form Factor
DM-SMR	Seagate	ST8000AS0002	8 TB	3.5 inch
DM-SMR	Seagate	ST5000AS0011	5 TB	3.5 inch
DM-SMR	Seagate	ST4000LM016	4 TB	2.5 inch
DM-SMR	Western Digital	WD40NMZW	4 TB	2.5 inch
CMR	Western Digital	WD5000YS	500 GB	3.5 inch

Table 1: CMR and DM-SMR disks from two vendors used for evaluation.

[1]The suffix *-lazy* is short for Lazy Writeback Journaling.

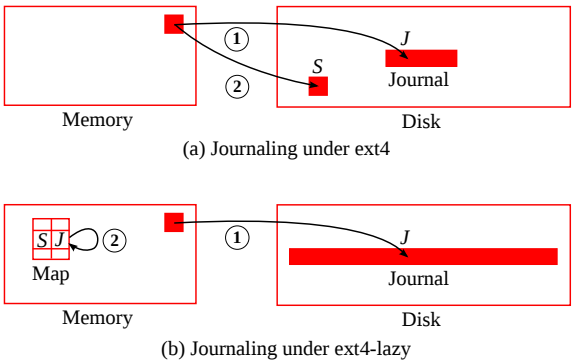


Figure 2: (a) Ext4 writes a metadata block to disk twice. It first writes the metadata block to the journal at some location J and marks it dirty in memory. Later, the writeback thread writes the same metadata block to its static location S on disk, resulting in a random write. (b) Ext4-lazy, writes the metadata block approximately once to the journal and inserts a mapping (S,J) to an in-memory map so that the file system can find the metadata block in the journal.

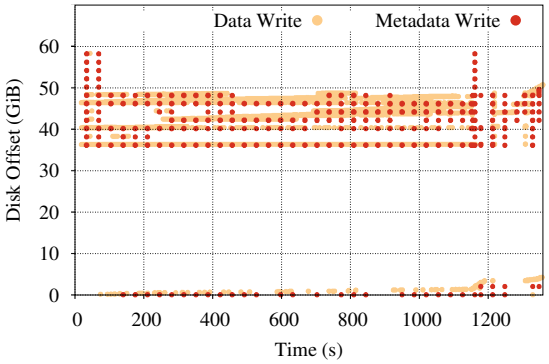


Figure 3: Offsets of data and metadata writes obtained with `blktrace` [4], when compiling Linux kernel 4.6 with all of its modules on a fresh ext4 file system. The workload writes 12 GiB of data, 185 MiB of journal (omitted from the graph), and only 98 MiB of metadata, making it 0.77% of total writes.

it first writes the metadata block to a temporary location J in the journal and then marks the block as *dirty* in memory. Once it has been in memory for long enough[2], the *writeback* (or *flusher*) thread writes the block to its *static location* S, resulting in a random write. Although metadata writeback is typically a small portion of a workload, it results in many random writes, as Figure 3 shows. Ext4-lazy, on the other hand, marks the block as *clean* after writing it to the journal, to prevent the writeback, and inserts a mapping (S,J) to an in-memory map allowing the file system to access the block in the journal, as seen in Figure 2 (b). Ext4-lazy uses a large journal so that it can continue writing updated blocks while reclaiming the space from the stale blocks. During mount, it reconstructs the in-memory map from the journal resulting in a modest increase in mount time. Our results show that ext4-lazy significantly improves performance on DM-SMR disks, as well as on CMR disks for metadata-heavy workloads.

Our key contribution in this paper is the design, implementation, and evaluation of ext4-lazy on DM-SMR and CMR disks. Our change is minimally invasive—we modify 80 lines of existing code and introduce the new functionality in additional files totaling 600 lines of C code. On a metadata-light ($\leq$ 1% of total writes) file server benchmark, ext4-lazy increases DM-SMR disk throughput by 1.7-5.4×. For directory traversal and metadata-heavy workloads it achieves 2-13× improvement on both DM-SMR and CMR disks.

In addition, we make two contributions that are applicable beyond our proposed approach:

- For purely sequential write workloads, DM-SMR disks perform at full throughput and do not suffer performance degradation. We identify the minimal sequential I/O size to trigger this behavior for a popular DM-SMR disk.
- We show that for physical journaling [45], a small journal is a bottleneck for metadata-heavy workloads. Based on our result, ext4 developers have increased the default journal size from 128 MiB to 1 GiB for file systems over 128 GiB [54].

In the rest of the paper, we first give background on SMR technology, describe why random writes are expensive in DM-SMR disks, and show why metadata writeback in ext4 is causing more random writes (§ 2). Next, we motivate ext4-lazy and describe its design and implementation (§ 3). Finally, we evaluate our implementation (§ 4), cover related work (§ 5) and present our conclusions (§ 6). Source code and other artifacts to reproduce our results are available at `http://www.pdl.cmu.edu/Publications/downloads.shtml`.

2 Background

We introduce SMR technology in general and describe how DM-SMR disks work. We then describe how ext4 lays out data on a disk and how it uses a generic layer in the kernel to enable journaling.

2.1 DM-SMR Internals

SMR leverages the difference in the width of the disk's *write head* and *read head* to squeeze more tracks into the same area than CMR. In CMR, tracks have the width of the write head even though they are read with a narrow read head, as seen in Figure 4 (a). In SMR, however, the tracks are written on top of each other, leaving just enough space for the read head to distinguish them, increasing track density, as seen in Figure 4 (b). Unlike CMR, however, overlapping writes cause the sector updates to corrupt data in adjacent tracks. Therefore, the surface of an SMR disk is divided into *bands* that are collections of narrow tracks divided by wide tracks called *guard regions*, as seen in Figure 4 (c). A band in an SMR disk represents a unit that can be safely overwritten sequentially, beginning at the first track and ending at the last. A write to any sector in a band—except to sectors in the last track of the band—will require read-modify-write (RMW) of all the tracks forming the band.

[2]Controlled by /proc/sys/vm/dirty_expire_centisecs in Linux.

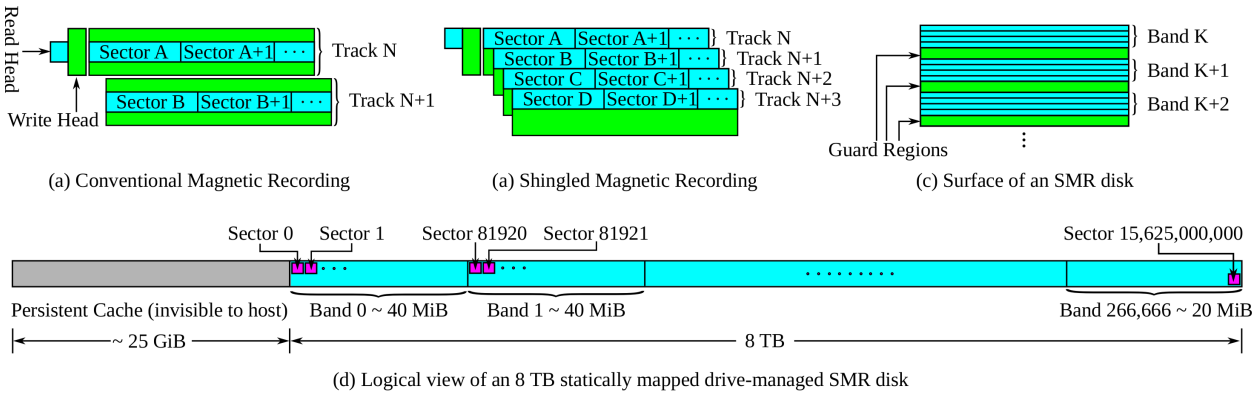


Figure 4: (a) In conventional recording the tracks have the width of the write head. (b) In shingled recording the tracks are laid partially on top of each other reducing the track width to the read head. This allows for more tracks, however, unlike with conventional recording, overwriting a sector corrupts other sectors. (c) Therefore, the surface of an SMR disk is divided into bands made up of multiple tracks separated by guard regions. (d) An SMR disk also contains a persistent cache for absorbing random writes, in addition to a sequence of bands to whom a group of sectors are mapped.

HM-SMR disks provide an interface that exposes the band information and let the host manage data on disk. One such interface is going through the standardization [23] and researchers are coming up with others [17,25,34]. DM-SMR disks, on the other hand, implement a Shingle Translation Layer (STL)—similar to the Flash Translation Layer (FTL) in SSDs—that manages data in firmware while exposing the block interface to the host.

All STLs proposed in the literature and found in actual DM-SMR disks to this date [1,9,21,22,56] contain one or more *persistent caches* for absorbing random writes, to avoid expensive RMW operation for each write. Consequently, when a write operation updates some part of a band, the STL writes the update to the persistent cache, and the band becomes dirty. An STL *cleans* a dirty band in the background or during idle times, by merging updates for the band from the persistent cache with unmodified data from the band, and writing back to the band, freeing space used by the updates in the persistent cache.

The cost of cleaning a band may vary based on the type of the *block mapping* used. With *dynamic mapping* an STL can read a band, update it in memory, write the updated band to a different band, and fix the mapping, resulting in a read and a write of a band. With *static mapping*, however, an STL needs to persist the updated band to a scratch space first—directly overwriting the band can corrupt it in case of a power failure—resulting in a read and two writes of a band.

As a concrete example, Figure 4 (d) shows the logical view of Seagate ST8000AS0002 DM-SMR disk that was recently studied in detail [1]. With an average band size of 30 MiB, the disk has over 260,000 bands with sectors statically mapped to the bands, and a ≈25 GiB persistent cache that is not visible to the host. The STL in this disk detects sequential writes and starts *streaming* them directly to the bands, bypassing the persistent cache. Random writes, however, end up in the persistent cache, dirtying bands. Cleaning a single band typically takes 1-2 seconds, but can take up to 45 seconds in extreme cases.

STLs also differ in their cleaning strategies. Some STLs constantly clean in small amounts, while others clean during idle times. If the persistent cache fills before the workload completes, the STL is forced to interleave cleaning with work, reducing throughput. Figure 1 shows the behavior of DM-SMR disks from two vendors. Seagate disks are known to clean during idle times and to have static mapping [1]. Therefore, they have high throughput while the persistent cache is not full, and ultra-low throughput after it fills. The difference in the time when the throughput drops suggests that the persistent cache size varies among the disks. Western Digital disks, on the other hand, are likely to clean constantly and have dynamic mapping [9]. Therefore, they have lower throughput than Seagate disks while the persistent cache is not full, but higher throughput after it fills.

2.2 Ext4 and Journaling

The ext4 file system evolved [30,36] from ext2 [8], which was influenced by Fast File System (FFS) [37]. Similar to FFS, ext2 divides the disk into *cylinder groups*—or as ext2 calls them, *block groups*—and tries to put all blocks of a file in the same block group. To further increase locality, the metadata blocks (*inode bitmap*, *block bitmap*, and *inode table*) representing the files in a block group are also placed within the same block group, as Figure 5 (a) shows. *Group descriptor blocks*, whose location is fixed within the block group, identify the location of these metadata blocks that are typically located in the first megabyte of the block group.

In ext2 the size of a block group was limited to 128 MiB—the maximum number of 4 KiB data blocks that a 4 KiB block bitmap can represent. Ext4 introduced *flexible block groups* or *flex_bgs* [30], a set of contiguous block groups[3] whose metadata is consolidated in the first 16 MiB of the first block group within the set, as shown in Figure 5 (b).

[3]We assume the default size of 16 block groups per flex_bg.

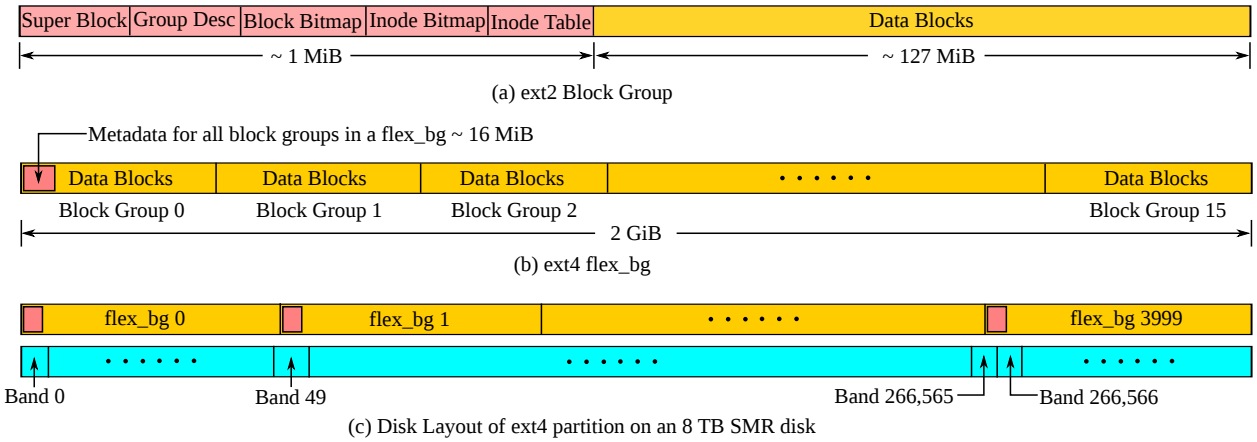


Figure 5: (a) In ext2, the first megabyte of a 128 MiB block group contains the metadata blocks describing the block group, and the rest is data blocks. (b) In ext4, a single flex_bg concatenates multiple (16 in this example) block groups into one giant block group and puts all of the metadata in the first block group. (c) Modifying data in a flex_bg will result in a metadata write that may dirty one or two bands, seen at the boundary of bands 266,565 and 266,566.

Ext4 ensures metadata consistency via journaling, however, it does not implement journaling itself; rather, it uses a generic kernel layer called the *Journaling Block Device* [55] that runs in a separate kernel thread called *jbd2*. In response to file system operations, ext4 reads metadata blocks from disk, updates them in memory, and exposes them to jbd2 for journaling. For increased performance, jbd2 batches metadata updates from multiple file system operations (by default, for 5 seconds) into a *transaction* buffer and atomically *commits* the transaction to the journal—a *circular log* of transactions with a head and tail pointer. A transaction may commit early if the buffer reaches maximum size, or if a synchronous write is requested. In addition to metadata blocks, a committed transaction contains *descriptor blocks* that record the static locations of the metadata blocks within the transaction. After a commit, jbd2 marks the in-memory copies of metadata blocks as dirty so that the writeback threads would write them to their static locations. If a file system operation updates an in-memory metadata block before its dirty timer expires, jbd2 writes the block to the journal as part of a new transaction and delays the writeback of the block by resetting its timer.

On DM-SMR disks, when the metadata blocks are eventually written back, they dirty the bands that are mapped to the metadata regions in a flex_bg, as seen in Figure 5 (c). Since a metadata region is not aligned with a band, metadata writes to it may dirty zero, one, or two extra bands, depending on whether the metadata region spans one or two bands and whether the data around the metadata region has been written.

3 Design and Implementation of ext4-lazy

We start by motivating ext4-lazy, follow with a high-level view of our design, and finish with the implementation details.

3.1 Motivation

The motivation for ext4-lazy comes from two observations: (1) metadata writeback in ext4 results in random writes that cause a significant cleaning load on a DM-SMR disk, and (2) file system metadata comprises a small set of blocks, and *hot* (frequently updated) metadata is an even smaller set. The corollary of the latter observation is that managing hot metadata in a circular log several times the size of hot metadata turns random writes into purely sequential writes, reducing the cleaning load on a DM-SMR disk. We first give calculated evidence supporting the first observation and follow with empirical evidence for the second observation.

On an 8 TB partition, there are about 4,000 flex_bgs, the first 16 MiB of each containing the metadata region, as shown in Figure 5 (c). With a 30 MiB band size, updating every flex_bg would dirty 4,000 bands on average, requiring cleaning of 120 GiB worth of bands, generating 360 GiB of disk traffic. A workload touching 1/16 of the whole disk, that is 500 GiB of files, would dirty at least 250 bands requiring 22.5 GiB of cleaning work. The cleaning load increases further if we consider floating metadata like extent tree blocks and directory blocks.

To measure the hot metadata ratio, we emulated the I/O workload of a build server on ext4, by running 128 parallel Compilebench [35] instances, and categorized all of the writes completed by disk. Out of 433 GiB total writes, 388 GiB were data writes, 34 GiB were journal writes, and 11 GiB were metadata writes. The total size of unique metadata blocks was 3.5 GiB, showing that it was only 0.8% of total writes, and that 90% of journal writes were overwrites.

3.2 Design

At a high level, ext4-lazy adds the following components to ext4 and jbd2:

Map: Ext4-lazy tracks the location of metadata blocks in the journal with *jmap*—an in-memory map that associates the static location S of a metadata block with its location J in the journal. The mapping is updated whenever a metadata block is written to the journal, as shown in Figure 2 (b).

Indirection: In ext4-lazy all accesses to metadata blocks go through jmap. If the most recent version of a block is in the journal, there will be an entry in jmap pointing to it; if no entry is found, then the copy at the static location is up-to-date.
Cleaner: The cleaner in ext4-lazy reclaims space from locations in the journal which have become *stale*, that is, invalidated by the writes of new copies of the same metadata block.
Map reconstruction on mount: On every mount, ext4-lazy reads the descriptor blocks from the transactions between the tail and the head pointer of the journal and populates jmap.

3.3 Implementation

We now detail our implementation of the above components and the trade-offs we make during the implementation. We implement jmap as a standard Linux red-black tree [31] in jbd2. After jbd2 commits a transaction, it updates jmap with each metadata block in the transaction and marks the in-memory copies of those blocks as clean so they will not be written back. We add indirect lookup of metadata blocks to ext4 by changing the call sites that read metadata blocks to use a function which looks up the metadata block location in jmap, as shown in Listing 1, modifying 40 lines of ext4 code in total.

```
−  submit_bh (READ | REQ_META | REQ_PRIO, bh);
+  jbd2_submit_bh ( journal ,  READ | REQ_META | REQ_PRIO, bh);
```

Listing 1: Adding indirection to a call site reading a metadata block.

The indirection allows ext4-lazy to be backward-compatible and gradually move metadata blocks to the journal. However, the primary reason for indirection is to be able to migrate *cold* (not recently updated) metadata back to its static location during cleaning, leaving only hot metadata in the journal.

We implement the cleaner in jbd2 in just 400 lines of C, leveraging the existing functionality. In particular, the cleaner merely reads live metadata blocks from the tail of the journal and adds them to the transaction buffer using the same interface used by ext4. For each transaction it keeps a doubly-linked list that links jmap entries containing live blocks of the transaction. Updating a jmap entry invalidates a block and removes it from the corresponding list. To clean a transaction, the cleaner identifies the live blocks of a transaction in constant time using the transaction's list, reads them, and adds them to the transaction buffer. The beauty of this cleaner is that it does not "stop-the-world", but transparently mixes cleaning with regular file system operations causing no interruptions to them, as if cleaning was just another operation. We use a simple cleaning policy—after committing a fixed number of transactions, clean a fixed number of transactions—and leave sophisticated policy development, such as hot and cold separation, for future work.

Map reconstruction is a small change to the recovery code in jbd2. Stock ext4 resets the journal on a normal shutdown; finding a non-empty journal on mount is a sign of crash and triggers the recovery process. With ext4-lazy, the state of the journal represents the persistent image of jmap, therefore, ext4-lazy never resets the journal and always "recovers". In our prototype, ext4-lazy reconstructs the jmap by reading descriptor blocks from the transactions between the tail and head pointer of the journal, which takes 5-6 seconds when the space between the head and tail pointer is ≈ 1 GiB.

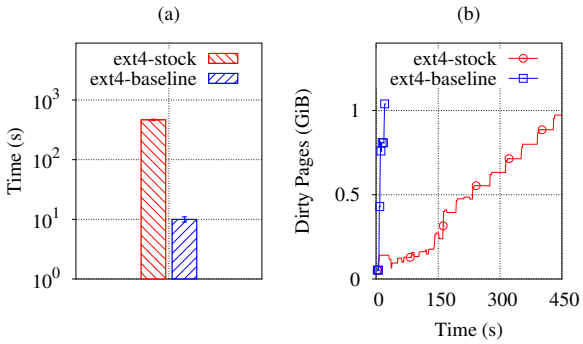


Figure 6: (a) Completion time for a benchmark creating 100,000 files on ext4-stock (ext4 with 128 MiB journal) and on ext4-baseline (ext4 with 10 GiB journal). (b) The volume of dirty pages during benchmark runs obtained by sampling `/proc/meminfo` every second.

4 Evaluation

We run all experiments on a system with a quad-core Intel i7-3820 (Sandy Bridge) 3.6 GHz CPU, 16 GB of RAM running Linux kernel 4.6 on the Ubuntu 14.04 distribution, using the disks listed in Table 1. To reduce the variance between runs, we unmount the file system between runs, always start with the same file system state, disable lazy initialization[4] when formatting ext4 partitions, and fix the writeback cache ratio [62] for our disks to 50% of the total—by default, this ratio is computed dynamically from the writeback throughput [53]. We repeat every experiment at least five times and report the average and standard deviation of the runtime.

4.1 Journal Bottleneck

Since it affects our choice of baseline, we start by showing that for metadata-heavy workloads, the default 128 MiB journal of ext4 is a bottleneck. We demonstrate the bottleneck on the CMR disk WD5000YS from Table 1 by creating 100,000 small files in over 60,000 directories, using *CreateFiles* microbenchmark from Filebench [52]. The workload size is ≈ 1 GiB and fits in memory.

Although ext4-lazy uses a large journal by definition, since enabling a large journal on ext4 is a command-line option to `mkfs`, we choose ext4 with a 10 GiB journal[5] as our baseline. In the rest of this paper, we refer to ext4 with the default journal size of 128 MiB as *ext4-stock*, and we refer to ext4 with 10 GiB journal as *ext4-baseline*.

[4] mkfs.ext4 -E lazy_itable_init=0,lazy_journal_init=0 /dev/<dev>
[5] Created by passing "-J size=10240" to mkfs.ext4.

We measure how fast ext4 can create the files in memory and do not consider the writeback time. Figure 6 (a) shows that on ext4-stock the benchmark completes in ≈460 seconds, whereas on ext4-baseline it completes 46× faster, in ≈10 seconds. Next we show how a small journal becomes a bottleneck.

The ext4 journal is a circular log of transactions with a head and tail pointer (§ 2.2). As the file system performs operations, jbd2 commits transactions to the journal, moving the head forward. A committed transaction becomes *checkpointed* when every metadata block in it is either written back to its static location due to a dirty timer expiration, or it is written to the journal as part of a newer transaction. To recover space, at the end of every commit jbd2 checks for transactions at the tail that have been checkpointed, and when possible moves the tail forward. On a metadata-light workload with a small journal and default dirty timer, jbd2 always finds checkpointed transactions at the tail and recovers the space without doing work. However, on a metadata-heavy workload, incoming transactions fill the journal before the transactions at the tail have been checkpointed. This results in a *forced checkpoint*, where jbd2 synchronously writes metadata blocks at the tail transaction to their static locations and then moves the tail forward, so that a new transaction can start [55].

We observe the file system behavior while running the benchmark by enabling tracepoints in the jbd2 code[6]. On ext4-stock, the journal fills in 3 seconds, and from then on until the end of the run, jbd2 moves the tail by performing forced checkpoints. On ext4-baseline the journal never becomes full and no forced checkpoints happen during the run.

Figure 6 (b) shows the volume of dirtied pages during the benchmark runs. On ext4-baseline, the benchmark creates over 60,000 directories and 100,000 files, dirtying about 1 GiB worth of pages in 10 seconds. On ext4-stock, directories are created in the first 140 seconds. Forced checkpoints still happen during this period, but they complete fast, as the small steps in the first 140 seconds show. Once the benchmark starts filling directories with files, the block groups fill and writes spread out to a larger number of block groups across the disk. Therefore, forced checkpoints start taking as long as 30 seconds, as indicated by the large steps, during which the file system stalls, no writes to files happen, and the volume of dirtied pages stays fixed.

This result shows that for disks, a small journal is a bottleneck for metadata-heavy buffered I/O workloads, as the journal wraps before metadata blocks are written to disk, and file system operations are stalled until the journal advances via synchronous writeback of metadata blocks. With a sufficiently large journal, all transactions will be written back before the journal wraps. For example, for a 190 MiB/s disk and a 30 second dirty timer, a journal size of 30s × 190 MiB/s = 5,700 MiB will guarantee that when the journal wraps, the transactions at the tail will be checkpointed. Having established our baseline, we move on to evaluation of ext4-lazy.

[6] /sys/kernel/debug/tracing/events/jbd2/

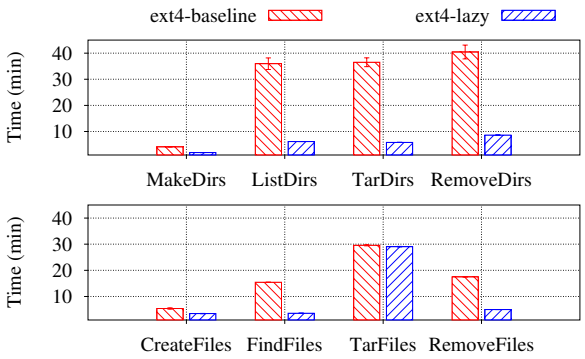


Figure 7: Microbenchmark runtimes on ext4-baseline and ext4-lazy.

4.2 Ext4-lazy on a CMR disk

We first evaluate ext4-lazy on the CMR disk WD5000YS from Table 1 via a series of microbenchmarks and a file server macrobenchmark. We show that on a CMR disk, ext4-lazy provides a significant speedup for metadata-heavy workloads, and specifically for massive directory traversal workloads. On metadata-light workloads, however, ext4-lazy does not have much impact.

4.2.1 Microbenchmarks

We evaluate directory traversal and file/directory create operations using the following benchmarks. *MakeDirs* creates 800,000 directories in a directory tree of depth 10. *ListDirs* runs `ls -lR` on the directory tree. *TarDirs* creates a tarball of the directory tree, and *RemoveDirs* removes the directory tree. *CreateFiles* creates 600,000 4 KiB files in a directory tree of depth 20. *FindFiles* runs `find` on the directory tree. *TarFiles* creates a tarball of the directory tree, and *RemoveFiles* removes the directory tree. MakeDirs and CreateFiles—microbenchmarks from Filebench—run with 8 threads and execute sync at the end. All benchmarks start with a cold cache[7].

Benchmarks that are in the file/directory create category (MakeDirs, CreateFiles) complete 1.5-2× faster on ext4-lazy than on ext4-baseline, while the remaining benchmarks that are in the directory traversal category, except TarFiles, complete 3-5× faster, as seen in Figure 7. We choose MakeDirs and RemoveDirs as a representative of each category and analyze their performance in detail.

MakeDirs on ext4-baseline results in ≈4,735 MiB of journal writes that are transaction commits containing metadata blocks, as seen in the first row of Table 2 and at the center in Figure 8 (a); as the dirty timer on the metadata blocks expires, they are written to their static locations, resulting in a similar amount of metadata writeback. The block allocator is able to allocate large contiguous blocks for

[7] `echo 3 > /proc/sys/vm/drop_caches`

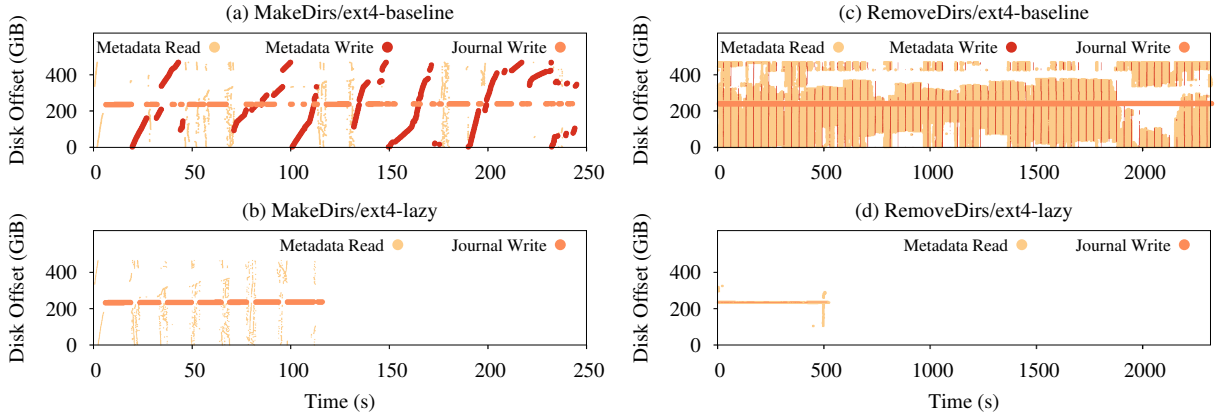


Figure 8: Disk offsets of I/O operations during MakeDirs and RemoveDirs microbenchmarks on ext4-baseline and ext4-lazy. Metadata reads and writes are spread out while journal writes are at the center. The dots have been scaled based on the I/O size. In part (d), journal writes are not visible due to low resolution. These are pure metadata workloads with no data writes.

	Metadata Reads (MiB)	Metadata Writes (MiB)	Journal Writes (MiB)
MakeDirs/ext4-baseline	143.7±2.8	4,631±33.8	4,735±0.1
MakeDirs/ext4-lazy	144±4	0	4,707±1.8
RemoveDirs/ext4-baseline	4,066.4±0.1	322.4±11.9	1,119±88.6
RemoveDirs/ext4-lazy	4,066.4±0.1	0	472±3.9

Table 2: Distribution of the I/O types with MakeDirs and RemoveDirs benchmarks running on ext4-baseline and ext4-lazy.

the directories, because the file system is fresh. Therefore, in addition to journal writes, metadata writeback is sequential as well. The write time dominates the runtime in this workload, hence, by avoiding metadata writeback and writing only to the journal, ext4-lazy halves the writes as well as the runtime, as seen in the second row of Table 2 and Figure 8 (b). On an aged file system, the metadata writeback is more likely to be random, resulting in even higher improvement on ext4-lazy.

An interesting observation about Figure 8 (b) is that although the total volume of metadata reads—shown as periodic vertical spreads—is ≈ 140 MiB (3% of total I/O in the second row of Table 2), they consume over 30% of runtime due to long seeks across the disk. In this benchmark, the metadata blocks are read from their static locations because we run the benchmark on a fresh file system, and the metadata blocks are still at their static locations. As we show next, once the metadata blocks migrate to the journal, reading them is much faster since no long seeks are involved.

In RemoveDirs benchmark, on both ext4-baseline and ext4-lazy, the disk reads ≈ 4,066 MiB of metadata, as seen in the last two rows of Table 2. However, on ext4-baseline the metadata blocks are scattered all over the disk, resulting in long seeks as indicated by the vertical spread in Figure 8 (c), while on ext4-lazy they are within the 10 GiB region in the journal, resulting in only short seeks, as Figure 8 (d) shows. Ext4-lazy also benefits from skipping metadata writeback, but most of the improvement comes from eliminating long seeks for metadata reads. The significant difference in the volume of journal writes between ext4-baseline and ext4-lazy seen in Table 2 is caused by metadata write coalescing: since ext4-lazy completes faster, there are more operations in each transaction, with many modifying the same metadata blocks, each of which is only written once to the journal.

The improvement in the remaining benchmarks, are also due to reducing seeks to a small region and avoiding metadata writeback. We do not observe a dramatic improvement in TarFiles, because unlike the rest of the benchmarks that read only metadata from the journal, TarFiles also reads data blocks of files that are scattered across the disk.

Massive directory traversal workloads are a constant source of frustration for users of most file systems [3, 18, 33, 43, 50]. One of the biggest benefits of consolidating metadata in a small region is an order of magnitude improvement in such workloads, which to our surprise was not noticed by previous work [44, 46, 61]. On the other hand, the above results are obtainable in the ideal case that all of the directory blocks are hot and therefore kept in the journal. If, for example, some part of the directory is cold and the policy decides to move those blocks to their static locations, removing such a directory will incur an expensive traversal.

4.2.2 File Server Macrobenchmark

We first show that ext4-lazy slightly improves the throughput of a metadata-light file server workload. Next we try to reproduce a result from previous work without success.

To emulate a file server workload, we started with the *Fileserver* macrobenchmark from Filebench but encountered bugs for large configurations. The development on Filebench has been recently restarted and the recommended version is still in alpha stage. Therefore, we decided to use Postmark [27], with some modifications.

Like the Fileserver macrobenchmark from Filebench, Postmark first creates a *working set* of files and directories

	Data Writes (MiB)	Metadata Writes (MiB)	Journal Writes (MiB)
ext4-baseline	34,185±10.3	480±0.2	1,890±18.6
ext4-lazy	33,878±9.8	0	1,855±15.4

Table 3: Distribution of write types completed by the disk during Postmark run on ext4-baseline and ext4-lazy. Metadata writes make 1.3% of total writes in ext4-baseline, only ⅓ of which is unique.

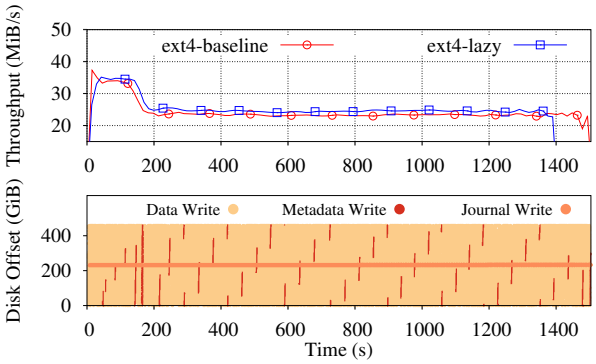


Figure 9: The top graph shows the throughput of the disk during a Postmark run on ext4-baseline and ext4-lazy. The bottom graph shows the offsets of write types during ext4-baseline run. The graph does not reflect sizes of the writes, but only their offsets.

and then executes *transactions* like reading, writing, appending, deleting, and creating files on the working set. We modify Postmark to execute `sync` after creating the working set, so that the writeback of the working set does not interfere with transactions. We also modify Postmark not to delete the working set at the end, but to run `sync`, to avoid high variance in runtime due to the race between deletion and writeback of data.

Our Postmark configuration creates a working set of 10,000 files spread sparsely across 25,000 directories with file sizes ranging from 512 bytes to 1 MiB, and then executes 100,000 transactions with the I/O size of 1 MiB. During the run, Postmark writes 37.89 GiB of data and reads 31.54 GiB of data from user space. Because ext4-lazy reduces the amount of writes, to measure its effect, we focus on writes.

Table 3 shows the distribution of data writes completed by the disk while the benchmark is running on ext4-baseline and on ext4-lazy. On ext4-baseline, metadata writes comprise 1.3% of total writes, all of which ext4-lazy avoids. As a result, the disk sees 5% increase in throughput on ext4-lazy from 24.24 MiB/s to 25.47 MiB/s and the benchmark completes 100 seconds faster on ext4-lazy, as the throughput graph in Figure 9 shows. The increase in throughput is modest because the workload spreads out the files across the disk resulting in traffic that is highly non-sequential, as data writes in the bottom graph of Figure 9 show. Therefore, it is not surprising that reducing random writes of a non-sequential write traffic by 1.3% results in a 5% throughput improvement. However, the same random writes result in extra cleaning work for DM-SMR disks (§ 2).

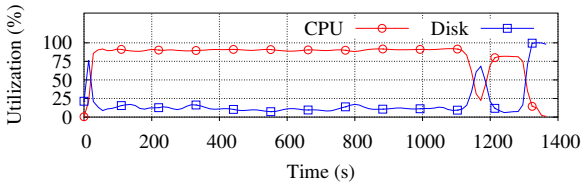


Figure 10: Disk and CPU utilization sampled from `iostat` output every second, while compiling Linux kernel 4.6 including all its modules, with 16 parallel jobs (`make -j16`) on a quad-core Intel i7-3820 (Sandy Bridge) CPU with 8 hardware threads.

Previous work [44] that writes metadata only once reports performance improvements even in a metadata-light workloads, like kernel compile. This has not been our experience. We compiled Linux kernel 4.6 with all its modules on ext4-baseline and observed that it generated 12 GiB of data writes and 185 MiB of journal writes. At 98 MiB, metadata writes comprised only 0.77% of total writes completed by the disk. This is expected, since metadata blocks are cached in memory, and because they are journaled, unlike data pages their dirty timer is reset whenever they are modified (§ 3), delaying their writeback. Furthermore, even on a system with 8 hardware threads running 16 parallel jobs, we found kernel compile to be CPU-bound rather than disk-bound, as Figure 10 shows. Given that reducing writes by 1.3% on a workload that utilized the disk 100% resulted in only 5% increase in throughput (Figure 9), it is not surprising that reducing writes by 0.77% on such a low-utilized disk does not cause improvement.

4.3 Ext4-lazy on DM-SMR disks

We show that unlike CMR disks, where ext4-lazy had a big impact on just metadata-heavy workloads, on DM-SMR disks it provides significant improvement on both, metadata-heavy and metadata-light workloads. We also identify the minimal sequential I/O size to trigger streaming writes on a popular DM-SMR disk.

An additional critical factor for file systems when running on DM-SMR disks is the cleaning time after a workload. A file system resulting in a short cleaning time gives the disk a better chance of emptying the persistent cache during idle times of a bursty I/O workload, and has a higher chance of continuously performing at the persistent cache speed, whereas a file system resulting in a long cleaning time is more likely to force the disk to interleave cleaning with file system user work.

In the next section we show microbenchmark results on just one DM-SMR disk—ST8000AS0002 from Table 1. At the end of every benchmark, we run a vendor provided script that polls the disk until it has completed background cleaning and reports the total cleaning time, which we report in addition to the benchmark runtime. We achieve similar normalized results for the remaining disks, which we skip to save space.

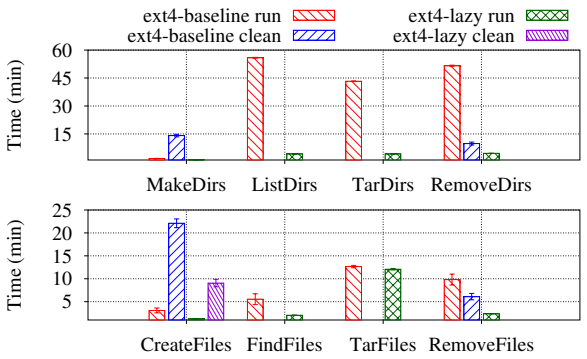


Figure 11: Microbenchmark runtimes and cleaning times on ext4-baseline and ext4-lazy running on an DM-SMR disk. Cleaning time is the additional time after the benchmark run that the DM-SMR disk was busy cleaning.

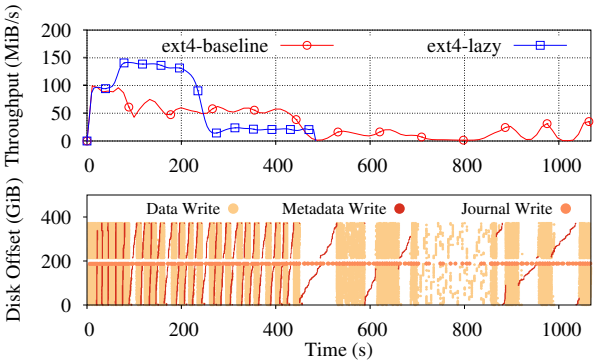


Figure 12: The top graph shows the throughput of a ST8000AS0002 DM-SMR disk with a 400 GB partition during a Postmark run on ext4-baseline and ext4-lazy. The bottom graph shows the offsets of write types during the run on ext4-baseline. The graph does not reflect sizes of the writes, but only their offsets.

4.3.1 Microbenchmarks

Figure 11 shows results of the microbenchmarks (§ 4.2.1) repeated on ST8000AS0002 with a 2 TB partition, on ext4-baseline and ext4-lazy. MakeDirs and CreateFiles do not fill the persistent cache, therefore, they typically complete 2-3× faster than on CMR disk. Similar to CMR disk, MakeDirs and CreateFiles are 1.5-2.5× faster on ext4-lazy. On the other hand, the remaining directory traversal benchmarks, ListDir for example, completes 13× faster on ext4-lazy, compared to being 5× faster on CMR disk.

The cleaning times for ListDirs, FindFiles, TarDirs, and TarFiles are zero because they do not write to disk[8]. However, cleaning time for MakeDirs on ext4-lazy is zero as well, compared to ext4-baseline's 846 seconds, despite having written over 4 GB of metadata, as Table 2 shows. Being a pure metadata workload, MakeDirs on ext4-lazy consists of journal writes only, as Figure 8 (b) shows, all of which are streamed, bypassing the persistent cache and resulting in zero cleaning time. Similarly, cleaning time for RemoveDirs and RemoveFiles are 10-20 seconds on ext4-lazy compared to 590-366 seconds on ext4-baseline, because these too are pure metadata workloads resulting in only journal writes for ext4-lazy. During deletion, however, some journal writes are small and end up in persistent cache, resulting in short cleaning times.

We confirmed that the disk was streaming journal writes in previous benchmarks by repeating the MakeDirs benchmark on the DM-SMR disk with an observation window from Skylight [1] and observing the head movement. We observed that shortly after starting the benchmark, the head moved to the physical location of the journal on the platter[9] and remained there until the end of the benchmark. This observation lead to Test 1 for identifying the minimal sequential write size that triggers streaming. Using this test, we found that sequential writes of at least 8 MiB in size are streamed. We also observed that a single 4 KiB random write in the middle of a sequential write disrupted streaming and moved the head to the persistent cache; soon the head moved back and continued streaming.

Test 1: Identify the minimal sequential write size for streaming

```
1 Choose identifiable location L on the platter
2 Start with a large sequential write size S
3 do
      Write S bytes sequentially at L
      S = S - 1 MiB
  while Head moves to L and stays there until the end of the write
4 S = S + 1 MiB
5 Minimal sequential write size for streaming is S
```

4.3.2 File Server Macrobenchmark

We show that on DM-SMR disks the benefit of ext4-lazy increases with the partition size, and that ext4-lazy achieves a significant speedup on a variety of DM-SMR disks with different STLs and persistent cache sizes.

Table 4 shows the distribution of write types completed by a ST8000AS0002 DM-SMR disk with a 400 GB partition during the file server macrobenchmark (§ 4.2.2). On ext4-baseline, metadata writes make up 1.6% of total writes. Although the unique amount of metadata is only ≈ 120 MiB, as the storage slows down, metadata writeback increases slightly, because each operation takes a long time to complete and the writeback of a metadata block occurs before the dirty timer is reset.

Unlike the CMR disk, the effect is profound on a ST8000AS0002 DM-SMR disk. The benchmark completes more than 2× faster on ext4-lazy, in 461 seconds, as seen in Figure 12. On ext4-lazy, the disk sustains 140 MiB/s throughput and fills the persistent cache in 250 seconds, and then drops to a steady 20 MiB/s until the end of the run. On ext4-baseline, however, the large number of small metadata writes reduce throughput to 50 MiB/s taking the disk 450 seconds to fill the persistent cache. Once the persistent cache fills, the disk interleaves cleaning and file system user work, and small metadata writes become prohibitively expensive,

[8] TarDirs and TarFiles write their output to a different disk.

[9] Identified by observing the head while reading the journal blocks.

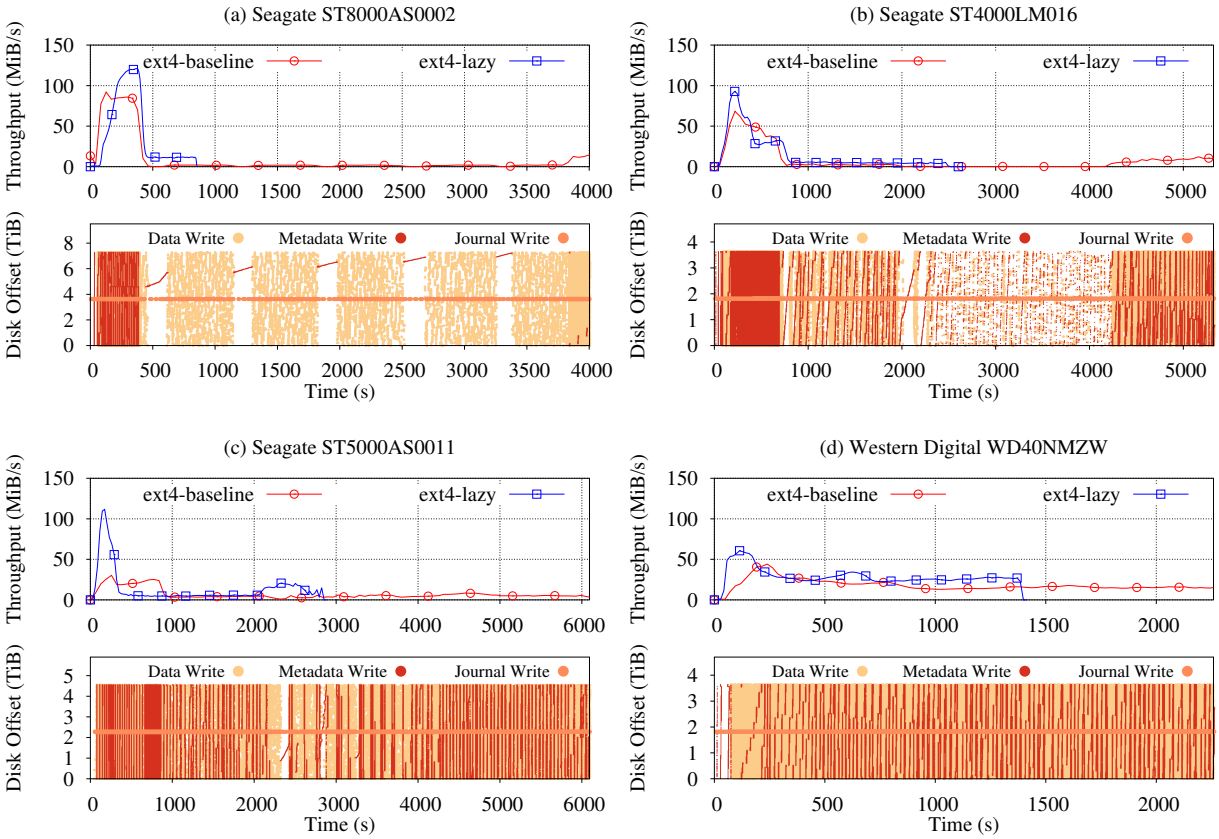


Figure 13: The top graphs show the throughput of four DM-SMR disks on a full disk partition during a Postmark run on ext4-baseline and ext4-lazy. Ext4-lazy provides a speedup of 5.4× 2×, 2×, 1.7× in parts (a), (b), (c), and (d), respectively. The bottom graphs show the offsets of write types during ext4-baseline run. The graphs do not reflect sizes of the writes, but only their offsets.

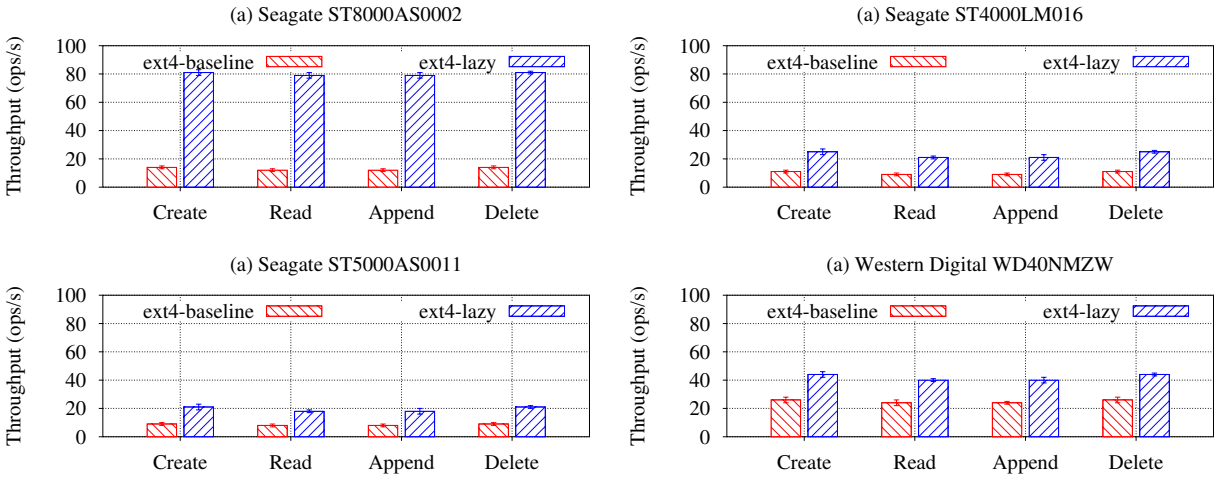


Figure 14: Postmark reported transaction throughput numbers for ext4-baseline and ext4-lazy running on four DM-SMR disks with on a full disk partition. Only includes numbers from the transaction phase of the benchmark.

	Data Writes (MiB)	Metadata Writes (MiB)	Journal Writes (MiB)
ext4-baseline	32,917±9.7	563±0.9	1,212±12.6
ext4-lazy	32,847±9.3	0	1,069±11.4

Table 4: Distribution of write types completed by a ST8000AS0002 DM-SMR disk during a Postmark run on ext4-baseline and ext4-lazy. Metadata writes make up 1.6% of total writes in ext4-baseline, only 1/5 of which is unique.

as seen, for example, between seconds 450-530. During this period we do not see any data writes, because the writeback thread alternates between page cache and buffer cache when writing dirty blocks, and it is the buffer cache's turn. We do, however, see journal writes because jbd2 runs as a separate thread and continues to commit transactions.

The benchmark completes even slower on a full 8 TB partition, as seen in Figure 13 (a), because ext4 spreads the same workload over more bands. With a small partition, updates to different files are likely to update the same metadata region. Therefore, cleaning a single band frees more space in the persistent cache, allowing it to accept more random writes. With a full partition, however, updates to different files are likely to update different metadata regions; now the cleaner has to clean a whole band to free a space for a single block in the persistent cache. Hence, after an hour of ultra-low throughput due to cleaning, it recovers slightly towards the end, and the benchmark completes 5.4× slower on ext4-baseline.

On the ST4000LM016 DM-SMR disk, the benchmark completes 2× faster on ext4-lazy, as seen in Figure 13 (b), because the disk throughput is almost always higher than on ext4-baseline. With ext4-baseline, the disk enters a long period of cleaning with ultra-low throughput starting at second 2000, and recovers around second 4200 completing the benchmark with higher throughput.

We observe a similar phenomenon on the ST5000AS0011 DM-SMR disk, as shown in Figure 13 (c). Unlike with ext4-baseline that continues with a low throughput until the end of the run, with ext4-lazy the cleaning cycle eventually completes and the workload finishes 2× faster.

The last DM-SMR disk in our list, WD40NMZW model found in My Passport Ultra from Western Digital [57], shows a different behavior from previous disks, suggesting a different STL design. We think it is using an S-blocks-like architecture [9] with dynamic mapping that enables cheaper cleaning (§ 2.1). Unlike previous disks that clean only when idle or when the persistent cache is full, WD40NMZW seems to regularly mix cleaning with file system user work. Therefore, its throughput is not as high as the Seagate disks initially, but after the persistent cache becomes full, it does not suffer as sharp of a drop, and its steady-state throughput is higher. Nevertheless, with ext4-lazy the disk achieves 1.4-2.5× increase in throughput over ext4-baseline, depending on the state of the persistent cache, and the benchmark completes 1.7× faster.

Figure 14 shows Postmark transaction throughput numbers for the runs. All of the disks show a significant improvement with ext4-lazy. An interesting observation is that, while with ext4-baseline WD40NMZW is 2× faster than ST8000AS0002, with ext4-lazy the situation is reversed and ST8000AS0002 is 2× faster than WD40NMZW, and fastest overall.

4.4 Performance Overhead

Indirection Overhead: To determine the overhead of in-memory jmap lookup, we populated jmap with 10,000 mappings pointing to random blocks in the journal, and measured the total time to read all of the blocks in a fixed random order. We then measured the time to read the same random blocks directly, skipping the jmap lookup, in the same order. We repeated each experiment five times, starting with a cold cache every time, and found no difference in total time read time—reading from disk dominated the total time of the operation.

Memory Overhead: A single jmap entry consists of a red-black tree node (3×8 bytes), a doubly-linked list node (2×8 bytes), a mapping (12 bytes), and a transaction id (4 bytes), occupying 66 bytes in memory. Hence, for example, a million-entry jmap that can map 3.8 GiB of hot metadata, requires 63 MiB of memory. Although this is a modest overhead for today's systems, it can further be reduced with memory-efficient data structures.

Seek Overhead: The rationale for introducing cylinder groups in FFS, which manifest themselves as block groups in ext4, was to create clusters of inodes that are spread over the disk close to the blocks that they reference, to avoid long seeks between an inode and its associated data [38]. Ext4-lazy, however, puts hot metadata in the journal located at the center of the disk, requiring a half-seek to read a file in the worst case. The TarFiles benchmark (§ 4.2.1) shows that when reading files from a large and deep directory tree, where directory traversal time dominates, putting the metadata at the center wins slightly over spreading it out. To measure the seek overhead on a shallow directory, we created a directory with 10,000 small files located at the outer diameter of the disk on ext4-lazy, and starting with a cold cache creating the tarball of the directory. We observed that since files were created at the same time, their metadata was written sequentially to the journal. The code for reading metadata blocks in ext4 uses readahead since the introduction of flex_bgs. As a result, the metadata of all files was brought into the buffer cache in just 3 seeks. After five repetitions of the experiment on ext4-baseline an ext4-lazy, the average times were 103 seconds and 101 seconds, respectively.

Cleaning Overhead: In our benchmarks, the 10 GiB journal always contained less than 10% live metadata. Therefore, most of the time the cleaner reclaimed space simply by advancing the tail. We kept reducing the journal size and the first noticeable slowdown occurred with a journal size of 1.4 GiB, that is, when the live metadata was ≈ 70% of the journal.

5 Related Work

Researchers have tinkered with the idea of separating metadata from data and managing it differently in local file systems before. Like many other good ideas, it may have been ahead of its time because the technology that would benefit most from it did not exist yet, preventing adoption.

The Multi-Structured File System [39] (MFS) is the first file system proposing the separation of data and metadata. It was motivated by the observation that the file system I/O is becoming a bottleneck because data and metadata exert different access patterns on storage, and a single storage system cannot respond to these demands efficiently. Therefore, MFS puts data and metadata on isolated disk arrays, and for each data type it introduces on-disk structures optimized for the respective access pattern. Ext4-lazy differs from MFS in two ways: (1) it writes metadata as a log, whereas MFS overwrites metadata in-place; (2) facilitated by (1), ext4-lazy does not require a separate device for storing metadata in order to achieve performance improvements.

DualFS [44] is a file system influenced by MFS—it also separates data and metadata. Unlike MFS, however, DualFS uses well known data structures for managing each data type. Specifically, it combines an FFS-like [37] file system for managing data, and LFS-like [47] file system for managing metadata. hFS [61] improves on DualFS by also storing small files in a log along with metadata, thus exploiting disk bandwidth for small files. Similar to these file systems ext4-lazy separates metadata and data, but unlike them it does not confine metadata to a log—it uses a hybrid design where metadata can migrate back and forth between file system and log as needed. However, what really sets ext4-lazy apart is that it is not a new prototype file system; it is an evolution of a production file system, showing that a journaling file system can benefit from the metadata separation idea with a small set of changes that does not require on-disk format changes.

ESB [26] separates data and metadata on ext2, and puts them on CMR disk and SSD, respectively, to explore the effect of speeding up metadata operations on I/O performance. It is a virtual block device that sits below ext2 and leverages the fixed location of static metadata to forward metadata block requests to an SSD. The downside of this approach is that unlike ext4-lazy, it cannot handle floating metadata, like directory blocks. ESB authors conclude that for metadata-light workloads speeding up metadata operations will not improve I/O performance on a CMR disk, which aligns with our findings (§ 4.2.2).

A separate metadata server is the norm in distributed object-based file systems like Lustre [7], Panasas [59], and Ceph [58]. TableFS [46] extends the idea to a local file system: it is a FUSE-based [51] file system that stores metadata in LevelDB [19] and uses ext4 as an object store for large files. Unlike ext4-lazy, TableFS is disadvantaged by FUSE overhead, but still it achieves substantial speedup against production file systems on metadata-heavy workloads.

In conclusion, although it is likely that the above file systems could have taken a good advantage of DM-SMR disks, they could not have shown it because all of them predate the hardware. We reevaluate the metadata separation idea in the context of a technological change and demonstrate its amplified advantage.

6 Conclusion

Our work is the first step in adapting a legacy file system to DM-SMR disks. It shows how effective a well-chosen small change can be. It also suggests that while three decades ago it was wise for file systems depending on the block interface to scatter the metadata across the disk, today, with large memory sizes that cache metadata and with changing recording technology, putting metadata at the center of the disk and managing it as a log looks like a better choice. Our work also rekindles an interesting question: How far can we push a legacy file system to be SMR friendly?

We conclude with the following general takeaways:

- We think modern disks are going to practice more extensive "lying" about their geometry and perform deferred cleaning when exposed to random writes; therefore, file systems should work to eliminate structures that induce small isolated writes, especially if the user workload is not forcing them.
- With modern disks operation costs are asymmetric: Random writes have a higher ultimate cost than random reads, and furthermore, not all random writes are equally costly. When random writes are unavoidable, file systems can reduce their cost by confining them to the smallest perimeter possible.

7 Acknowledgements

We thank anonymous reviewers, Sage Weil (our shepherd), Phil Gibbons, and Greg Ganger for their feedback; Tim Feldman and Andy Kowles for the SMR disks and for their help with understanding the SMR disk behavior; Lin Ma, Prashanth Menon, and Saurabh Kadekodi for their help with experiments; Jan Kara for reviewing our code and for explaining the details of Linux virtual memory. We thank the member companies of the PDL Consortium (Broadcom, Citadel, Dell EMC, Facebook, Google, HewlettPackard Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate Technology, Tintri, Two Sigma, Uber, Veritas, Western Digital) for their interest, insights, feedback, and support. This research is supported in part by National Science Foundation under award CNS-1149232.

References

[1] A. Aghayev and P. Desnoyers. Skylight—A Window on Shingled Disk Operation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 135–149, Santa Clara, CA, USA, Feb. 2015. USENIX Association.

[2] T. R. Albrecht, H. Arora, V. Ayanoor-Vitikkate, J.-M. Beaujour, D. Bedau, D. Berman, A. L. Bogdanov, Y.-A. Chapuis, J. Cushen, E. E. Dobisz, et al. Bit-Patterned Magnetic Recording: Theory, Media Fabrication, and Recording Performance. *IEEE Transactions on Magnetics*, 51(5):1–42, 2015.

[3] AskUbuntu. Is there any faster way to remove a directory than rm -rf? http://askubuntu.com/questions/114969.

[4] J. Axboe et al. blktrace. http://git.kernel.org/cgit/linux/kernel/git/axboe/blktrace.git.

[5] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[6] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[7] P. J. Braam et al. The Lustre Storage Architecture. http://lustre.org/.

[8] R. Card, T. Tso, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. In *Proceedings of the first Dutch International Symposium on Linux*, volume 1, 1994.

[9] Y. Cassuto, M. A. A. Sanvido, C. Guyot, D. R. Hall, and Z. Z. Bandic. Indirection Systems for Shingled-recording Disk Drives. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–14, Washington, DC, USA, 2010. IEEE Computer Society.

[10] S. P. M. Chi-Young Ku. An SMR-aware Append-only File System. In *Storage Developer Conference*, Santa Clara, CA, USA, Sept. 2015.

[11] D. Chinner. SMR Layout Optimization for XFS. http://xfs.org/images/f/f6/Xfs-smr-structure-0.2.pdf, Mar. 2015.

[12] D. Chinner. XFS: There and Back... ...and There Again? In *Vault Linux Storage and File System Conference*, Boston, MA, USA, Apr. 2016.

[13] E. A. Dobisz, Z. Bandic, T.-W. Wu, and T. Albrecht. Patterned Media: Nanofabrication Challenges of Future Disk Drives. *Proceedings of the IEEE*, 96(11):1836–1846, Nov. 2008.

[14] Å. Dragland. Big Data – for better or worse. http://www.sintef.no/en/latest-news/big-data--for-better-or-worse/, May 2013.

[15] J. Edge. Ideas for supporting shingled magnetic recording (SMR). https://lwn.net/Articles/592091/, Apr 2014.

[16] J. Edge. Filesystem support for SMR devices. https://lwn.net/Articles/637035/, Mar 2015.

[17] T. Feldman and G. Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *USENIX ;login issue*, 38(3), 2013.

[18] FreeNAS. ZFS and lots of files. https://forums.freenas.org/index.php?threads/zfs-and-lots-of-files.7925/.

[19] S. Ghemawat and J. Dean. LevelDB. https://github.com/google/leveldb.

[20] G. Gibson and G. Ganger. Principles of Operation for Shingled Disk Devices. Technical Report CMU-PDL-11-107, CMU Parallel Data Laboratory, Apr. 2011.

[21] D. Hall, J. H. Marcos, and J. D. Coker. Data Handling Algorithms For Autonomous Shingled Magnetic Recording HDDs. *IEEE Transactions on Magnetics*, 48(5):1777–1781, 2012.

[22] W. He and D. H. C. Du. Novel Address Mappings for Shingled Write Disks. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, pages 5–5, Berkeley, CA, USA, 2014. USENIX Association.

[23] INCITS T10 Technical Committee. Information technology - Zoned Block Commands (ZBC). Draft Standard T10/BSR INCITS 536, American National Standards Institute, Inc., Sept. 2014. Available from http://www.t10.org/drafts.htm.

[24] C. Jin, W.-Y. Xi, Z.-Y. Ching, F. Huo, and C.-T. Lim. HiSMRfs: a High Performance File System for Shingled Storage Array. In *Proceedings of the 2014 IEEE 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, June 2014.

[25] S. Kadekodi, S. Pimpale, and G. A. Gibson. Caveat-Scriptor: Write Anywhere Shingled Disks. In *7th USENIX Workshop on Hot Topics in Storage and File*

Systems (HotStorage 15), Santa Clara, CA, USA, July 2015. USENIX Association.

[26] J. Kaiser, D. Meister, T. Hartung, and A. Brinkmann. ESB: Ext2 Split Block Device. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, ICPADS '12, pages 181–188, Washington, DC, USA, 2012. IEEE Computer Society.

[27] J. Katcher. Postmark: A New File System Benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997.

[28] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux Implementation of a Log-structured File System. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006.

[29] M. Kryder, E. Gage, T. McDaniel, W. Challener, R. Rottmayer, G. Ju, Y.-T. Hsia, and M. Erden. Heat Assisted Magnetic Recording. *Proceedings of the IEEE*, 96(11):1810–1835, Nov. 2008.

[30] A. K. KV, M. Cao, J. R. Santos, and A. Dilger. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium*, volume 1, 2008.

[31] R. Landley. Red-black Trees (rbtree) in Linux. https://www.kernel.org/doc/Documentation/rbtree.txt, Jan. 2007.

[32] D. Le Moal, Z. Bandic, and C. Guyot. Shingled file system host-side management of Shingled Magnetic Recording disks. In *Proceedings of the 2012 IEEE International Conference on Consumer Electronics (ICCE)*, pages 425–426, Jan. 2012.

[33] C. MacCárthaigh. Scaling Apache 2.x beyond 20,000 concurrent downloads. In *ApacheCon EU*, July 2005.

[34] A. Manzanares, N. Watkins, C. Guyot, D. LeMoal, C. Maltzahn, and Z. Bandic. ZEA, A Data Management Approach for SMR. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, USA, June 2016. USENIX Association.

[35] C. Mason. Compilebench. https://oss.oracle.com/~mason/compilebench/.

[36] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, 2007.

[37] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[38] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2014.

[39] K. Muller and J. Pasquale. A High Performance Multi-structured File System Design. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 56–67, New York, NY, USA, 1991. ACM.

[40] NetBSD-Wiki. How to install a server with a root LFS partition. https://wiki.netbsd.org/tutorials/how_to_install_a_server_with_a_root_lfs_partition/.

[41] A. Palmer. SMRFFS-EXT4—SMR Friendly File System. https://github.com/Seagate/SMR_FS-EXT4.

[42] A. Palmer. SMR in Linux Systems. In *Vault Linux Storage and File System Conference*, Boston, MA, USA, Apr. 2016.

[43] PerlMonks. Fastest way to recurse through VERY LARGE directory tree. http://www.perlmonks.org/?node_id=883444.

[44] J. Piernas. DualFS: A New Journaling File System without Meta-data Duplication. In *In Proceedings of the 16th International Conference on Supercomputing*, pages 137–146, 2002.

[45] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, CA, USA, April 2005.

[46] K. Ren and G. Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156, San Jose, CA, USA, 2013. USENIX.

[47] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 1–15, New York, NY, USA, 1991. ACM.

[48] 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.

[49] 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*, USENIX'93, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[50] ServerFault. Doing an rm -rf on a massive directory tree takes hours. http://serverfault.com/questions/46852.

[51] M. Szeredi et al. FUSE: Filesystem in userspace. https://github.com/libfuse/libfuse/.

[52] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A Flexible Framework for File System Benchmarking. *USENIX ;login issue*, 41(1), 2016.

[53] L. Torvalds and P. Zijlstra. __wb_calc_thresh. http://lxr.free-electrons.com/source/mm/page-writeback.c?v=4.6#L733.

[54] T. Ts'o. Release of e2fsprogs 1.43.2. http://www.spinics.net/lists/linux-ext4/msg53544.html, Sept. 2016.

[55] S. C. Tweedie. Journaling the Linux ext2fs Filesystem. In *The Fourth Annual Linux Expo*, Durham, NC, USA, May 1998.

[56] J. Wan, N. Zhao, Y. Zhu, J. Wang, Y. Mao, P. Chen, and C. Xie. High Performance and High Capacity Hybrid Shingled-Recording Disk System. In *2012 IEEE International Conference on Cluster Computing*, pages 173–181. IEEE, 2012.

[57] WDC. My Passport Ultra. https://www.wdc.com/products/portable-storage/my-passport-ultra-new.html, July 2016.

[58] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[59] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.

[60] R. Wood, M. Williams, A. Kavcic, and J. Miles. The Feasibility of Magnetic Recording at 10 Terabits Per Square Inch on Conventional Media. *IEEE Transactions on Magnetics*, 45(2):917–923, Feb. 2009.

[61] Z. Zhang and K. Ghose. hFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 175–187, New York, NY, USA, 2007. ACM.

[62] P. Zijlstra. sysfs-class-bdi. https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-class-bdi, Jan. 2008.

SMaRT: An Approach to Shingled Magnetic Recording Translation

Weiping He and David H.C. Du

Department of Computer Science, University of Minnesota, Twin Cities

Abstract

Shingled Magnetic Recording (SMR) is a new technique for increasing areal data density in hard drives. Drive-managed SMR (DM-SMR) drives employ a shingled translation layer to mask internal data management and support block interface to the host software. Two major challenges of designing an efficient shingled translation layer for DM-SMR drives are metadata overhead and garbage collection overhead.

In this paper we introduce SMaRT, an approach to **S**hingled **Ma**gnetic **R**ecording **T**ranslation which adapts its data management scheme as the drive utilization changes. SMaRT uses a hybrid update strategy which performs in-place update for the qualified tracks and out-of-place updates for the unqualified tracks. Background Garbage Collection (GC) operations and on-demand GC operations are used when the free space becomes too fragmented. SMaRT also has a specially crafted space allocation and track migration scheme that supports automatic cold data progression to minimize GC overhead in the long term.

We implement SMaRT and compare it with a regular Hard Disk Drive (HDD) and a simulated Seagate DM-SMR drive. The experiments with several block I/O traces demonstrate that SMaRT performs better than the Seagate drive and even provides comparable performance as regular HDDs when drive space usage is below a certain threshold.

1 Introduction

Perpendicular magnetic recording technique used by traditional HDDs is reaching its areal data density limit. SMR addresses this challenge by overlapping the neighboring tracks. Assuming the write head is two-track wide in an SMR drive, a write will now impact 2 tracks. That is, writing to a track may destroy the valid data in its adjacent track. Consequently data is preferred to be written onto the tracks in a sequential manner. However, random read is still supported by SMR. In the scope of this paper, for simplicity we assume the write head width is 2 tracks.

Tracks in SMR drives are usually grouped into logical units called "bands". Band size depends on specific designs. There are generally two types of SMR drives: the drive-managed SMR (DM-SMR) drives and the host-managed/host-aware SMR (HM-SMR/HA-SMR) drives. DM-SMR drives, such as the Seagate Archive HDD (8 TB) [4] currently available on the market, maintain a logical block addresses (LBAs) to physical block address (PBAs) mapping layer and therefore provide block interface to the host software such as file systems and databases. As a result, they can be used to replace the traditional HDDs without changes to the upper level applications. On the other hand, HM-SMR and HA-SMR drives are simply raw devices and rely on specially designed upper level applications to interact with the PBAs directly.

Depending on the update strategy, DM-SMR drives can further be classified into in-place update SMR (I-SMR) drives and out-of-place update SMR (O-SMR) drives. To perform an update operation to a previously written track in an I-SMR drive, data on the following tracks has to be safely read out first and then written back to their original positions after the data on the targeted track has been updated. To minimize this overhead, only a few tracks (4 or 5) per band are used to avoid long update propagation in some designs. There will be enough separation between any two adjacent bands called "safety gap" such that writing to the last track of each band will not destroy the valid data in the following band. Importantly, static LBA-to-PBA mappings are possible in I-SMR drives requiring no mapping tables and GC operations. However, a considerable percentage of drive space may be consumed for safety gaps due to small band size. For example, at least 20% of the total space is used as safety gaps if the band size is 4 tracks [10]. Generally, a bigger band size provides better space gain but worse

update performance.

O-SMR drives provide much more space gain by using larger bands or zones. Therefore only a negligible amount of space is used for safety gaps. To perform an update operation in O-SMR drives, the updated data will first be written to a new place and the old data will be invalidated. Those invalidated data must be reclaimed later by GC operations for reusing. A mapping table is required to keep track of these data movements. Therefore, challenges exist for designing efficient O-SMR drives which include the metadata overhead and the GC overhead.

In order for O-SMR drives to be adopted in the current storage systems and used for primary workloads (in addition to cold workloads), metadata overhead and GC overhead must be minimized. In this paper we propose a SMaRT scheme, a track-based shingled magnetic translation layer for DM-SMR drives that supports an address mapping at the block level. SMaRT is motivated by two unique properties of SMR. First, an unused/free track can serve as a safety gap for the preceding track to qualify for in-place updates. Second, different from an invalidated page in Solid State Drives, an invalidated track in SMR drives is essentially a free track and can be immediately reused as long as its next track is free too. Based on this property, SMaRT adopts a hybrid track update strategy which maintains a loop of track invalidating and reusing that minimizes the need of triggering GC operations. GC operations are therefore invoked only when the free SMR drive space becomes too fragmented.

Two major modules are designed in SMaRT in order to fully exploit these two properties. One is a track level mapping table and the other is a space management module. The former supports LBA-to-PBA mapping or block interface to the host software and the latter supports free track allocations and GC operations. During a GC operation, valid tracks are migrated to create bigger contiguous free space. Our design of the space management module also enables SMaRT to support an automatic cold data progression feature which can separate cold data from frequently updated or hot data over time to minimize GC overhead.

We implement SMaRT and compare it to a regular HDD and a simulated Seagate DM-SMR drive described in Skylight [6]. The experiments with several workloads demonstrate that SMaRT performs better than this Seagate DM-SMR drive and nearly as well as a regular HDD.

The remainder of the paper is organized as follows. Section 2 discusses the different layouts for I-SMR drives and O-SMR drives. Some related studies are introduced in Section 3. SMaRT is described in Section 4. Experiments and evaluations are presented in Section 5 and some conclusion is made in Section 6.

2 SMR Layout

SMR drives generally follow the geometry of regular HDDs except the tracks are overlapped. Similar to HDDs, each SMR drive may contain several platters. Physical data blocks are also addressed by Cylinder-Head-Sector (CHS). Since outer tracks are larger than inner tracks, the SMR drive space is divided into multiple zones. Tracks in the same zone have the same size. Each zone can be further organized into bands if needed. A small portion (about 1% to 3%) of the total space is usually used as unshingled random access zone (RAZ) or conventional zone for persistent metadata storage[7, 13, 14].

I-SMR drives and O-SMR drives organize and use the bulk shingled access zone (SAZ) differently. Drive-managed I-SMR drives usually organize the tracks into small bands for a good balance between space gain and performance as discussed and evaluated in [10]. Most existing work on O-SMR drives divide the shingled access zone into an E-region and an I-region. Sometimes multiple E-regions and I-regions may be used. E-region is essentially a persistent cache space organized as a circular log and used for buffering incoming writes, while I-region is used for permanent data storage and organized into big bands. Obviously, writes to E-region and I-region have to be done in a sequential manner and GC operations are required for both regions. The E-region size is suggested to be no more than 3% [7, 8, 9, 14].

3 Related Work

There are a few studies that have been done for I-SMR drives. Shingled file system [13] is a host-managed design for I-SMR where the file system directly works on SMR drive PBAs. The SMR drive space is organized into bands of 64 MB. Files are written sequentially from head to tail in a selected band. He *et al.* proposed several static address mapping schemes for drive-managed I-SMRs [10]. The I-SMR drive space is organized into small bands of four tracks. By changing the order of utilizing the tracks, the new address mapping schemes can significantly reduce write amplification overhead compared to the traditional mapping scheme. However, a non-ignorable percentage of total capacity (about 20%) has to be used as safety gaps between neighbouring bands in order to achieve desired performance.

Several studies have also been done for drive-managed O-SMR drives. For example, Cassuto *et al.* proposed two indirection systems in [8]. Both systems use two types of data regions, one for caching incoming write requests and the other for permanent data storage. They proposed an S-block concept in their second scheme. S-blocks have the same size and each S-block consists of a

pre-defined number of sequential regular blocks/sectors such as 2000 blocks as used in [8]. GC operations have to be performed in both data regions in an on-demand way. Hall *et al.* proposed a background GC algorithm [9] to refresh the tracks in the I-region while data is continuously written into the E-region buffer. The tracks in the I-region have to be sequentially refreshed at a very fast rate in order to ensure enough space in the E-region, which is expensive and creates performance and power consumption issues.

Host-managed O-SMR is another new design trend. Jin *et al.* proposed the HiSMRfs [11] which is a host-managed solution. HiSMRfs pairs some amount of SSD with the SMR drive so that file metadata (hot data) can be stored in the SSD while file data (cold data) can be stored in the SMR drive. HiSMRfs uses file-based or band-based GC operations to reclaim the invalid space created by file deletions and file updates. However, the details of the GC operations are not discussed. Caveat-Scriptor [12] is another host-managed design which protects valid data with a Drive Isolation Distance (DID) and a Drive Prefix Isolation Distance (DPID). It is implemented as SMRfs which is a simple FUSE-based file system. Caveat-Scriptor supports a Free Cleaning scheme to delay and reduce on-demand GC operations. Different from HiSMRfs and Caveat-Scriptor, SMRDB [17] is a host-managed design for key-value store that is file system free. SMRDB defines a set of data access operations including GET/PUT/DELETE/SCAN to comply with the successful KV data access model used in recent cloud storage systems.

The disk drive industry has introduced several openly available SMR drives on the market including the Seagate Archive HDD (8 TB) [4] and the Western Digital Ultrastar Archive Ha10 (10 TB) [5]. These drives are specially designed for cold workloads such as archive systems and backup systems. Aghayev and Desnoyers conducted a series of microbenchmarks and video recording through a drilled hole on the drive shell to reverse engineer Seagate SMR drives [6]. Their observations revealed the presence of an on-disk persistent cache with a lazy write back policy and they further studied other aspects including the cache size and band size. We use these information to implement a simulated Seagate DM-SMR drive in our experiments. Recently, Wu *et al.* conducted evaluations on special features of real HA-SMR sample drives such as open zone issue and media cache cleaning efficiency. They also proposed a host-controlled indirection buffer to improve I/O performance [18].

4 SMaRT Scheme

In this section, we describe the proposed SMaRT scheme and discuss its designs and implementations. There are two major function modules in SMaRT: a track-based mapping table (Section 4.1) that supports LBA-to-PBA mapping and a space management module (Section 4.2) that manages free track allocations and GCs.

4.1 Track Level Mapping Table

The first main function module of SMaRT is a LBA-to-PBA mapping table at a track level which enables SMR drives to communicate with the host software using the block interface. Therefore SMR drives can be used in existing storage systems in a drop-in manner. The mapping table is updated when: 1) A used track is updated to a new location, 2) Used tracks are migrated during GCs or 3) Tracks are allocated for new data.

Given an LBA, SMaRT will first calculate its corresponding logical track number (LTN) and its offset inside this track based on the number of zones and the track size in each zone. It then looks up the mapping table and translates LTN into physical track number (PTN). The final PBA can be easily computed with PTN and the offset inside the physical track.

Assuming an average track size of 1 MB, an 8 TB SMR drive requires at most a 64 MB track level mapping table (assuming 4 bytes for each LTN and PTN). The mapping table is initially empty and gradually grows as the space utilization increases. Therefore, the metadata overhead of the track level mapping table is reasonably low considering the fact that the standard DRAM size for large capacity HDDs/SMRs on the market today is 128 MB [4] or 256 MB [5].

As a comparison, the Seagate DM-SMR drive described in [6] uses a persistent cache (E-region). When adopting a block-level mapping scheme, an E-region of size 80 GB (= 1% of 8 TB) yields a 160 MB mapping table based on 4 KB block size or 1280 MB based on 512 B block size. Alternatively, extent mapping [6] can be used to reduce the mapping table size which however provides less effective persistent cache size to the host writes.

4.1.1 Hybrid Track Update

SMaRT uses a hybrid track update strategy that updates qualified tracks in place and updates the unqualified tracks out of place. Track update has been proven to be beneficial and affordable because it creates a track invalidating and reusing loop. When a track is invalidated, it actually becomes a free track and can be reused as long as its next track is free or as soon as its next track becomes free. As a result, track updates in SMaRT continuously invalidate tracks and turn them into free tracks without triggering explicit on-demand GC operations. This greatly reduces the frequency of invoking

GCs which compensates for the cost of the track update operations. Explicit on-demand GC operations are only invoked when the free SMR space becomes too fragmented as discussed in Section 4.2.4.

4.2 Space Management

The second main component of SMaRT is a space management module which is responsible for free track allocation and garbage collections. Space management is done for each zone separately.

Free track allocation means that when a track is updated, SMaRT has to choose a new track position from the available usable free tracks. The updated track will be written to the new position and the old track is invalidated/freed.

As described previously, usable free tracks and unusable free tracks coexist. Garbage collection is essentially a free space consolidation that can turn unusable free tracks into usable free tracks by migrating the used tracks.

We now introduce a concept of "space element" and a "fragmentation ratio" before describing the details of SMaRT.

4.2.1 Space Elements

There are two types of tracks in an SMR drive: the used (or valid) tracks and the free tracks. When a used track is invalidated, it becomes a free track but it is not considered as a usable free track if its following track is not free. However, a free but unusable track can at least serve as a safety gap and allow its preceding track to be updated in place.

All the used tracks constitute the **used space** and all the free tracks constitute the **free space**. We call a group of consecutive used tracks or free tracks a *space element* which is used to describe the current track usage. For the example of Figure 1, we say the used space includes elements {0, 1, 2, 3}, {6}, {10, 11, 12}, {14, 15, 16, 17} and {20, 21} while the free space includes elements {4, 5}, {7,8,9}, {13}, {18, 19} and {22, 23}. The **size** of a particular space element is defined as the number of tracks in it. The last track in each free space element is not usable and can not be written because writing to this last track will destroy the valid data on the following track which is a used track. Particularly, a free space element of size 1 contains no usable free track such as element {13}. The number of elements and their sizes continuously change as incoming requests are processed. A free track that is previously unusable can become usable later as soon as its following track becomes free too. Accordingly the last track in a used space element can be updated in place because its next track is a free track.

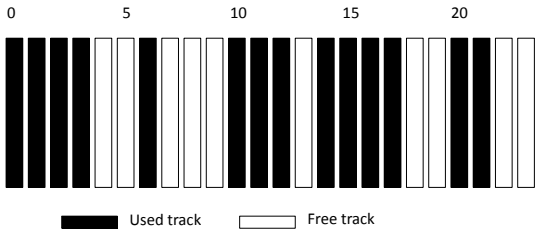


Figure 1: SMR Usage State

4.2.2 Fragmentation Ratio

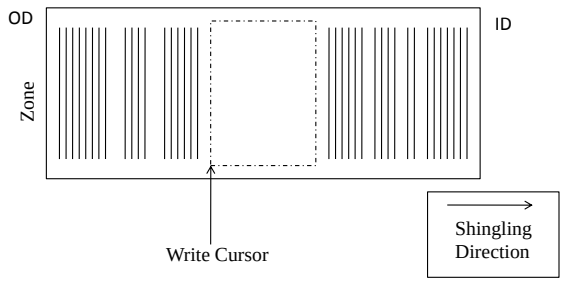
We define the free space fragmentation ratio to help decide when to invoke on-demand GCs in each zone. Assuming the total number of **free tracks** in a selected zone is F and the total number of **free space elements** is N, the free space *fragmentation ratio* (R) for this zone can be computed according to Equation 1. In fact, the fragmentation ratio represents the percentage of usable free tracks in all the free tracks. Fragmentation ratio of 0 means the free space is too fragmented. In fact, 0 means all free space elements are of size 1 and thus no track can be used.

$$R = \frac{F - N}{F}, where\ 1 \leq N \leq F \tag{1}$$

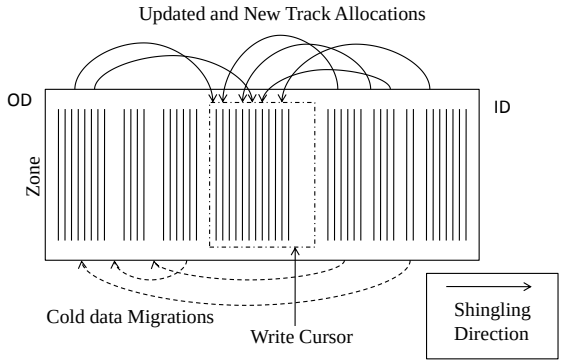
We conduct a series of permutation tests to study the impacts of the fragmentation ratio threshold (R) on SMaRT performance which show that neither a small R threshold nor a large R threshold produces good performance. A small R makes SMaRT adopt a lazy garbage collection strategy. On-demand GC operations are only invoked when the free space is extremely fragmented which requires more victim elements to be migrated in a single GC operation and causes I/O burstiness. A big R ratio is not suggested either since frequent unnecessary GCs will be invoked even though the free space is not fragmented. A larger ratio also means a smaller N and thus a smaller number of tracks that support in-place updates. We use 0.5 in our experiments to allow SMaRT to maintain relatively big contiguous free space and trigger a GC only if necessary.

4.2.3 Space Allocation

SMaRT always designates the largest free space element as an "allocation pool" and maintains a dedicated track pointer called "*write cursor*" that initially points to the first track of the allocation pool, as shown in Figure 2a. The free tracks in this allocation pool are allocated to accommodate updated tracks as well as new write data in a sequential manner in the shingling direction. A modified track is always written to the free track pointed by the



(a) New Allocation Pool



(b) Partially Used Pool

Figure 2: The Process of SMaRT

write cursor which will be incremented accordingly. After the write cursor reaches the end of the allocation pool, SMaRT will select the latest largest free space element as the new allocation pool and update the write cursor to its first track.

Specially, SMaRT defines data that is recently updated as hot. New write (first write) data and data not updated for some time will be treated as cold. For example, the least recently updated track is definitely cold and the most recently updated track is hot. SMaRT will allocate an extra guarding track for a hot data track when the drive utilization is lower than a certain threshold (such as 50%) because there are a sufficient amount of free tracks to be allocated as safety tracks. Therefore these hot data tracks now support in-place updates which reduces unnecessary out-of-place updates. This feature is disabled once the space utilization goes over a threshold and will be re-enabled when space utilization falls below the threshold. The transition is fully transparent.

4.2.4 Space Consolidation

A garbage collection in SMaRT is essentially a free space consolidation. There are two types of GCs in SMaRT: background GCs and on-demand GCs. Background GC operations are only performed during the drive idle times. When a time period between two writes is longer than a threshold value, it is considered as an idle time [6]. Background GC operations keep running until 1) space fragmentation ratio surpasses its threshold or 2) idle time ends. On the other hand, the fragmentation ratio is checked upon each incoming write request. If it is equal to or smaller than the threshold value, an on-demand GC operation will be invoked to migrate used tracks and combine the small free space elements into bigger elements. This will improve I/O performance for big writes and updates, as well as increase usable free space. An on-demand GC usually immediately stops after one element has been moved so as to minimize the overhead of a single GC operation. This minimizes the performance interference on serving the incoming requests. The fragmentation ratio sometimes remains below its threshold after the current GC operation. The next GC operation in this zone will continue to improve the fragmentation. Moving multiple elements happens only when there is not enough usable free tracks to accommodate the updated track(s) or new tracks.

To perform a GC operation, SMaRT searches for a victim element starting from the leftmost used space element of this zone and proceeding in the shingling direction. A victim element has a size of smaller than a threshold W. W is initialized to be a small value based on the current SMR drive space usage U. It can be calculated according to Equation 2. In fact, W practically represents the used space to free space ratio. The theory behind this equation is that the average used space element size is bigger when the SMR space utilization is higher. For example, W will be initially set to 2 if the current SMR drive usage is 60% or 9 if the usage is 90%. If no element is found to be smaller than W, W will be doubled and SMaRT will redo a search based on the new W. This will be repeated until a satisfying element is found.

$$W = \frac{U}{1-U}, where\ 0 < U < 1 \tag{2}$$

SMaRT always tries to append a victim element to the leftmost free space element that fits the victim element[1]. If no free space to the left can accommodate the victim element, SMaRT simply shifts it left (against shingling direction) and appends it to its left used space element neighbour. Besides, if the victim element resides in the allocation pool, it will also be appended to the left neighbour because this victim element contains recently written/updated tracks and thus is deemed as possibly hot which should not be appended to cold data.

In Figure 1, R is 0.5 by Equation 1 and W is 2 by Equation 2. Assuming an on-demand GC is triggered, SMaRT will select the used space element {6} as the victim and

[1]The used space element containing track 0 is a special case.

append it to element {0, 1, 2, 3}. Consequently, the free space elements {4, 5} and {7, 8, 9} will be consolidated into a single bigger element {5, 6, 7, 8, 9}. The resulting fragmentation ratio R is 0.6. SMaRT will detect upon the next request that R is above the threshold and thus it will not invoke another GC operation.

4.2.5 Cold Data Progression

The track-based mapping and data migrations provide a good opportunity of automatic cold data progression. This is achieved as part of the free track allocation and GC operations, requiring no dedicated hot/cold data identification algorithms. The cold data progression in SMaRT is illustrated in Figure 2b. The allocation pool accumulates the recently updated tracks or hot tracks. Cold data gets migrated against the shingling direction to the left by GC operations. Eventually the cold data will mostly stay at the left side of the zones and hot data gets updated and pushed to the right side of the zones which reduces unnecessary cold data movements during GC operations.

The downside of this is that cold data (least recently updated data) will eventually reside on the outer tracks which possibly wastes the sequential I/O performance of the outer tracks. We provide one argument and one solution to this concern. We argue that cold data can be frequently read by our definition. Least recently updated data can be most frequently read data and/or most recently read data.

To directly address this concern, we can reverse the shingling direction by shingling from ID (Inner Diameter) to OD (Outer Diameter) so that writes start with inner tracks and move toward outer tracks. This way, cold data will be gradually moved to the inner tracks. Note that we assume the normal shingling direction in our experiments.

4.3 I/O Processing Summary

The overall I/O procedure for read requests is straightforward. SMaRT simply reads the data after translating the LBAs into PBAs. Multiple reads will be issued if read is fragmented.

On receiving a write request, SMaRT first checks the fragmentation ratio to decide if a GC operation should be invoked. After a necessary GC operation completes, SMaRT checks whether the write request operates on existing data by consulting the mapping table. For new data, SMaRT will allocate free tracks and add corresponding new mapping entries. For existing data, SMaRT checks whether a track supports in-place update. If not, SMaRT allocates new tracks and updates the existing mapping entries.

During the workload, once an idle time is identified and meanwhile the free space fragmentation ratio is below the threshold, background GCs will be launched.

4.4 SMaRT for Cold Write Workload

SMaRT performs extremely like a regular HDD for cold workloads such as backup, archive and RAID rebuild workloads. Data will be written sequentially into the allocation pool without extra guarding safety tracks because new write data is treated as cold. Data can still be updated based on the hybrid update strategy if ever needed. No changes to the space management scheme are needed.

4.5 Reliability

The mapping table of SMaRT is periodically synchronized to a persistent mapping table copy in the random access zone on disk. However, there is still a risk that a power failure occurs before the latest mapping table updates are pushed to the disk. This is a common reliability challenge to flash translation layers (FTLs) and shingled translation layers (STLs). Here we sketch a low-cost scheme for SMaRT inspired by a *Backpointer-Assisted Lazy Indexing*[15].

Since SMaRT always writes new tracks and updated tracks to the allocation pool, SMaRT can simply trigger a synchronization whenever an allocation pool is fully consumed and flush the mapping table updates to the disk. Upon flush completion, SMaRT records the timestamp and picks a new allocation pool. Tracks updated after the latest synchronization are ensured to reside in the latest allocation pool only. When writing a new track or updating an existing track to a new position, a backpointer to the logical track number (LTN) along with the current timestamp will be stored together with the track such that each physical track internally has a PTN-to-LTN reverse mapping entry. We assume there will be some tiny spare space associated with each physical track that can be used to store the LTN and the timestamp. Otherwise, we can simply reserve a sector or block in each physical track to be used for storing these extra information.

To recover from a power failure, only the latest allocation pool needs to be scanned instead of the whole disk. The synchronization interval can also be the time to consume a list of most recent allocation pools. In this case, the manufacturing cost of SMR drives is minimal because no extra media is introduced. To perform a recover, SMaRT scans the latest allocation pool and identifies tracks with timestamps newer than that of the latest synchronization. SMaRT then reads their associated LTNs and PTNs to construct the corresponding LTN-to-PTN mapping entries which will be merged to the map-

ping table copy on disk so that the latest LTN-to-PTN table will be restored.

Alternatively, Non-Volatile Memory and flash media can be used to persist the mapping table when the cost of the former and the durability of the latter become acceptable.

5 Evaluations

In this section, we evaluate the performance of SMaRT and make comparisons with other schemes.

5.1 Competing Schemes

We compare SMaRT to a regular HDD and a simulated Seagate DM-SMR drive. Since our objective is to design SMR drives that can perform well under primary workloads in existing storage systems, we choose the regular HDD as the baseline for comparison. We are hoping the performance of the new design can be close to that of HDDs.

We also implement a Seagate DM-SMR drive based on the configurations and schemes explored and described in Skylight [6]. We denote this SMR drive as **Skylight** which is configured with a single 2 GB persistent cache at OD, 40 MB band size, static address mapping for bands and aggressive GC operation with a 1242ms triggering idle window. Incoming writes go to the cache first. A Skylight background GC would be triggered if there is a 1242ms idle time window in the workload since the last request. It keeps running until the next request comes in. A GC operation scans from the tail of the circular-log structured persistent cache, reads all the sibling blocks that belongs to the same band as the tail block and read-modify-writes the band. In our experiments, we find that on-demand GCs are also needed when there are not enough idle times in the workloads. An on-demand GC is triggered when the persistent cache space is 80% full.

5.2 Implementations

Due to the needs of defining zone sizes and controlling the starting/ending PBAs of the zones, we have to adopt simulation instead of using real hard disk drives. We implement these schemes on top of Disksim [2] to simulate an SMR drive based on the parameters of a Seagate Cheetah disk drive [1]. This is the newest and largest validated disk model that is available to Disksim. It has a capacity of 146 GB (based on 512 B sector size) [2]. We divide the total capacity into 3 parts: one 2 GB random access zone, one 2 GB perstistent cache (i.e., E-region) and the rest of the 142 GB for persistent storage space (i.e., I-region). The random access zone and the E-region will not be allocated if they are not used in a specific scheme. Particularly, both the random access zone and the E-region are not used in HDD. And the E-region is not used in SMaRT. On receiving disk I/O requests (in the form of LBAs), these schemes will translate the block addresses into PBAs based on their own implementation logic. The translated requests (in the form of PBAs) are then passed to the Disksim for processing.

[2]The drive capacity is about 1.1 TB based on 4 KB sector size.

Table 1: Trace Statistics

Trace	MAX LBA	factor	Req. Size	Write%
mds_0	71,127,264	5	18	88.11%
proj_0	34,057,712	10	76	87.52%
stg_0	22,680,944	12	23	84.81%
rsrch_0	35,423,624	10	17	90.67%

5.3 Trace Information

Four write intensive MSR traces [3, 16] are used in our experiments since other read intensive traces may not trigger media cache cleaning. The characteristics of the MSR traces are shown in Table 1 which include the maximum LBA, the average request size (R.S.) in blocks and the write ratio. We scale up the LBAs in the four traces to make sure the whole 142GB space is covered. Based on the maximum LBA, a different scale up factor is used for each trace.

5.4 Experiment Design

We test the schemes at different drive space utilizations including 30%, 60% and 90% which allow us to understand the impact of space utilizations on the drive performance. The drives will be pre-populated with data according to the utilization which is done by manipulating the metadata or mapping table in each scheme. Oversized LBAs in the traces will be trimmed with modular operation to fit in the accessed LBA range.

5.5 Performance Comparison

We use *response time*, *Write Amplification*, *Read Fragmentation* and *GC Overhead* as the main performance comparison metrics.

5.5.1 Response Time

The response time for each request is calculated by subtracting the queueing time stamp from the request completion time stamp. The overall average response times

for different schemes under different workloads at different drive space utilizations are shown in Figures 3, 4 and 5. X-axis is the average response time and Y-axis is the corresponding Cumulative Distribution Function (CDF).

The results show that in general HDD performs best for all utilizations. SMaRT has better performance than that of Skylight for the 30% and 60% utilizations and the two schemes provide a comparable performance for the 90% utilization. The Skylight only shortly crosses over the HDD and SMaRT in some of the low response time ranges while lags behind for the majority of the response times. The cross-overs are mainly contributed by more physically sequential writes due to the using of persistent cache.

5.5.2 Read Fragmentation

SMaRT suffers read fragmentation because of track movements. A read request can span over multiple logically consecutive but physically scattered tracks. Skylight incurs read fragmentation because of persistent cache. Some blocks in a read request may exist and scatter inside the persistent cache while the rest blocks still reside in the bands. As a result, a single read request can be fragmented into several smaller requests in both schemes, although the total number of blocks accessed remains the same.

We first show the percentages of fragmented reads in both schemes in Figure 6 and then compare the read fragmentation ratio in Figures 7, 8 and 9. We define the read fragmentation ratio as the number of "smaller read requests" produced by a single fragmented read request. Theoretically, the ratio is upper-bounded by the number of tracks accessed by the read request for SMaRT while bounded by the number of blocks in the request for Skylight. This is also the reason why fragmented read percentages in SMaRT are less affected by the space utilizations as seen in the figure.

As expected, the result shows that SMaRT has higher fragmented read percentages (Figure 6) with lower fragmentation ratios (Figures 7, 8 and 9). In general, more than 95% of the fragmented reads have a fragmentation ratio of 3 for SMaRT.

5.5.3 Write Amplification

We define *amplified write percentage* as the percentage of the write requests that trigger on-demand GCs outside the idle times. The result is shown in Figure 10.

Note that the amplified write percentages for 30% are all zero for SMaRT. No GC operation is incurred because the space allocation scheme in SMaRT will allocate guarding safety tracks when space utilization is less than 50% which suffices to maintain a good free fragmentation ratio so as to avoid triggering on-demand GCs.

Table 2: GC Statistics

Traces	SMaRT		Skylight	
30%	GC_F	GC_B	GC_F	GC_B
mds_0	0	0	1588	118417
proj_0	0	0	1861	97314
stg_0	0	0	1823	72229
rsrch_0	0	0	2003	108665
60%	GC_F	GC_B	GC_F	GC_B
mds_0	2045	603	1840	118417
proj_0	1560	536	2278	97314
stg_0	2583	1203	2191	72229
rsrch_0	3544	789	3099	108665
90%	GC_F	GC_B	GC_F	GC_B
mds_0	18367	5226	1866	118417
proj_0	13040	4595	2646	97314
stg_0	50131	23029	2834	72229
rsrch_0	35416	9722	3921	108665

However, as space utilization increases to 60% and 90%, up to 6% of the write requests will trigger on-demand GCs compared to 0.5% for Skylight. These percentage numbers are low mainly because of the contributions of background GCs.

5.5.4 GC Overhead

As described previously, there are two types of GC operations in both schemes: on-demand GCs (GC_F) and background GCs (GC_B). Table 2 shows the average numbers of on-demand GCs and background GCs incurred per 1 millions write requests. The table covers different traces and space utilizations.

In general, these numbers increase as the space utilization climbs. SMaRT triggers more on-demand GCs and Skylight triggers more background GCs. The fact that SMaRT relies more on on-demand GCs makes it theoretically more suitable for workloads with less idle times.

Specially, the numbers of both the on-demand GCs and the background GCs are 0s for the 30% utilization. SMaRT also uses a 1242ms idle time window as the first background GC triggering condition and the free space fragmentation ratio as the second condition. A background GC would be launched when both conditions are met and stops when the free space fragmentation ratio goes above the threshold or when the next write arrives. As a result, the 0s are simply because the free space fragmentation ratio stays high enough that no GC is needed.

Additionally, the numbers of background GCs for Skylight stay the same across different utilizations because Skylight starts background GCs when a 1242ms idle window is detected and stops when the next request arrives which is not affected by the utilization changes.

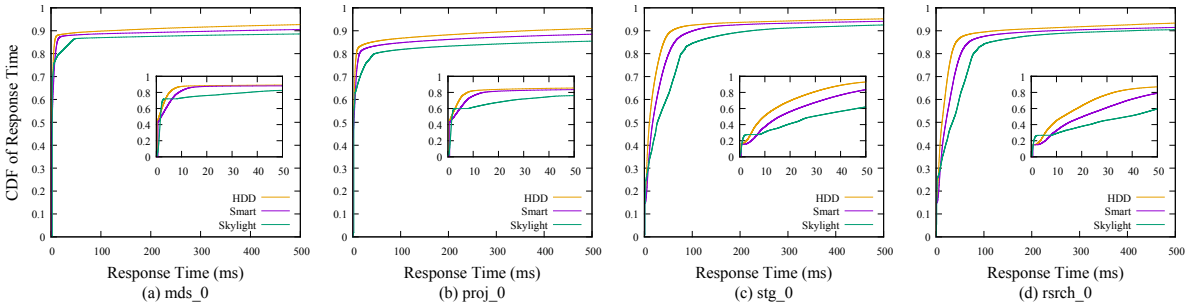


Figure 3: CDF of Response Time at 30% Utilization

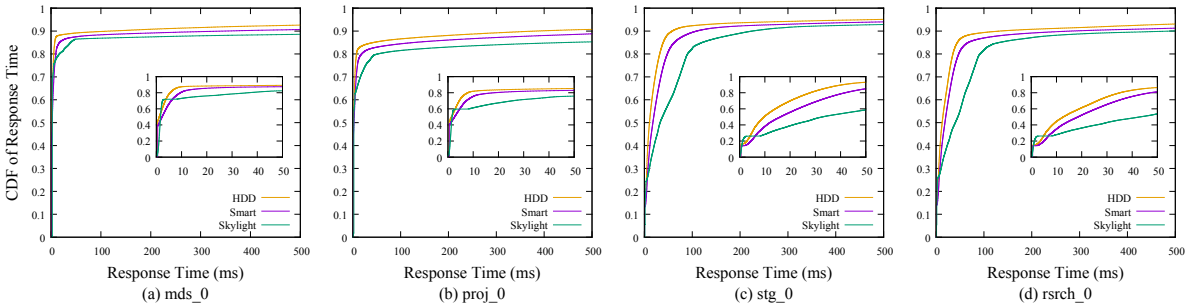


Figure 4: CDF of Response Time at 60% Utilization

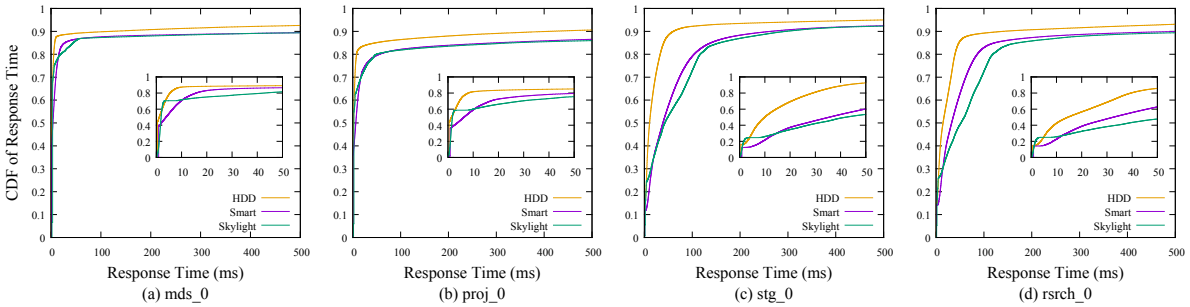


Figure 5: CDF of Response Time at 90% Utilization

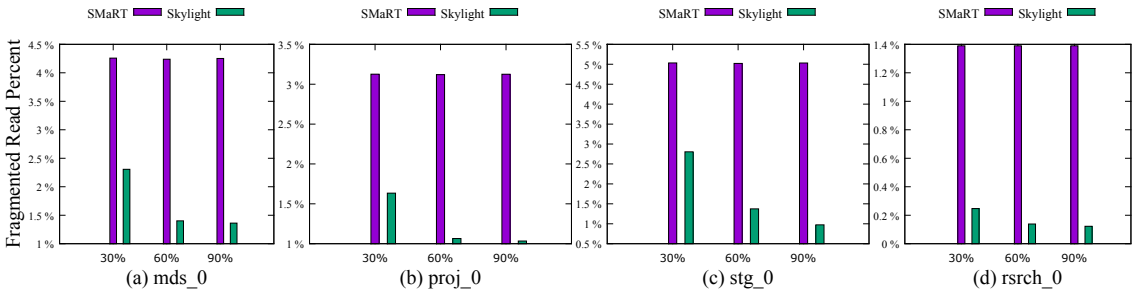


Figure 6: Fragmented Read Percent

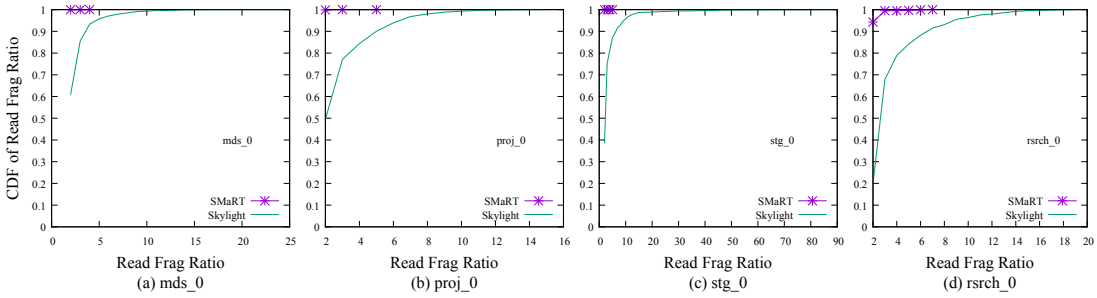


Figure 7: CDF of Read Frag Ratio at 30% Utilization

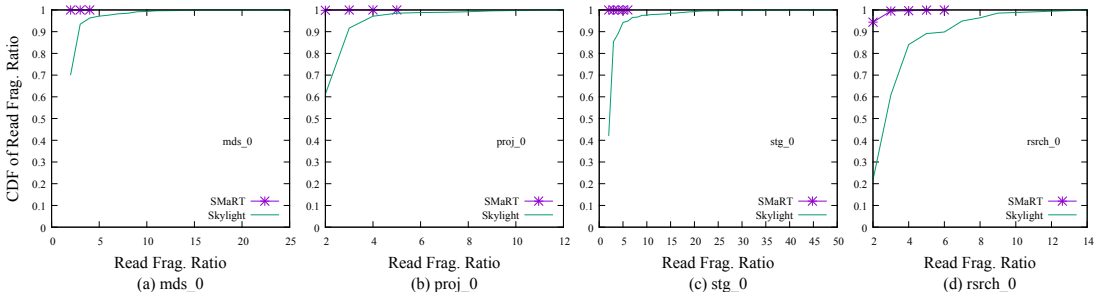


Figure 8: CDF of Read Frag Ratio at 60% Utilization

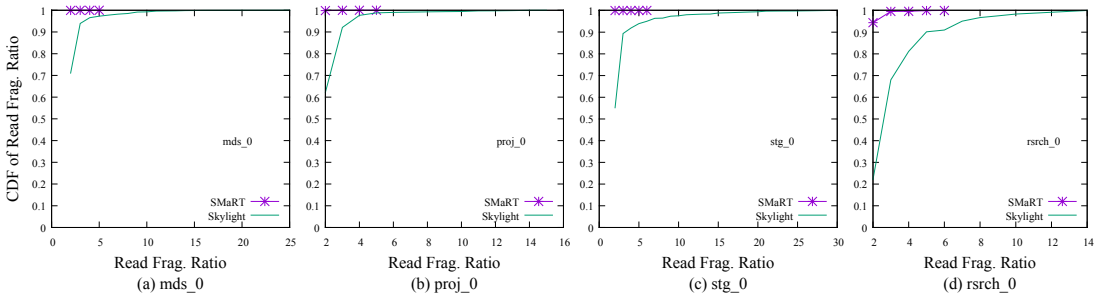


Figure 9: CDF of Read Frag Ratio at 90% Utilization

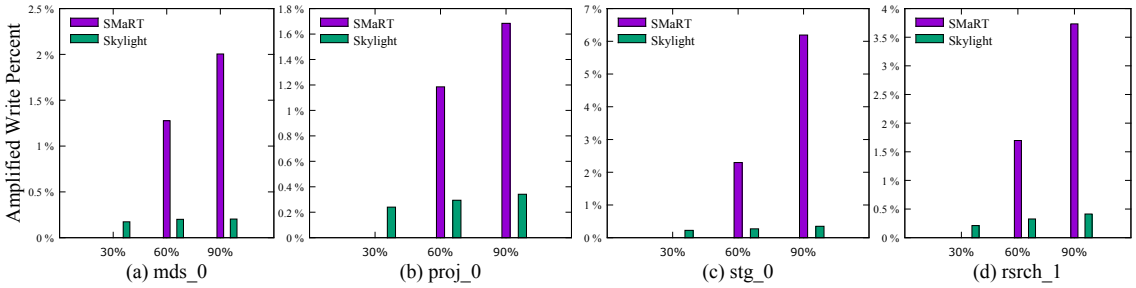


Figure 10: Amplified Write Percent

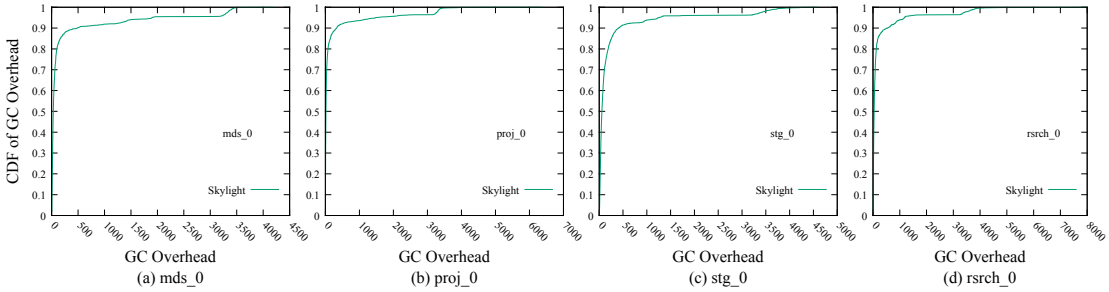


Figure 11: CDF of GC Overhead at 30% Utilization

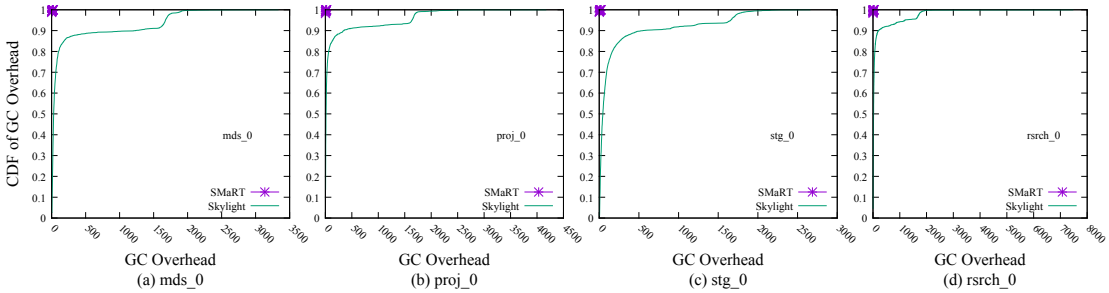


Figure 12: CDF of GC Overhead at 60% Utilization

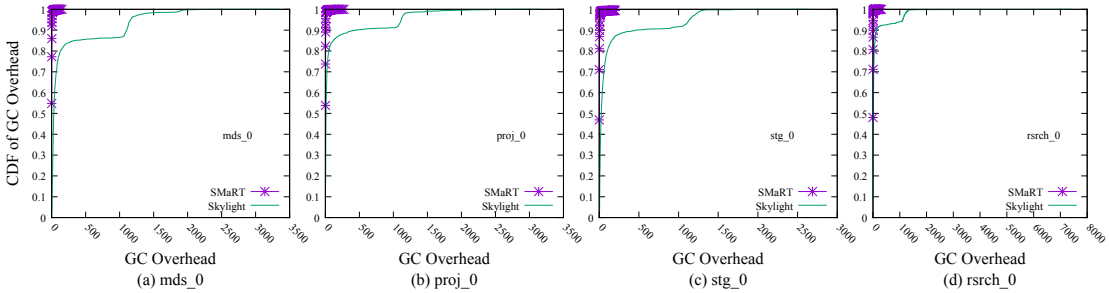


Figure 13: CDF of GC Overhead at 90% Utilization

Table 3: Scheme Comparison Summary

Schemes	E-region?	Performance	Cold Workload Friendly	Metadata Overhead	Space Gain
HDD	no	very good	yes	no	100%
SMaRT	no	good	yes	low	>95%
Skylight	yes	fair	no	high	>95%

Since we have shown the GC counting, we now discuss the on-demand GC overhead shown in Figures 11, 12 and 13. GC overhead (X-axis) is defined differently for SMaRT and Skylight because of the different natures of their GCs, although the two schemes share the X-axis. It is defined as the number of tracks migrated during a GC operation for SMaRT while defined as the total number of I/O requests needed for a single GC operation for Skylight. So there is no direct performance comparison here. Instead, this serves as helping understand the overhead origination in each GC scheme. About 50% of SMaRT's on-demand GCs move a single track and about 99% of the GCs move a single-digit number of tracks. On the other hand, Skylight's on-demand GCs have a wide overhead spectrum. However, about 80% of its on-demand GCs consists of less than 100 I/O requests.

5.5.5 Comparison Summary

We now summarize the comparison results in Table 3. The schemes are compared according to different metrics including performance, cold write workload suitability and metadata overhead. SMaRT is the preferred design which achieves a good balance among the listed metrics. It produces better performance than that of Skylight for low-range and mid-range space utilizations. Besides, SMaRT is more friendly to cold workloads because it does not stage incoming data in a persistent cache and destage later as Skylight does. SMaRT also has a lower metadata overhead because of not using persistent cache.

6 Conclusion and Future Work

In this paper we propose an efficient SMaRT scheme for drive-managed SMR drives by exploiting two unique properties in the SMR drives. SMaRT performs copy-on-write updates only when in-place updates are impossible. The track level mapping, when combined with a novel space management scheme can automatically filter the cold data to minimize data migration overhead for GC operations. The Experiments with real world workloads demonstrate that SMaRT can perform as well as regular HDDs under cold write workloads and even under primary workloads when space usage is in the lower and middle ranges.

In the future, we plan on designing a request scheduling algorithm that bundles multiple write requests as a single request if they belong to the same track or adjacent tracks. This would further reduce the write amplification overhead and improve write performance especially for high disk utilization situations. We also plan to extract parameters from the latest SMR drives and validate the resulting drive model for further simulation purposes. We plan to also investigate the performance of SMaRT when the write head width is more than 2 tracks.

Acknowledgement

We thank the anonymous reviewers and our shepherd Erik Riedel for their insightful comments on earlier drafts of the work. This work was partially supported by NSF awards 130523, 1439622, and 1525617.

References

[1] Database of Validated Disk Parameters. `http://www.pdl.cmu.edu/DiskSim/diskspecs.shtml`.

[2] DiskSim. `http://www.pdl.cmu.edu/DiskSim/`.

[3] MSR Cambridge Block I /O Traces. `http://iotta.snia.org/traces/388`.

[4] Seagate Archive HDD. `http://www.seagate.com/products/enterprise-servers-storage/nearline-storage/archive-hdd/`.

[5] WD UltraStar Ha10. `http://www.hgst.com/products/hard-drives/ultrastar-archive-ha10`.

[6] A. Aghayev and P. Desnoyers. Skylight—a window on shingled disk operation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 135–149, Santa Clara, CA, Feb. 2015. USENIX Association.

[7] A. Amer, D. D. Long, E. L. Miller, J.-F. Paris, and S. Schwarz. Design issues for a shingled write disk system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–12. IEEE, 2010.

[8] Y. Cassuto, M. A. Sanvido, C. Guyot, D. R. Hall, and Z. Z. Bandic. Indirection systems for shingled-recording disk drives. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–14. IEEE, 2010.

[9] D. Hall, J. H. Marcos, and J. D. Coker. Data handling algorithms for autonomous shingled magnetic recording hdds. *Magnetics, IEEE Transactions on*, 48(5):1777–1781, 2012.

[10] W. He and D. H. Du. Novel address mappings for shingled write disks. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*. USENIX Association.

[11] C. Jin, W.-Y. Xi, Z.-Y. Ching, F. Huo, and C.-T. Lim. Hismrfs: A high performance file system for shingled storage array. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–6. IEEE, 2014.

[12] S. Kadekodi, S. Pimpale, and G. A. Gibson. Caveat-scriptor: write anywhere shingled disks. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems*, pages 16–16. USENIX Association, 2015.

[13] D. Le Moal, Z. Bandic, and C. Guyot. Shingled file system host-side management of shingled magnetic recording disks. In *Consumer Electronics (ICCE), 2012 IEEE International Conference on*, pages 425–426. IEEE, 2012.

[14] C.-I. Lin, D. Park, W. He, and D. H. Du. H-swd: Incorporating hot data identification into shingled write disks. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 321–330. IEEE, 2012.

[15] Y. Lu, J. Shu, W. Zheng, et al. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *FAST*, pages 257–270, 2013.

[16] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[17] R. Pitchumani, J. Hughes, and E. L. Miller. Smrdb: key-value data store for shingled magnetic recording disks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 18. ACM, 2015.

[18] F. Wu, M.-C. Yang, Z. Fan, B. Zhang, X. Ge, and D. H. Du. Evaluating host aware smr drives. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, 2016.

Facilitating Magnetic Recording Technology Scaling for Data Center Hard Disk Drives through Filesystem-level Transparent Local Erasure Coding

Yin Li[⋆]*, Hao Wang*[⋆]*, Xuebin Zhang*[†]*, Ning Zheng*[*]*, Shafa Dahandeh*[†]*, and Tong Zhang*[⋆]
[⋆]*ECSE Department, Rensselaer Polytechnic Institute, USA*
[†] *EMC/DSSD, USA* [*] *ScaleFlux, USA* [†] *Western Digital (WD), USA*

Abstract

This paper presents a simple yet effective design solution to facilitate technology scaling for hard disk drives (HDDs) being deployed in data centers. Emerging magnetic recording technologies improve storage areal density mainly through reducing the track pitch, which however makes HDDs subject to higher read retry rates. More frequent HDD read retries could cause intolerable tail latency for large-scale systems such as data centers. To reduce the occurrence of costly read retry, one intuitive solution is to apply erasure coding locally on each HDD or JBOD (just a bunch of disks). To be practically viable, local erasure coding must have very low coding redundancy, which demands very long codeword length (e.g., one codeword spans hundreds of 4kB sectors) and hence large file size. This makes local erasure coding mainly suitable for data center applications. This paper contends that local erasure coding should be implemented transparently within filesystems, and accordingly presents a basic design framework and elaborates on important design issues. Meanwhile, this paper derives the mathematical formulations for estimating its effect on reducing HDD read tail latency. Using Reed-Solomon (RS) based erasure codes as test vehicles, we carried out detailed analysis and experiments to evaluate its implementation feasibility and effectiveness. We integrated the developed design solution into *ext4* to further demonstrate its feasibility and quantitatively measure its impact on average speed performance of various big data benchmarks.

1 Introduction

Flash-based solid-state data storage has been rapidly displacing magnetic recording hard disk drives (HDDs) in traditional market segments, e.g., consumer and personal computing, and tier-1 (and even tier-2) enterprise storage. Although this trend will inevitably continue in the advent of 3D NAND flash memory, it by no means proclaims a doomed future for HDD technology. The blossoming data center and cloud-based infrastructure provide a new area with tremendous growth potential for HDDs. Nevertheless, as pointed out by a well-received position paper by Google Research [6], such a paradigm shift demands fundamental re-thinking on the design of HDDs and system architecture. Authors of [6] shared a collection view on open research opportunities and challenges for the new era of *data center HDDs*.

This paper studies the design of data center HDDs with the focus on leveraging workload characteristics to facilitate magnetic recording technology scaling. In particular, we are interested in how one could alleviate the HDD areal density vs. read retry rate conflict. As conventional magnetic recording technology approaches its limit at around 1Tb/in^2, the industry is now exploring a variety of new technologies including heat assisted magnetic recording (HAMR) [23,26], shingled magnetic recording (SMR) [15, 17], and two-dimensional magnetic recording (TDMR) [13,24,28]. All these new technologies improve bit areal density by shrinking the track pitch (i.e., the distance between adjacent tracks). However, due to the mechanical rotating nature of HDDs, a smaller track pitch inevitably makes HDDs more sensitive to head offset (i.e., the read/write head flies off the center of the target track). This leads to a higher probability of read retry. As a result, future magnetic recording technologies are increasingly subject to the conflict between areal density and read retry rate. The long latency penalty of HDD read retry may cause intolerable tail latency for data centers [6, 8].

Aiming to alleviate the areal density vs. read retry rate conflict for data center HDDs, this work is motivated by a very simple fact: if the data being stored in HDDs have inherent redundancy for error correction, we may recover the failed sectors through system-level error correction (at tens or hundreds of μs latency) other than HDD read retry (at tens or hundreds of ms latency). One may argue that RAID and distributed erasure coding already conve-

niently provide such a feature. However, RAID is being replaced by distributed erasure coding in data centers, and using distributed erasure coding to mitigate HDD read retry can cause significant overheads (e.g., network traffic), in particular under high HDD retry rates (e.g., 10^{-4} and above). This work focuses on the scenarios of storing user data and associated coding redundancy locally together in one HDD or JBOD (just a bunch of disks) and recovering failed sectors locally at the server that directly connects to the HDDs. It is referred to as *local erasure coding* in this work. We note that local erasure coding only aims to reduce the occurrence of costly HDD read retry operations and does not provide any guarantee for tolerating catastrophic HDD failures. Hence, it is completely orthogonal to distributed erasure coding, and must have very small coding redundancy.

A *soft sector read failure* occurs when an HDD fails to decode one sector during its normal operation. Let p_h denote the soft sector read failure probability, which is HDD read retry rate in current practice, i.e., the probability that an HDD switches from the normal operation mode into a retry mode to repeatedly read the failed sector by adjusting the read head position. Current HDDs keep p_h relatively low (e.g., 10^{-6} and below). The probability that one sector cannot be correctly read even after the long-latency HDD read retry is called hard sector failure rate, which must be extremely low (e.g., 10^{-14} and below). When using the local erasure coding, HDDs do not immediately switch to the retry mode upon a soft sector read failure, instead we first try to fix it through the local erasure decoding. Only when the erasure decoding fails, HDDs switch into the read retry mode. Let p_s denote the probability that the local erasure decoding fails. We should minimize the coding redundancy and meanwhile ensure $p_s << p_h$. In order to minimize the local erasure coding redundancy, we must use long codeword length. Therefore, local erasure coding should be applied to systems with dominantly large flies, e.g., data centers.

In spite of the simple basic idea, its practical realization is non-trivial. The first question is whether the local erasure coding should be implemented at the application layer, OS layer, or inside HDD. As elaborated later in Section 2.2, we believe that it should be implemented by the local filesystem at the OS layer with complete transparency to the upper application layer. Integrating local erasure coding into filesystem is far beyond merely implementing a high-speed encoding/decoding library, and one has to modify and enhance the filesystem architecture. This work uses the journaling filesystem *ext4* as a test vehicle to study the integration of local erasure coding into the filesystem. In particular, we investigate the separate treatment of data and filesystem metadata, and develop techniques to handle scenarios when the HDD write is unaligned with the erasure codeword boundary and when fine-grained data update occurs. Meanwhile, we derive mathematical formulations for estimating its effectiveness on reducing HDD read tail latency.

We carried out a variety of analysis and experiments to study the effectiveness and feasibility of the proposed design solution, where we use Reed-Solomon (RS) codes as local erasure codes. Our analysis results show that RS codes with the coding redundancy of less than 2% can reduce the 99-percentile latency by more than 65% when p_h is 1×10^{-3}. Since the local erasure encoding and decoding can add noticeable extra latency into the data I/O path, it will degrade the average system speed performance compared with the ideal retry-free scenario. To evaluate such impact and meanwhile further demonstrate the practical feasibility of filesystem-level transparent local erasure coding, we integrate the proposed design solution into Linux kernel 3.10.102 I/O path (in particular the VFS and *ext4*), and carried out experiments using the big data benchmark suite HiBench 3.0 [1]. Motivated by the emergence of data center CPUs with built-in FPGA (e.g., the Xeon CPU with built-in FPGA as lately announced by Intel), we investigated the use of both software-based and hardware-based RS coding engine. The measurement results show that, even under p_h of 10^{-3}, local erasure coding only incurs (much) less than 3.5% average speed performance degradation. Finally, we also present results on the storage capacity overhead under various benchmarks in HiBench 3.0, and the latency overhead induced by fine-grained update.

2 Background and Rationale

2.1 Magnetic Recording Technologies

To move the storage areal density beyond 1Tb/in^2 and towards 10Tb/in^2 over the next decade, the HDD industry is exploring several new technologies including HAMR, SMR, and TDMR, all of which improve areal density through significantly reducing the track pitch. A smaller track pitch results in a stronger inter-track interference (ITI) and hence worse signal-to-noise ratio, which makes HDDs more sensitive to read head off-set. This can be illustrated in Fig. 1. The read head off-set is defined as the distance between the target track center and head center. The perfect head-track alignment (i.e., zero read head off-set) corresponds to the minimal ITI and hence best read channel signal processing performance, leading to the minimal sector read failure probability. Nevertheless, the mechanical disk rotation inevitably causes run-time fluctuation of the read head position. As shown in Fig. 1, the same read head off-set induces stronger ITI from neighboring track (i.e., track N-1 in the figure) in HDDs with a smaller track pitch. A stronger ITI directly results in worse read channel signal processing perfor-

mance and hence a higher sector read failure probability. Therefore, regardless to the specific magnetic recording technology, HDDs are fundamentally subject to areal density vs. read retry rate conflict.

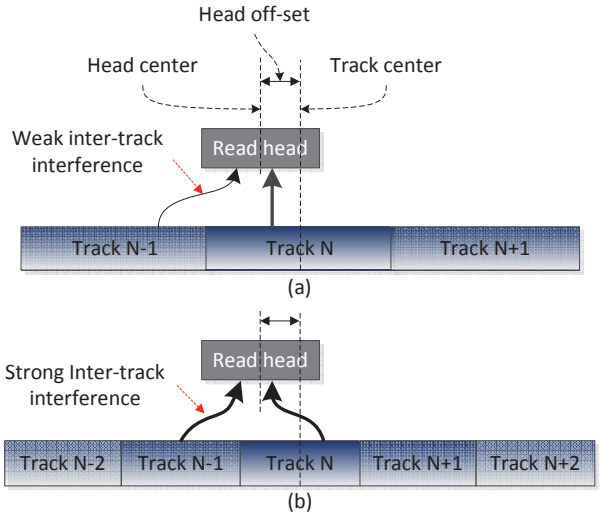


Figure 1: Illustration of different effect of head off-set under (a) large track pitch and (b) small track pitch, where the track-N is the target track being read.

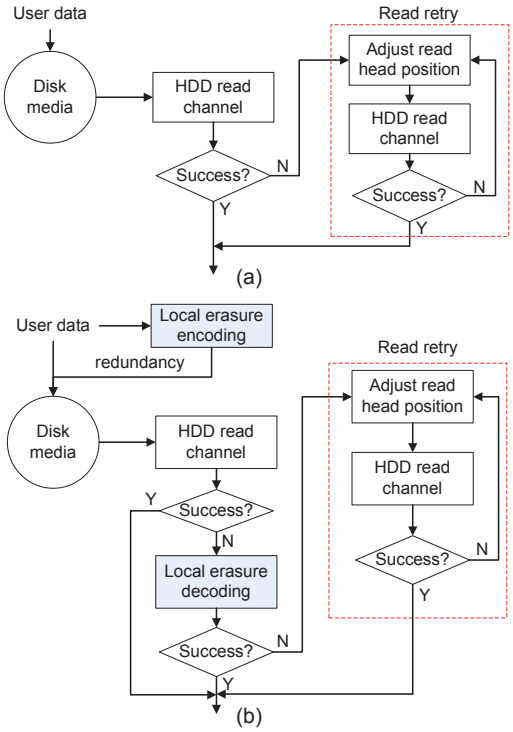


Figure 2: Illustration of (a) current practice handling sector read failures, and (b) using the local erasure coding to reduce the occurrence of read retry.

2.2 Local Erasure Coding

Local erasure coding aims to reduce the occurrence of HDD read retry by storing additional coding redundancy together with the original user data in the same HDD or JBOD. Fig. 2 illustrates the simple basic concept: In current practice, once the read channel signal processing fails during the normal operation (most likely due to runtime read head off-set), HDDs switch into the read retry mode to repeatedly read the failed sector by adjusting the read head position. When using local erasure coding, the user data and associated coding redundancy are stored together locally. Upon a soft sector read failure, we first try to recover the data through the local erasure decoding, and invoke HDD read retry only if the local erasure decoding fails, as shown in Fig. 2(b).

Along the data storage hierarchy, we could implement the local erasure coding at either application layer, OS layer, or hardware layer inside HDDs. With the full knowledge about their own data access characteristics, applications can best optimize the use of local erasure coding. Nevertheless, the efforts of integrating/optimizing the local erasure coding in each application may not be justified in practice. Moreover, not all the data being stored on HDDs are visible to applications (e.g., filesystem metadata). On the other hand, although intra-HDD implementation keeps the software stack completely intact, it has several problems: (1) Because filesystem metadata have fine-grained access characteristics, we should configure the local erasure coding differently for metadata and data. However, HDD on its own cannot distinguish between data and metadata. (2) HDDs are not aware of the validity of data in each sector (i.e., whether or not the data are being used by the filesystem), which could cause a large number of unnecessary local erasure encoding operations and disk rotations, especially under long codeword length. (3) Local erasure coding could leverage data cached in memory to reduce the disk access. HDDs typically have very small internal cache memory, especially compared with the host DRAM. As a result, intra-HDD realization of local erasure coding is subject to more disk rotations.

In comparison, OS (in particular filesystem) can the most appropriate place to implement the local erasure coding. On one hand, filesystem-based implementation is completely transparent to the upper application layers, which lowers the barrier of deploying the local erasure coding in real systems. On the other hand, with the full awareness and control of data access/storage on HDDs, the filesystem can effectively optimize the realization of local erasure coding. Therefore, this work focuses on filesystem-level transparent local erasure coding.

2.3 Choosing the Code

An error correction code (ECC) is characterized by four parameters $\{k, m, w, d\}$, i.e., each codeword protects k user data symbols using m redundant symbols (hence the codeword length is $k+m$), each symbol contains w bits, and the minimum codeword distance is d. For any linear ECC code, its minimum distance d is subject to the well-known Singleton bound, i.e., $d \leq m+1$ [16]. A code that achieves the equality in the Singleton bound is called an MDS (maximum distance separable) code, which can achieve the maximum guaranteed error and erasure correction strength. As the most well-known MDS code, RS code [27] has been used in numerous data communication and storage systems. An ECC with the minimum distance of d can correct up to $\lfloor \frac{d-1}{2} \rfloor$ errors or up to $d-1$ erasures (note that the term *erasure* means an error with the known location). Hence, when being used for erasure coding (i.e., the location of all the errors is known prior decoding), a (k, m) RS code guarantees to correctly recover up to m erasures within one codeword.

This work uses RS codes to realize the local erasure coding. RS codes are typically constructed over binary Galois Field (GF). Given the underlying GF with the order of 2^w, the codeword length can be up to $2^w - 1$ and 2^w for cyclic and non-cyclic RS codes, respectively, and each symbol contains w bits. Non-cyclic RS codes (e.g., the widely used Cauchy RS codes [5, 20]) are primarily used for erasure coding, and their encoding/decoding are realized through GF matrix-vector multiplication (and GF matrix inversion for decoding). Cyclic RS codes have a much richer set of encoding/decoding algorithms and can more conveniently handle both errors and erasures. Interested readers are referred to [16, 27] for details.

3 Proposed Design Solution

This section first presents the basic framework on realizing the filesystem-level transparent local erasure coding, then mathematically formulates its effect on reducing the read tail latency, and finally presents solutions to address two non-trivial issues for its practical implementation.

3.1 Basic Framework

When implementing transparent local erasure coding, it is important to treat filesystem metadata and user data differently. Leveraging the large file size and typically coarse-grained data access patterns in data centers, we apply long RS codes to user data on the per-file basis, i.e., each RS codeword spans over hundreds of 4kB sectors and all the data within one RS codeword belong to the same file. This is illustrated in Fig. 3: One (k, m, w) RS codeword spans over $k+m$ consecutive sectors and each w-bit symbol comes from one sector. Hence, HDD read failures on any m sectors could be recovered by RS code decoding. To simplify the implementation, we use the same (k, m, w) RS code for all the files. Let N denote the number of 4kB sectors in one file. The filesystem partitions all the N sectors into $\lceil \frac{N}{k} \rceil$ groups, and appends m sectors to each group for the storage of the RS coding redundancy. The last group may have $k' < k$ sectors, for which we use a shortened (k', m, w) RS code. Note that the shortened (k', m, w) RS code shares the same encoder and decoder as the original (k, m, w) RS code by simply setting the content of the other $k - k'$ sectors as zeros.

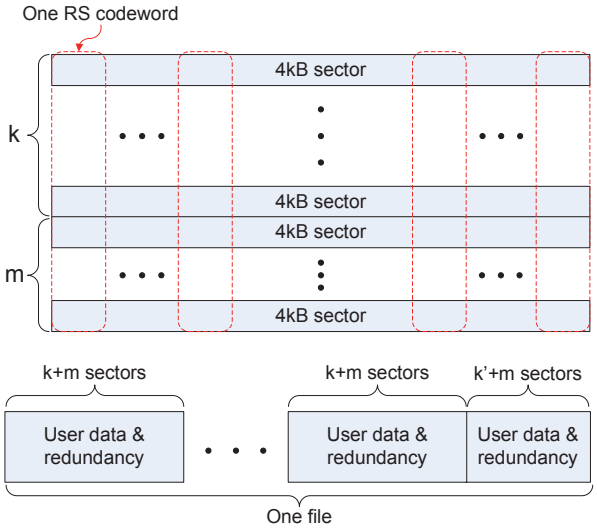


Figure 3: Illustration of per-file local erasure coding.

Let p_h denote the HDD soft sector read failure probability, and let $p_s << p_h$ denote the target local erasure decoding failure probability (i.e., reduce the HDD read retry rate from p_h to p_s via the local erasure coding). Given the value of k, we should search for the minimum value of m subject to

$$\sum_{i=m+1}^{k+m} \binom{k+m}{i} p_h^i \cdot (1-p_h)^{(k+m-i)} \leq p_s. \qquad (1)$$

To illustrate the dependence of coding redundancy ratio (defined as m/k) on the codeword length, we set p_s as 10^{-8} and calculate the required coding redundancy over different k and p_h, as shown in Fig. 4. The results show that, in order to minimize the coding redundancy, we must deploy long RS codes.

When the local erasure coding is implemented on the per-file basis, the average storage capacity overhead can be calculated as follows. Let $g(x)$ denote the probability density function (PDF) of the file size, where x denotes size of one file in terms of the number of 4kB sectors. We

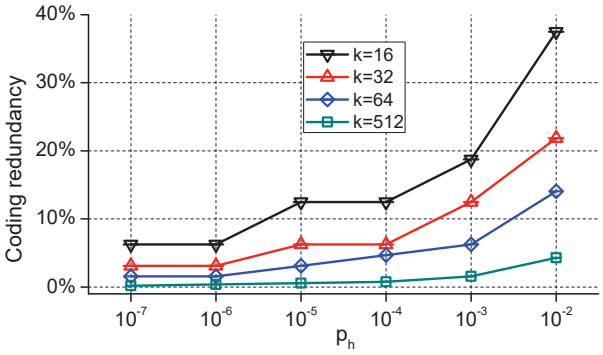


Figure 4: Calculated coding redundancy under different values of k and p_h, where k is the number of user data symbols per codeword and p_h is the soft sector read failure probability. The target decoding failure rate is 10^{-8}.

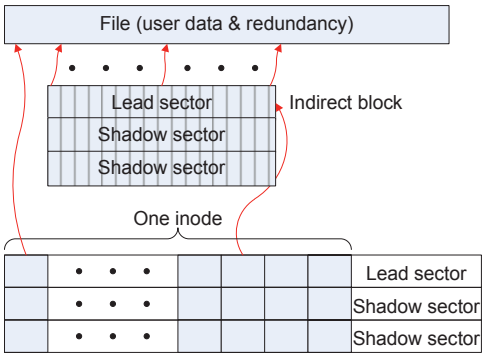


Figure 5: Illustration of replica-based protection for inode.

can express the average storage capacity overhead r_{avg} as

$$r_{avg} = \frac{\int_0^\infty g(x) \cdot \lceil \frac{x}{k} \rceil \cdot m \, dx}{\int_0^\infty g(x) \cdot x \, dx}. \quad (2)$$

This clearly shows that the space overhead reduces as the file size increases, and the minimum overhead per file is m sectors no matter how small the file is. For large files, the overhead can be approximated as m/k (i.e., m redundant sectors per k user data sectors), e.g., the overhead is about 0.39% when using a (1019, 4) RS code.

To accommodate the fine access granularity of filesystem metadata, we apply erasure coding to the filesystem metadata on the per-sector basis, i.e., erasure coding reduces to data replication. In particular, to store one sector of filesystem metadata, we allocate m' consecutive sectors to store m' replicas of the same sector content. Given the soft sector failure probability p_h and the target local erasure decoding failure probability $p_s << p_h$, we should search for the minimum value of m' subject to

$$p_h^{m'} \leq p_s. \quad (3)$$

Since p_h should not be too high (e.g., 10^{-2} and below), a small value of m' (e.g., 3 or 4) is sufficient to make p_s small enough. Therefore, the m' consecutive sectors most likely reside on the same track, and it does not incur noticeable HDD access latency overhead. Among all the m' replica sectors, we designate the first sector as the lead sector and the following $m'-1$ sectors as shadow sectors. All the pointers in the filesystem metadata structure point to the lead sector, and all the shadow sectors are only used for tolerating HDD sector read failures. Using *ext4* as an example, Fig. 5 illustrates its use in the context of inode pointer structure. Each sector storing inode or singly/doubly/triply indirect blocks is replicated m' times, and the pointed data sectors fall into one or multiple local erasure coding groups within the same file.

In order to practically implement this design strategy, the host-to-HDD interface protocol and HDD firmware should be appropriately modified. In particular, the host should be able to notify the HDD whether the host would like the HDD to simply return an immediate error message, upon a read failure, without internal retry. The HDD firmware should support two operational modes: skip a failed sector without retry, or immediately invoke read retry upon a read failure.

Although the above basic framework is indeed very simple and straightforward, its practical implementation involves the following two issues:

- *Unaligned HDD write*: Unaligned HDD write occurs when the user data being written to the HDD do not exactly fill one or multiple coding groups. Ideally we hope to align HDD write with the local erasure coding group boundary, i.e., we can always first accumulate each group of k consecutive user data pages in the host memory, then carry out the local erasure encoding and write the total $k+m$ pages to the HDD. Unfortunately, this may not be always possible in practice, e.g., in the case of direct I/O and synchronous I/O. Even in the case of asynchronous I/O, filesystem must carry out periodic flushing, which could make disk write unaligned with the local erasure coding group boundary.
- *Fine-grained data update*: Although our interested data center applications have predominantly coarse-grained data access, especially data write, it is not impossible to have fine-grained data update (e.g., update of tens of 4kB sectors) in practice. Given the very long RS codeword length, it is not trivial to effectively support such fine-grained data update.

In the remainder of this section, we first derive mathematical formulations for estimating the effectiveness of using local erasure coding to reduce HDD read tail laten-

cy, and then present techniques to address the above two non-trivial implementation issues.

3.2 Tail Latency Estimation

The objective of local erasure coding is to relax the constraint on HDD soft sector read failure probability without sacrificing HDD read tail latency. This subsection derives mathematical formulations for estimating read tail latency with and without using the local erasure coding. We consider the scenario of reading N consecutive sectors from an HDD, and let T denote the latency of successfully reading all the N sectors. Due to the varying additional latency caused by read retry and RS coding, T can be considered as a discrete variable. Therefore, in order to obtain the tail latency (e.g., 99-percentile latency), we should derive its discrete probability distribution, also known as probability mass function (PMF), denoted as $f(T)$. Given the target tail percentage P_{tail} (e.g., 99% or 99.9%), we can search for the tail latency T_{tail} subject to

$$\sum_{T=0}^{T_{tail}} f(T) \geq P_{tail}. \tag{4}$$

Let τ_{retry} denote the latency to recover one sector during the read retry mode, and τ_u denote the latency for an HDD to read one sector during its normal mode. Note that τ_{retry} is approximately the latency of one or more disk rotations. Since we are interested in comparing the latency with and without using the local erasure coding, we omit the HDD seek latency. Recall that p_h denote the soft sector read failure probability. Based on our interaction with HDD vendors, it is reasonable to approximately model the soft sector read failure as an independent and identical distributed (i.i.d.) random variable. Therefore, in this work, we assume there is no correlation in soft sector errors between contiguous sectors. Recall that $f(T)$ denote the probability mass function of the latency T. Hence, we have the formulations for the case without using the local erasure coding:

$$\begin{cases} T = t \cdot \tau_{retry} + \tau_u \cdot N \\ f(T) = \binom{N}{t} p_h^t \cdot (1-p_h)^{(N-t)} \end{cases} \tag{5}$$

where $t \geq 0$ is the number of soft sector read failures.

Next, let us consider the case of using the local erasure coding. Given the coding parameters k and m, we can calculate the local erasure decoding failure probability p_s using Eq. (1). Define $l = \lfloor N/k \rfloor$, and $k' = N - l \cdot k$, i.e., there are total l complete $(k+m)$-sector groups followed by one shortened $(k'+m)$-sector group. Since local erasure decoding can be carried out concurrently with HDD sector read, we assume that only the decoding of the last group contributes additional latency to the overall latency. The formulations for T and $f(T)$ are derived for the following four different cases:

1. Case I: None of the first l groups suffers from local erasure decoding failure, and none of the last k' user data sectors suffers from soft sector read failure:

$$\begin{cases} T = \tau_u \cdot (N + l \cdot m) \\ f(T) = (1-p_s)^l \cdot (1-p_h)^{k'} \end{cases} \tag{6}$$

2. Case II: None of the first l groups experiences local erasure decoding failure, but $e_0 \leq m$ sector read failures occur in the last $(k'+m)$ sectors. Let $\tau_{dec}(e_0)$ denote the latency to correct e_0 sectors, we have

$$\begin{cases} T = \tau_u \cdot \big(N + (l+1) \cdot m\big) + \tau_{dec}(e_0) \\ f(T) = (1-p_s)^l \cdot \binom{k'+m}{e_0} \cdot p_h^{e_0} \\ \qquad \cdot (1-p_h)^{(k'+m-e_0)} \end{cases} \tag{7}$$

3. Case III: Among the first l groups, j groups experience local erasure decoding failures due to $e_1,\ e_2, \cdots, e_j > m$ sector read failures, and $e_0 \leq m$ soft sector read failures occur in the last $(k'+m)$-sector group. Let p_i denote the probability that e_i soft sector read failures occur within one group, where $p_i = \binom{k+m}{e_i} \cdot p_h^{e_i} \cdot (1-p_h)^{(k+m-e_i)}$, we have

$$\begin{cases} T = \tau_u \cdot \big(N + (l+1) \cdot m\big) + j \cdot \tau_{dec}(m) \\ \qquad + \tau_{dec}(e_0) + \sum_{i=1}^{j} \tau_{retry} \cdot (e_i - m) \\ f(T) = \Big(\dfrac{l!}{(l-j)!} \cdot \prod_{i=1}^{j} p_i\Big) \cdot (1-p_s)^{(l-j)} \\ \qquad \cdot \binom{k'+m}{e_0} p_h^{e_0} \cdot (1-p_h)^{(k'+m-e_0)} \end{cases} \tag{8}$$

4. Case IV: It only differs from the Case III in that the last $(k'+m)$-sector group suffers from local erasure decoding failure as well (i.e., $e_0 > m$), and we have

$$\begin{cases} T = \tau_u \cdot \big(N + (l+1) \cdot m\big) + (j+1) \cdot \\ \qquad \tau_{dec}(m) + \sum_{i=0}^{j} \tau_{retry} \cdot (e_i - m) \\ f(T) = \Big(\dfrac{l!}{(l-j)!} \cdot \prod_{i=0}^{j} p_i\Big) \cdot (1-p_s)^{(l-j)} \end{cases} \tag{9}$$

When using the Eq. (4) to estimate the tail latency, we should exhaustively exam all the four cases above to obtain a complete PMF $f(T)$ of the access latency.

3.3 Addressing Unaligned HDD Write

We can formulate an unaligned HDD write as follows: A vector $\mathbf{d} = [\mathbf{d}_1, \mathbf{d}_2]$ contains k user data symbols in one

codeword, where sub-vectors $\mathbf{d}_1$ and $\mathbf{d}_2$ contain k_1 and k_2 user data symbols ($k_1 + k_2 = k$). Unaligned HDD write occurs once the filesystem must write data to HDD when only $\mathbf{d}_1$ is available. The straightforward solution is to store dynamically shortened RS codewords on HDDs, as illustrated in Fig. 6(a). This however suffers from two problems: (1) Filesystem must accordingly record the length of each shortened codeword in the file metadata, which could noticeably complicate filesystem design. (2) The average coding redundancy could increase and hence degrade the effective HDD bit cost.

Therefore, we should still use the same RS code with the fixed k in the presence of unaligned HDD write. To achieve this objective, we propose a cache-assisted progressive encoding strategy. Recall that ECC encoding can be modeled as a matrix-vector multiplication $\mathbf{r} = \mathbf{G} \cdot \mathbf{d}$, where $\mathbf{r}$ represents the m redundant symbols to be computed, $\mathbf{d}$ represents the k user data symbols, and the $m \times k$ matrix $\mathbf{G}$ is the generator matrix. We define two length-k vectors $\mathbf{d}^{(1)} = [\mathbf{d}_1, \mathbf{O}]$ and $\mathbf{d}^{(2)} = [\mathbf{O}, \mathbf{d}_2]$, where $\mathbf{O}$ represents an all-zero vector. Because the RS codes are constructed over binary GF, we have $\mathbf{d} = \mathbf{d}^{(1)} \oplus \mathbf{d}^{(2)}$, where $\oplus$ represents bit-wise XOR operation. Therefore, the encoding procedure can be written as

$$\mathbf{r} = \mathbf{G} \cdot \mathbf{d} = \mathbf{G} \cdot \mathbf{d^{(1)}} \oplus \mathbf{G} \cdot \mathbf{d^{(2)}}. \tag{10}$$

Define $\mathbf{r}^{(1)} = \mathbf{G} \cdot \mathbf{d^{(1)}}$ and $\mathbf{r}^{(2)} = \mathbf{G} \cdot \mathbf{d^{(2)}}$, we have that $\mathbf{r} = \mathbf{r}^{(1)} \oplus \mathbf{r}^{(2)}$. As illustrated in Fig. 6(b), the filesystem first writes the k_1 sectors and associated m sectors storing $\mathbf{r}^{(1)}$ to the HDD, and meanwhile keeps $\mathbf{r}^{(1)}$ in OS page cache. Once the subsequent k_2 sectors are ready to be written to the HDD, the filesystem carries out encoding to obtain $\mathbf{r}^{(2)}$ and then compute the overall redundancy $\mathbf{r} = \mathbf{r}^{(1)} \oplus \mathbf{r}^{(2)}$. Finally, filesystem appends k_2 sectors storing $\mathbf{d}_2$ and m sectors storing $\mathbf{r}$ after the previous k_1 sectors (i.e., overwrites the previously written m sectors storing $\mathbf{r}^{(1)}$). In this way, we can ensure that the filesystem always uses the same fixed-length RS codewords on each file (except the last portion of the file). Since m is typically very small (e.g., 5 or 10) and we only need to keep one intermediate coding redundancy (i.e., $\mathbf{r}^{(1)}$) for each file, this design strategy will not cause noticeable cache space overhead. Clearly, this design strategy can be applied recursively when one group of k sectors is written to the HDD in more than two batches.

Finally, we note that, if power failure occurs when we overwrite $\mathbf{r}^{(1)}$, the previously written data $\mathbf{d}_1$ are not protected by valid local erasure code and hence are more subject to read retry. This however will not cause any storage integrity degradation since we only use local erasure coding to mitigate soft sector read failure.

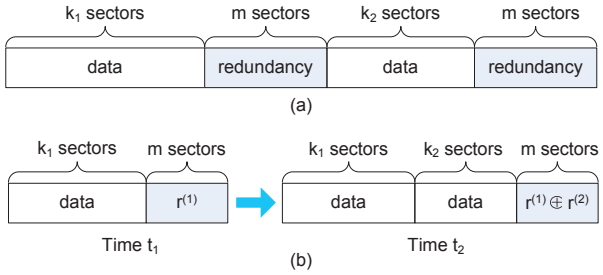


Figure 6: Addressing unaligned HDD write using (a) straightforward dynamic codeword shortening, and (b) proposed cached-assisted progressive encoding, where $k_1 + k_2 = k$.

3.4 Addressing Fine-grained Data Update

In the case of fine-grained data update, filesystem must carry out read-modify-write operations to update the m-sector coding redundancy. This may lead to noticeable system performance degradation, especially if the fine-grained data update is triggered by synchronous writes and meanwhile the other data in the same coding group are not present in host memory. We propose a two-phase write procedure to address this issue. As pointed out above, since local erasure coding only mitigates soft sector read failures, it does not degrade data storage integrity even if some data are temporarily not protected by local erasure code. In another word, when the m-sector coding redundancy becomes invalid due to fine-grained data update, the associated k sectors are simply subject to a higher risk of HDD read retry without any loss on their storage integrity. This observation directly suggests the use of a two-phase write procedure: Upon a fine-grained data update caused by synchronous writes, the filesystem first serves the synchronous write request as in current practice without modifying the m-sector coding redundancy. Then filesystem updates the corresponding m-sector coding redundancy asynchronously during background operations. Note that, if the fine-grained data update is not triggered by synchronous writes, the filesystem can merge these two phases together.

The next question is how to update the m-sector coding redundancy. To simplify the discussion, let us consider the scenario where we update the k user data symbols in one RS codeword on HDD from $\mathbf{d} = [\mathbf{d}_1, \mathbf{d}_2]$ to $\mathbf{d}' = [\mathbf{d}'_1, \mathbf{d}_2]$, where the unchanged content $\mathbf{d}_2$ is not cached in memory. To accordingly update the coding redundancy from $\mathbf{r} = \mathbf{G} \cdot \mathbf{d}$ to $\mathbf{r}' = \mathbf{G} \cdot \mathbf{d}'$, we have the following two different strategies:

1. We simply update the redundancy in the most straightforward manner, i.e., read the unchanged content $\mathbf{d}_2$ from HDD, then obtain the new redundancy $\mathbf{r}'$ by computing the complete matrix-vector

multiplication $\mathbf{G}\cdot\mathbf{d}'$, and finally write $\mathbf{r}'$ to HDD. Let k_1 denote the length of $\mathbf{d}_1$ (i.e., the data being updated). In order to update the m-sector coding redundancy, we need to read $k-k_1$ sectors from HDD.

2. The second option updates the redundancy through indirect computation. As discussed above in Section 3.3, we define $\mathbf{d}^{(1)}=[\mathbf{d}_1,\mathbf{O}]$, $\mathbf{d}^{(1)'}=[\mathbf{d}'_1,\mathbf{O}]$, and $\mathbf{d}^{(2)}=[\mathbf{O},\mathbf{d}_2]$. We define $\mathbf{r}^{(1)}=\mathbf{G}\cdot\mathbf{d}^{(1)}$, $\mathbf{r}^{(1)'}=\mathbf{G}\cdot\mathbf{d}^{(1)'}$, and $\mathbf{r}^{(2)}=\mathbf{G}\cdot\mathbf{d}^{(2)}$. The original redundancy can be expressed as $\mathbf{r}=\mathbf{r}^{(1)}\oplus\mathbf{r}^{(2)}$, which can also be re-written as $\mathbf{r}\oplus\mathbf{r}^{(1)}=\mathbf{r}^{(2)}$. As a result, we can express the updated redundancy $\mathbf{r}'$ as

$$\mathbf{r}'=\mathbf{r}^{(1)'}\oplus\mathbf{r}^{(2)}=\mathbf{r}^{(1)'}\oplus\mathbf{r}\oplus\mathbf{r}^{(1)}. \qquad (11)$$

Therefore, we read the original content $\mathbf{d}_1$ and original redundancy $\mathbf{r}$ from HDD, and accordingly generate the updated redundancy $\mathbf{r}'$ based upon Eq. (11). Using this strategy, in order to update the m-sector coding redundancy, we need to read k_1+m sectors from HDD.

These two strategies can be directly generalized for more complicated fine-grained data update (i.e., multiple separate regions within the same k-sector group are updated).

4 Analysis and Experiments

We evaluate the proposed design solution mainly from two aspects: (1) Improvement on tail latency: This proposed design solution aims to mitigate the impact of HDD technology scaling on tail latency. Using the mathematical formulations presented in Section 3.2, we show analysis results of HDD tail latency with and without using the proposed design solution under different configurations. (2) Impact on average system speed performance: The on-the-fly local erasure encoding and decoding can add noticeable extra latency into the data I/O path, which could degrade the average system speed performance. We use a variety of benchmarks in big data benchmark suite HiBench 3.0 [1] to evaluate the impact on average system speed performance. Moreover, we will present results on the storage capacity overhead under HiBench 3.0 benchmarks, and the latency overhead induced by fine-grained update.

4.1 Implementation of Coding Engine

We first discuss the construction of the RS codes being used in this study and their encoder/decoder implementation. We set the target local erasure code decoding failure probability p_s as 10^{-6}. Recall that p_h denotes the soft sector read failure probability. We consider four different values of p_h, including 1×10^{-4}, 5×10^{-4}, 1×10^{-3}, and 5×10^{-3}, and set the target codeword length as 255 and 1023. Accordingly, we can calculate the code parameters k and m, which are listed in Table 1. Note that the number of bits per symbol (i.e., w) is 8 and 16 when the codeword length is 255 and 1023, respectively.

Table 1: Parameters of RS-based local erasure codes.

p_h	$m+k=255$		$m+k=1023$	
	m	k	m	k
1×10^{-4}	3	252	4	1019
5×10^{-4}	4	251	7	1016
1×10^{-3}	5	250	9	1014
5×10^{-3}	9	246	19	1004

Based upon the open-source library jerasure [2], we developed and integrated an RS coding library into *ext4* in Linux kernel 3.10.102. Both encoding and decoding are realized through direct matrix-based computation instead of polynomial-based computation, which can readily leverage the on-chip cache resource in CPUs to maximize the throughput. We measured its encoding and decoding throughput on a PC with a 3.30GHz CPU and 8GB DRAM, and the results are shown in Fig. 7.

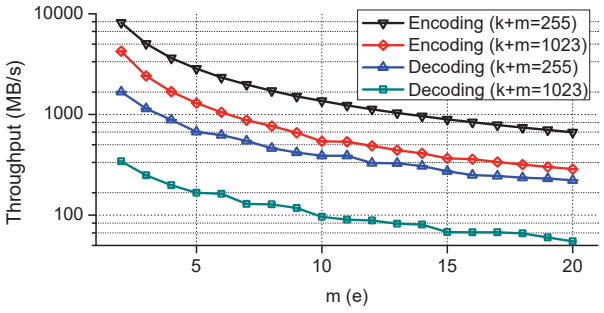


Figure 7: Measured encoding and decoding throughput under different value of m or the number of erasures e.

Let e denote the number of erased symbols per codeword. Given the same codeword length, the encoding (decoding) throughput reduces as m (e) increases. This is because the size of the matrices involved in encoding (decoding) is proportional the value of m (e). The results show that decoding throughput is significantly lower than encoding throughput under the same m and e. For example, with the codeword length of 255, the encoding throughput is about 1.5GB/s at $m=10$ while the decoding throughput is only 400MB/s at $e=10$. Fortunately, the probability that one codeword contains e erased symbols exponentially reduces as e increases. For example, with the codeword length of 255 and p_h of 1×10^{-3}, the probability that one codeword contains 2, 4, and 6 erased symbols is 2.5×10^{-2}, 1.3×10^{-4}, and 2.8×10^{-7}.

In addition, motivated by the emerging trend of integrating CPU with FPGA in one chip package (e.g., the

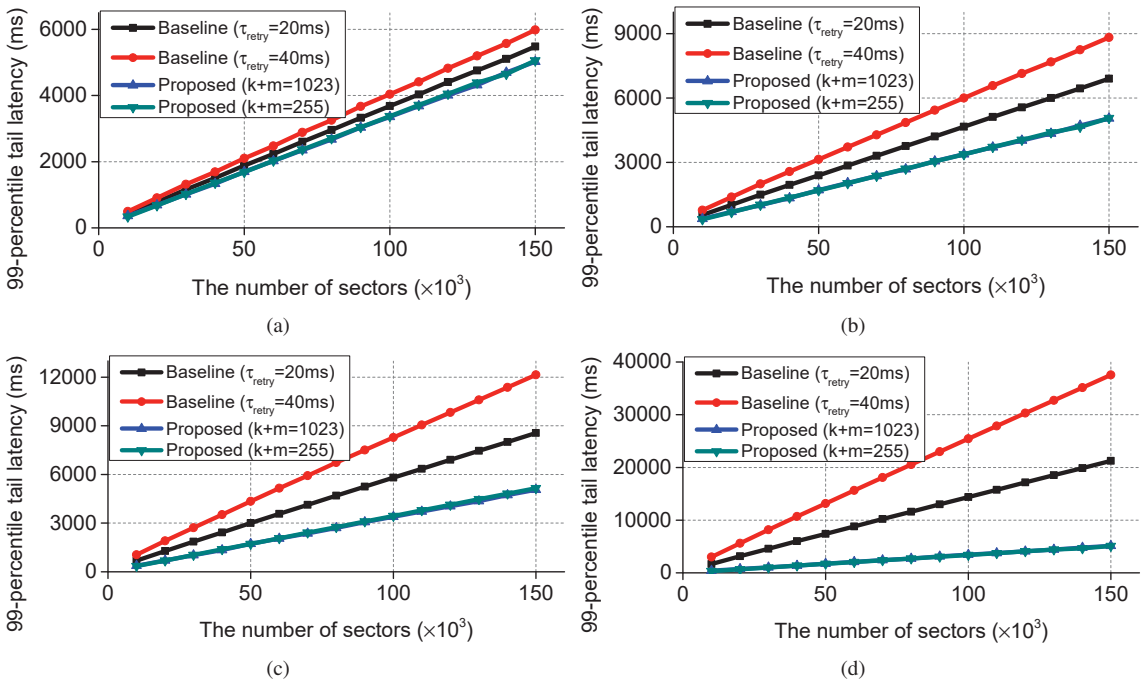


Figure 8: Calculated 99-percentile latency when reading different number of consecutive sectors with (a) $p_h = 10^{-4}$, (b) $p_h = 5 \times 10^{-4}$, (c) $p_h = 10^{-3}$, and (d) $p_h = 5 \times 10^{-3}$.

Xeon processors with built-in FPGAs announced by Intel at Open Compute Project (OCP) Summit 2016), we also studied the hardware-based implementation of RS coding engine. With abundant hardware-level parallelism and very high-speed CPU-FPGA interconnect, offloading RS encoding and decoding into the built-in hardware accelerator can significantly improve the achievable operational throughput. In this work, we designed parallel polynomial-based RS encoder, and RS decoder using the well-known Berlekamp-Massey algorithm and the parallel architecture presented in [14]. The design was carried out at the RTL level and Table 2 lists the synthesis results (in terms of equivalent XOR gate count) for all the eight RS codes, where all the implementations have the same throughput of 4GB/s with the clock frequency of 250MHz. The results show that, for the same RS code, the decoder consumes about 10x more silicon resource than the encoder at the same 4GB/s throughput. The decoder gate counts range from 156k to 894k, which can readily fit into modern FPGA devices.

Table 2: Hardware-based RS encoder/decoder implementation synthesis results.

Code Parameters		Equivalent XOR Gate Count	
m	k	Encoder	Decoder
3	252	11k	156k
4	251	11k	161k
5	250	17k	185k
9	246	28k	232k
4	1019	16k	634k
7	1016	31k	699k
9	1014	39k	732k
19	1004	78k	894k

4.2 HDD Tail Latency

Applying the mathematical formulations presented in Section 3.2, we computed the 99-percentile tail latency when reading consecutive N sectors from HDD, where N ranges from 10k to 150k (i.e., the data volume ranges from 40MB to 600MB). Recall that τ_{retry} represents the latency to recover one sector during the read retry mode, and τ_u represents the latency for HDD to read one sector during its normal mode. Assuming the use of 7200rpm HDD, we set τ_u as 33μs. Since read retry latency could significantly vary in practice, we treat τ_{retry} as the average read retry latency. Based upon our communications with HDD vendors, we consider two different values of τ_{retry}: 20ms and 40ms. The results are shown in Fig. 8.

The results show the effectiveness of using the local erasure coding to reduce the HDD read tail latency in the presence of high soft sector read failure probabilities. Since we constructed the RS codes with the target decoding failure probability of 10^{-6}, different value of per-

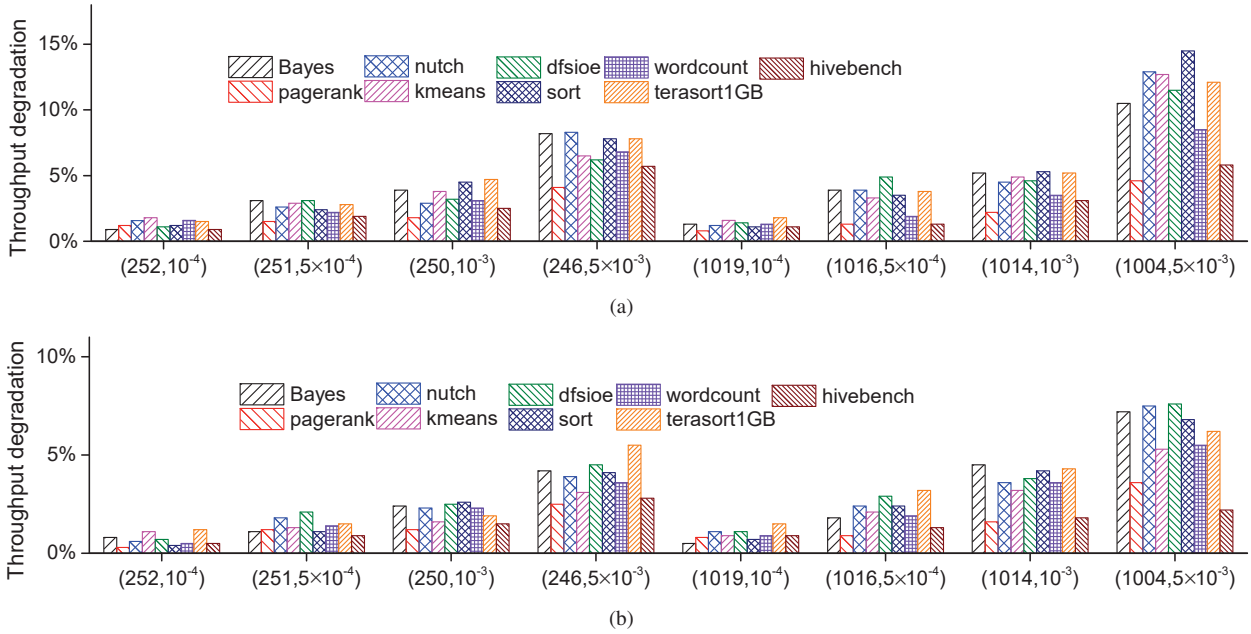


Figure 9: Measured throughput degradation of all the benchmarks when using (a) software-based and (b) hardware-based RS coding engine implementation. Each code is represented as (k, p_h).

sector read retry latency τ_{retry} (i.e., 20ms or 40ms in this study) or different codeword length (i.e., 255 and 1023 in this study) does not incur noticeable difference in terms of tail latency, as shown in Fig. 8. The gain of applying local erasure coding improves as the soft sector read failure probability increases. In addition, the gain is relatively weakly dependent on the number of sectors being read. When we read 10k consecutive 4kB sectors, the use of local erasure coding can reduce the 99-percentile latency by 50.1% (τ_{retry}=20ms) and 67.2% (τ_{retry}=40ms) under the soft sector read failure probability p_h is 1×10^{-3}. The gain improves to 78.6% (τ_{retry}=20ms) and 88.1% (τ_{retry}=40ms) respectively as we increase p_h to 5×10^{-3}. When we read 100k consecutive 4kB sectors with p_h of 5×10^{-3}, the 99-percentile latency is reduced by 76.2% (τ_{retry}=20ms) and 86.6% (τ_{retry}=40ms). It should be pointed out that the per-sector read retry latency τ_{retry} strongly depends on how aggressively the HDD industry is willing to exploit the use of local erasure coding to push the magnetic recording technology scaling. Hence the results above mainly serve as the preliminarily estimation on the potential of applying local erasure coding to reduce HDD read tail latency.

4.3 Impact on Average Speed Performance

This work measures the impact on average system speed performance by running the following workloads in the benchmark suite HiBench 3.0: (1) Job based micro benchmarks Sort (*sort*) and WordCount (*wordcount*), which sorts and counts input text data generated by RandomTextWriter; (2) SQL benchmark *hivebench* that performs scan, join and aggregate operations, based upon the workload characteristics presented in [18]; (3) Web search benchmarks PageRank (*pagerank* and Nutchindexing (*nutch*); (4) Machine learning benchmarks Bayesian Classification (*bayes*) and K-means clustering (*kmeans*); (5) HDFS benchmark enhanced DFSIO (*dfsioe*); and (6) *terasort* a standard benchmark here sorts 1GB generated by teragen. All the experiments are carried out on one PC with a 3.30GHz CPU, 8GB DRAM, and 500GB 7200rpm HDD.

We integrated the software-based RS coding library into *ext4* in Linux kernel 3.10.102, and accordingly modified the I/O stack to incorporate the use of local erasure coding. Fig. 9 shows the measured throughput degradation of all the benchmarks when using software-based or hardware-based RS coding engine. Due to the absence of commercial CPUs with built-in FPGA, we estimated the results in Fig. 9(b) by adding delay into the I/O stack to mimic the effect of hardware-based RS coding, where we set the encoding/decoding throughput as 4GB/s. When running each benchmark, we randomly set one sector being read from HDD as a failure with the probability p_h, and accordingly carry out RS code decoding and issue additional HDD read if required by the decoding.

The average speed degradation is mainly due to the following three factors: (1) RS code encoding latency,

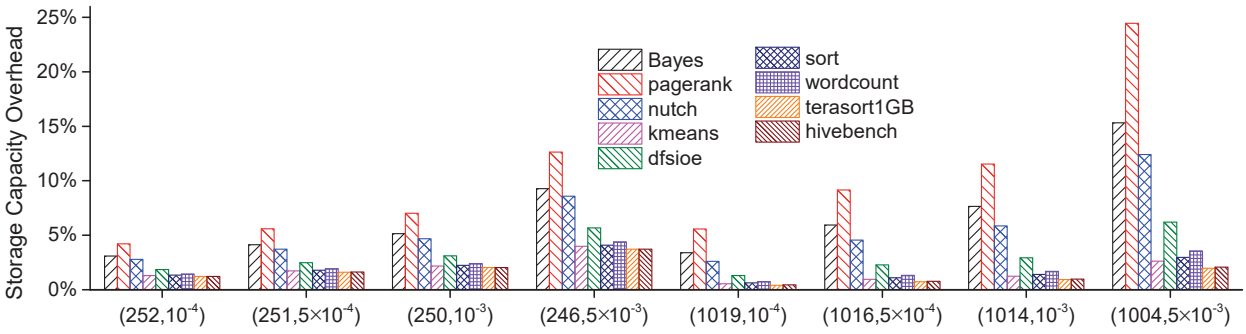


Figure 10: HDD storage capacity overhead under all the benchmarks when using the eight different RS codes. Each code is represented as (k, p_h).

(2) HDD latency for reading extra data required by decoding, and (3) RS code decoding latency. We note that, whenever RS code decoding is triggered for one group of $k+m$ sectors, we always first check whether the page cache contains some sectors in this group and only fetch un-cached sectors from HDD. Hence, the average speed performance degradation further depends on the data access patterns of each benchmark. Under the same value of p_h, a shorter codeword length (e.g., 255) incurs less degradation than a longer codeword length (e.g., 1023). This is because a longer codeword length results in a longer encoding latency and a higher probability of reading more HDD sectors during decoding. Compared with the software-based implementation, hardware-based RS coding implementation can reduce the average speed degradation by 60.6% on average. Even in the case of software-based RS coding implementation, the average speed performance degradation is not significant and can be (much) less than 10% when p_h is 1×10^{-3}.

4.4 Storage Capacity Overhead

Storage capacity overhead induced by the proposed design solution depends on both the coding parameters (i.e., k and m) and the file size distribution. We collected the file size distributions when running those HiBench 3.0 benchmarks, and accordingly estimated the storage capacity overhead as shown in Fig. 10, where each code is denoted using the parameters (k, m, p_h).

The results show that the storage capacity overhead increases as the sector read failure probability p_h becomes worse. This can be intuitively justified since, given the same target decoding failure probability, a worse p_h demands a stronger code with a larger coding redundancy. With the same code, different benchmarks have different storage capacity overhead as shown in Fig. 10, which is due to their different file size characteristics. The benchmarks (e.g., *pagerank*, *bayes*, and *nutch*), which have a large number of small files, incur relatively large storage capacity overhead. For example, in the case of *pagerank*, over 95% of its total storage usage is caused by small files with the size between 100kB and 1MB, and another 1.4% is caused by files smaller than 10kB. Because of the long codeword length (i.e., 255 and 1023 in this study), each small file is entirely protected by one shortened codeword with coding redundancy much higher than m/k. With p_h of 1×10^{-3} and codeword length of 1023, the storage capacity overhead is as high as over 12% for *pagerank*. In contrast, the other benchmarks (e.g., *kmeans* and *hivebench*), which are dominated by large-size files, have much less storage capacity overhead. For example, in the case of *hivebench*, over 99% of its total storage usage is caused by files larger then 100MB. As a result, with p_h of 1×10^{-3} and codeword length of 1023, the overall storage capacity overhead is only less than 2% for *hivebench*.

4.5 Impact of Fine-grained Data Update

Although data centers tend to avoid data update on HDD through the use of immutable data structure, it is still of practical interest to study the latency overhead incurred by data update. We note that only fine-grained data update (i.e., only a portion of data within one $(k+m)$-sector coding group is updated) is subject to latency penalty. As discussed in Section 3.4, we can use two different methods to carry out fine-grained data update. In order to update the data on HDD from $\mathbf{d} = [\mathbf{d}_1, \mathbf{d}_2]$ to $\mathbf{d}' = [\mathbf{d}'_1, \mathbf{d}_2]$, the first method reads $\mathbf{d}_2$ from HDD to directly re-compute the updated coding redundancy, while the second method reads $\mathbf{d}_1$ and old-version redundancy from HDD to in-directly compute the updated coding redundancy.

We carried the following experiments: With the codeword length of 255, we first encode and write 10GB data to a 7200rpm HDD. After clearing the OS page cache, we update l_u consecutive sectors at a random location within each codeword using either the first or second method.

We compare the total latency against the baseline without using local erasure coding. We repeat the experiments by setting l_u as 50 and 200, and Fig. 11 shows the measurement results when using different RS codes. The results show that the second method (i.e., in-direct coding redundancy re-computation) appears to be the better choice. In particular, for very fine-grained data update (i.e., update 50 sectors within 255 sectors), the second method significantly outperforms the first method.

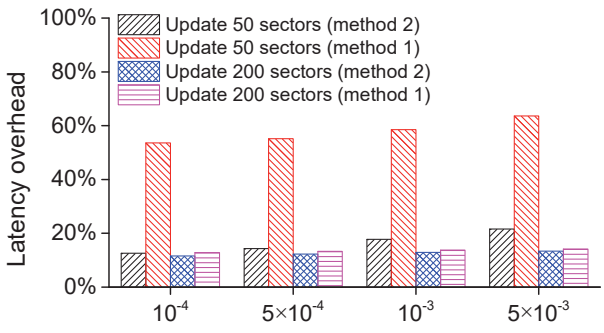


Figure 11: Measured fine-grained update latency overhead.

5 Related Work

Aiming to enhance storage systems by introducing coding redundancy across multiple sectors, this work shares the same nature as the widely used RAID and distributed erasure coding. Both RAID-5/6 and distributed erasure coding primarily target at tolerating catastrophic HDD failures, and typically employ erasure codes with relatively small codeword length (e.g., 12) and hence large coding redundancy (e.g., 20% to 50%). Recent work [3, 4, 19] presented specific erasure code construction techniques for RAID applications that can tolerate individual sector failures in addition to catastrophic HDD failures. However, RAID is being replaced by distributed erasure coding in data centers. Extensive research has been carried out on the construction of distributed erasure codes (e.g., see [9–11, 22]) and investigating/optimizing the system-level performance (e.g.,see [7, 12, 25, 29]). Nevertheless, the distributed nature makes it unsuitable for mitigating soft sector read failures of individual HDDs. Prabhakaran et al. [21] presented an Internal RObustNess (IRON) filesystem design framework that includes a variety of HDD failure detection and recovery techniques. Its transaction checksum technique inspired the implementation of journal checksum in *ext4* filesystem. It presented the metadata replication scheme, which is similar to the one being used in this work.

Researchers in the magnetic recording industry also recently investigated how intra-HDD erasure coding can complement with existing per-sector ECC (e.g., low-density parity-check (LDPC) codes) to improve the read channel performance. For example, the authors of [30] studied the effect of adding one parity check code across a number of 4kB sector inside one HDD. Nevertheless, intra-HDD realization of erasure coding is subject to several problems as discussed in Section 2.2.

6 Conclusions

This paper carries out an exploratory study on applying local erasure coding to facilitate technology scaling for data center HDDs. With finer track pitch, future HDDs are increasingly subject to areal density vs. read retry rate conflict. This is particularly serious for data centers that are very sensitive to bit cost and meanwhile cannot tolerate long HDD read tail latency. Aiming to alleviate such a dilemma, this paper investigates the potential and feasibility of using filesystem-level transparent local erasure coding to mitigate soft sector read failures. This paper presents the basic design framework and develops techniques to address two issues including unaligned HDD write and fine-grained data update. This paper further derives mathematical formulations for estimating the effectiveness on reducing tail latency, which has been quantitatively demonstrated through numerical analysis. To evaluate its impact on average system speed performance and demonstrate its practical implementation feasibility, we integrated this design solution into Linux kernel and carried out experiments using a variety of big data benchmarks. Its storage capacity overhead is also evaluated over various big data benchmarks. The analysis and experimental results demonstrate its promising potential and practical feasibility to address the bit cost vs. tail latency dilemma for future data center HDDs.

Acknowledgments

We would like to thank our shepherd Peter Desnoyers 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 IDEMA/ASTC and the National Science Foundation under Grant No. CCF-1629218.

References

[1] *HiBench 3.0*. https://github.com/intel-hadoop/HiBench/releases.

[2] *Jerasure*. https://github.com/tsuraan/Jerasure.

[3] M. Blaum, J. L. Hafner, and S. Hetzler. Partial-MDS codes and their application to RAID type of architectures. *IEEE Transactions on Information Theory*, 59(7):4510–4519, July 2013.

[4] M. Blaum, J. S. Plank, M. Schwartz, and E. Yaakobi. Construction of partial MDS and sector-disk codes with two global parity symbols. *IEEE Transactions on Information Theory*, 62(5):2673–2681, May 2016.

[5] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical report, Technical Report TR-95-048, International Computer Science Institute, August 1995.

[6] E. Brewer, L. Ying, L. Greenfield, R. Cypher, and T. T'so. Disks for data centers. Technical report, Google, 2016.

[7] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proc. of the ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[8] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[9] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010.

[10] K. M. Greenan, X. Li, and J. J. Wylie. Flat xor-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2010.

[11] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *Proc. of USENIX Annual Technical Conference (ATC)*, pages 15–26, 2012.

[12] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, pages 251–264, 2012.

[13] A. R. Krishnan, R. Radhakrishnan, B. Vasic, A. Kavcic, W. Ryan, and F. Erden. 2-d magnetic recording: Read channel modeling and detection. *IEEE Transactions on Magnetics*, 45(10):3830–3836, Oct 2009.

[14] H. Lee. High-speed vlsi architecture for parallel reed-solomon decoder. *IEEE transactions on very large scale integration (VLSI) systems*, 11(2):288–294, 2003.

[15] F. Lim, B. Wilson, and R. Wood. Analysis of shingle-write readback using magnetic-force microscopy. *IEEE Transactions on Magnetics*, 46(6):1548–1551, Jun. 2010.

[16] S. Lin and D. J. Costello. *Error Control Coding: Fundamentals and Applications (2nd Ed.)*. Prentice Hall, 2004.

[17] K. Miura, E. Yamamoto, H. Aoi, and H. Muraoka. Estimation of maximum track density in shingles writing. *IEEE Transactions on Magnetics*, 45(10):3722–3725, Oct. 2009.

[18] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 165–178, 2009.

[19] J. S. Plank and M. Blaum. Sector-disk (sd) erasure codes for mixed failure modes in RAID systems. *ACM Trans. Storage*, 10(1):4:1–4:17, Jan. 2014.

[20] J. S. Plank and L. Xu. Optimizing Cauchy Reed-solomon Codes for fault-tolerant network storage applications. In *IEEE International Symposium on Network Computing Applications*, July 2006.

[21] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 206–220, 2005.

[22] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proc. of the VLDB Endowment*, volume 6, pages 325–336, 2013.

[23] M. A. Seigler, W. A. Challener, E. Gage, N. Gokemeijer, G. Ju, B. Lu, K. Pelhos, C. Peng, R. E. Rottmayer, X. Yang, H. Zhou, and T. Rausch. Integrated head assisted magnetic recording head: design and recording demonstration. *IEEE Transactions on Magnetics*, 44(1), Jan. 2008.

[24] Y. Shiroishi, K. Fukuda, I. Tagawa, H. Iwasaki, S. Takenoiri, H. Tanaka, H. Mutoh, and N. Yoshikawa. Future options for hdd storage. *IEEE Transactions on Magnetics*, 45(10):3816–3822, Oct 2009.

[25] M. Su, L. Zhang, Y. Wu, K. Chen, and K. Li. Systematic data placement optimization in multi-cloud storage for complex requirements. *IEEE Transactions on Computers*, 65(6):1964–1977, 2016.

[26] D. Weller, G. Parker, O. Mosendz, E. Champion, B. Stipe, X. Wang, T. Klemmer, G. Ju, and A. Aian. A HAMR media technology roadmap to an areal density of 4 Tb/in^2. *IEEE Transactions on Magnetics*, 50(1), Jan. 2014.

[27] S. B. Wicker and V. K. Bhargava. *Reed-Solomon Codes and Their Applications*. IEEE Press, 1994.

[28] R. Wood, R. Galbraith, and J. Coker. 2-d magnetic recording: Progress and evolution. *IEEE Transactions on Magnetics*, 51(4):1–7, April 2015.

[29] L. Xiang, Y. Xu, J. Lui, and Q. Chang. Optimal recovery of single disk failure in rdp code storage systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 119–130, 2010.

[30] S. Yang, Y. Han, X. Wu, R. Wood, and R. Galbraith. A soft decodable concatenated LDPC code. *IEEE Transactions on Magnetics*, 51(11):1–4, Nov 2015.

Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions

Aishwarya Ganesan, Ramnatthan Alagappan,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
University of Wisconsin – Madison

Abstract

We analyze how modern distributed storage systems behave in the presence of file-system faults such as data corruption and read and write errors. We characterize eight popular distributed storage systems and uncover numerous bugs related to file-system fault tolerance. We find that modern distributed systems do not consistently use redundancy to recover from file-system faults: a single file-system fault can cause catastrophic outcomes such as data loss, corruption, and unavailability. Our results have implications for the design of next generation fault-tolerant distributed and cloud storage systems.

1 Introduction

Cloud-based applications such as Internet search, photo and video services [19, 65, 67], social networking [90, 93], transportation services [91, 92], and e-commerce [52] depend on *modern distributed storage systems* to manage their data. This important class of systems includes key-value stores (e.g., Redis), configuration stores (e.g., ZooKeeper), document stores (e.g., MongoDB), column stores (e.g., Cassandra), messaging queues (e.g., Kafka), and databases (e.g., RethinkDB).

Modern distributed storage systems store data in a replicated fashion for improved reliability. Each replica works atop a commodity local file system on commodity hardware, to store and manage critical user data. In most cases, replication can mask failures such as system crashes, power failures, and disk or network failures [22, 24, 30, 31, 40, 80]. Unfortunately, storage devices such as disks and flash drives exhibit a more complex failure model in which certain blocks of data can become inaccessible (read and write errors) [7, 9, 48, 54, 79, 81] or worse, data can be silently corrupted [8, 60, 85]. These complex failures are known as partial storage faults [63].

Previous studies [10, 63, 98] have shown how partial storage faults are handled by file systems such as ext3, NTFS, and ZFS. File systems, in some cases, simply propagate the faults as-is to applications; for example, ext4 returns corrupted data as-is to applications if the underlying device block is corrupted. In other cases, file systems react to the fault and transform it into a different one before passing onto applications; for example, btrfs transforms an underlying block corruption into a read error. In either case, we refer to the faults thrown by the file system to its applications as *file-system faults*.

The behavior of modern distributed storage systems in response to file-system faults is critical and strongly affects cloud-based services. Despite this importance, little is known about how modern distributed storage systems react to file-system faults.

A common and widespread expectation is that redundancy in higher layers (i.e., across replicas) enables recovery from local file-system faults [12, 22, 35, 41, 81]. For example, an inaccessible block of data in one node of a distributed storage system would ideally *not* result in a user-visible data loss because the same data is redundantly stored on many nodes. Given this expectation, in this paper, we answer the following questions: *How do modern distributed storage systems behave in the presence of local file-system faults? Do they use redundancy to recover from a single file-system fault?*

To study how modern distributed storage systems react to local file-system faults, we build a fault injection framework called CORDS which includes the following key pieces: *errfs*, a user-level FUSE file system that systematically injects file-system faults, and *errbench*, a suite of system-specific workloads which drives systems to interact with their local storage. For each injected fault, CORDS automatically observes resultant system behavior. We studied eight widely used systems using CORDS: Redis [66], ZooKeeper [6], Cassandra [4], Kafka [5], RethinkDB [70], MongoDB [51], LogCabin [45], and CockroachDB [14].

The most important overarching lesson from our study is this: a single file-system fault can induce catastrophic outcomes in most modern distributed storage systems. Despite the presence of checksums, redundancy, and other resiliency methods prevalent in distributed storage, a single untimely file-system fault can lead to data loss, corruption, unavailability, and, in some cases, the spread of corruption to other intact replicas.

The benefits of our systematic study are twofold. First, our study has helped us characterize file-system fault handling behaviors of eight systems and also uncover numerous bugs in these widely used systems. We find that these systems can silently return corrupted data to users, lose data, propagate corrupted data to intact replicas, become unavailable, or return an unexpected error on queries. For example, a single write error during log initialization can cause write unavailability in ZooKeeper. Similarly, corrupted data in one node in Redis and Cas-

sandra can be propagated to other intact replicas. In Kafka and RethinkDB, corruption in one node can cause a user-visible data loss.

Second, our study has enabled us to make several observations across all systems concerning file-system fault handling. Specifically, we first have found that *systems employ diverse data-integrity strategies*; while some systems carefully use checksums, others completely trust lower layers in the stack to detect and handle corruption. Second, *faults are often undetected locally, and even if detected, crashing is the most common reaction*; undetected faults on one node can lead to harmful global effects such as user-visible data corruption. Third, as mentioned above, *a single fault can have disastrous cluster-wide effects*. Although distributed storage systems replicate data and functionality across many nodes, a single file-system fault on a single node can result in harmful cluster-wide effects; surprisingly, many distributed storage systems do not consistently use redundancy as a source of recovery. Fourth, *crash and corruption handling are entangled*; systems often conflate recovering from a crash with recovering from corruption, accidentally invoking the wrong recovery subsystem to handle the fault, and ultimately leading to poor outcomes. Finally, *nuances in commonly used distributed protocols can spread corruption or data loss*; for example, we find that subtleties in the implementation of distributed protocols such as leader election, read-repair, and resynchronization can propagate corruption or data loss.

This paper contains three major contributions. First, we build a fault injection framework (CORDS) to carefully inject file-system faults into applications (§3). Second, we present a behavioral study of eight widely used modern distributed storage systems on how they react to file-system faults and also uncover numerous bugs in these storage systems(§4.1). We have contacted developers of seven systems and five of them have acknowledged the problems we found. While a few problems can be tolerated by implementation-level fixes, tolerating many others require fundamental design changes. Third, we derive a set of observations across all systems showing some of the common data integrity and error handling problems (§4.2). Our testing framework and bugs we reported are publicly available [1]. We hope that our results will lead to discussions and future research to improve the resiliency of next generation cloud storage systems.

The rest of the paper is organized as follows. First, we provide a background on file-system faults and motivate why file-system faults are important in the context of modern distributed storage systems (§2). Then, we describe our fault model and how our framework injects faults and observes behaviors (§3). Next, we present our behavior analysis and observations across systems (§4). Finally, we discuss related work (§5) and conclude (§6).

2 Background and Motivation

We first provide background on why applications running atop file systems can encounter faults during operations such as read and write. Next, we motivate why such file-system faults are important in the context of distributed storage systems and the necessity of end-to-end data integrity and error handling for these systems.

2.1 File-System Faults

The layers in a storage stack beneath the file system consist of many complex hardware and software components [2]. At the bottom of the stack is the media (a disk or a flash device). The firmware above the media controls functionalities of the media. Commands to the firmware are submitted by the device driver. File systems can encounter faults for a variety of underlying causes including media errors, mechanical and electrical problems in the disk, bugs in firmware, and problems in the bus controller [8, 9, 48, 54, 63, 79, 81]. Sometimes, corruptions can arise due to software bugs in other parts of the operating system [13], device drivers [88], and sometimes even due to bugs in file systems themselves [26].

Due to these reasons, two problems arise for file systems: *block errors*, where certain blocks are inaccessible (also called latent sector errors) and *block corruptions*, where certain blocks do not contain the expected data.

File systems can observe block errors when the disk returns an explicit error upon detecting some problem with the block being accessed (such as in-disk ECC complaining that the block has a bit rot) [9, 79]. A previous study [9] of over 1 million disk drives over a period of 32 months has shown that 8.5% of near-line disks and about 1.9% of enterprise class disks developed one or more latent sector errors. More recent results show similar errors arise in flash-based SSDs [48, 54, 81].

File systems can receive corrupted data due to a misdirected or a lost write caused by bugs in drive firmware [8, 60] or if the in-disk ECC does not detect a bit rot. Block corruptions are insidious because blocks become corrupt in a way not detectable by the disk itself. File systems, in many cases, obliviously access such corrupted blocks and silently return them to applications. Bairavasundaram et al., in a study of 1.53 million disk drives over 41 months, showed that more than 400,000 blocks had checksum mismatches [8]. Anecdotal evidence has shown the prevalence of storage errors and corruptions [18, 37, 75]. Given the frequency of storage corruptions and errors, there is a non-negligible probability for file systems to encounter such faults.

In many cases, when the file system encounters a fault from its underlying layers, it simply passes it as-is onto the applications [63]. For example, the default Linux file system, ext4, simply returns errors or corrupted data to applications when the underlying block is not accessi-

ble or is corrupted, respectively. In a few other cases, the file system may transform the underlying fault into a different one. For example, btrfs and ZFS transform an underlying corruption into an error – when an underlying corrupted disk block is accessed, the application will receive an error instead of corrupted data [98]. In either case, we refer to these faults thrown by the file system to its applications as *file-system faults*.

2.2 Why Distributed Storage Systems?

Given that local file systems can return corrupted data or errors, the responsibility of data integrity and proper error handling falls to applications, as they care about safely storing and managing critical user data. Most single-machine applications such as stand-alone databases and non-replicated key-value storage systems solely rely on local file systems to reliably store user data; they rarely have ways to recover from local file-system faults. For example, on a read, if the local file system returns an error or corrupted data, applications have no way of recovering that piece of data. Their best possible course of action is to reliably detect such faults and deliver appropriate error messages to users.

Modern distributed storage systems, much like single-machine applications, also rely on the local file system to safely manage critical user data. However, unlike single-machine applications, distributed storage systems inherently store data in a replicated fashion. A carefully designed distributed storage system can potentially use redundancy to recover from errors and corruptions, irrespective of the support provided by its local file system. Ideally, even if one replica is corrupted, the distributed storage system as whole should not be affected as other intact copies of the same data exist on other replicas. Similarly, errors in one node should not affect the global availability of the system given that the functionality (application code) is also replicated across many nodes.

The case for end-to-end data integrity and error handling can be found in the classical end-to-end arguments in system design [78]. Ghemawat et al. also describe the need for such end-to-end checksum-based detection and recovery in the Google File System as the underlying cheap IDE disks would often corrupt data in the chunk servers [29]. Similarly, lessons from Google [22] in building large-scale Internet services emphasize how higher layer software should provide reliability. Given the possibility of end-to-end data integrity and error handling for distributed systems, we examine if and how well modern distributed storage systems employ end-to-end techniques to recover from local file-system faults.

3 Testing Distributed Systems

As we discussed in the previous section, file systems can throw errors or return corrupted data to applications running atop them; robust applications need to be able to handle such file-system faults. In this section, we first discuss our file-system fault model. Then, we describe our methodology to inject faults defined by our model and observe the effects of the injected faults.

Type of Fault		Op	Example Causes
Corruption	zeros, junk	Read	misdirected and lost writes in *ext* and *XFS*
Error	I/O error (`EIO`)	Read	latent sector errors in all file systems, disk corruptions in *ZFS*, *btrfs*
		Write	file system mounted read-only, on-disk corruptions in *btrfs*
	Space error (`ENOSPC`, `EDQUOT`)	Write	disk full, quota exceeded in all file systems

Table 1: **Possible Faults and Example Causes**. *The table shows file-systems faults captured by our model and example root causes that lead to a particular fault during read and write operations.*

3.1 Fault Model

Our fault model defines what file-system fault conditions an application can encounter. The goal of our model is to inject faults that are representative of fault conditions in current and future file systems and to drive distributed storage systems into error cases that are rarely tested.

Our fault model has two important characteristics. First, our model considers injecting exactly a *single fault* to a *single file-system block* in a *single node* at a time. While correlated file-system faults [8, 9] are interesting, we focus on the most basic case of injecting a single fault in a single node because our fault model intends to give maximum recovery leeway for applications. Correlated faults, on the other hand, might preclude such leeway.

Second, our model injects faults only into application-level on-disk structures and not file-system metadata. File systems may be able to guard their own (meta)data [27]; however, if user data becomes corrupt or inaccessible, the application will either receive a corrupted block or perhaps receive an error (if the file system has checksums for user data). Thus, it is essential for applications to handle such cases.

Table 1 shows faults that are possible in our model during read and write operations and some examples of root causes in most commonly used file systems that can cause a particular fault. For all further discussion, we use the term block to mean a file-system block.

It is possible for applications to read a block that is corrupted (with zeros or junk) if a previous write to that block was lost or some unrelated write was misdirected to that block. For example, in the ext family of file systems and XFS, there are no checksums for user data and so it is possible for applications to read such corrupted data, without any errors. Our model captures such cases by corrupting a block with zeros or junk on reads.

Even on file systems such as btrfs and ZFS where user data is checksummed, detection of corruption may be

possible but not recovery (unless mounted with special options such as *copies=2* in ZFS). Although user data checksums employed by btrfs and ZFS prevent applications from accessing corrupted data, they return errors when applications access corrupted blocks. Our model captures such cases by returning similar errors on reads. Also, applications can receive `EIO` on reads when there is an underlying latent sector error associated with the data being read. This condition is possible on all commonly used file systems including ext4, XFS, ZFS, and btrfs.

Applications can receive `EIO` on writes from the file system if the underlying disk sector is not writable and the disk does not remap sectors, if the file system is mounted in read-only mode, or if the file being written is already corrupted in btrfs. On writes that require additional space (for instance, append of new blocks to a file), if the underlying disk is full or if the user's block quota is exhausted, applications can receive `ENOSPC` and `EDQUOT`, respectively, on any file system.

Our fault model injects faults in what we believe is a realistic manner. For example, if a block marked for corruption is written, subsequent reads of that block will see the last written data instead of corrupted data. Similarly, when a block is marked for read or write error and if the file is deleted and recreated (with a possible allocation of new data blocks), we do not return errors for subsequent reads or writes of that block. Similarly, when a space error is returned, all subsequent operations that require additional space will encounter the same space error.

3.2 Methodology

We now describe our methodology to study how distributed systems react to local file-system faults. We built CORDS, a fault injection framework that consists of *errfs*, a FUSE [28] file system, and *errbench*, a set of workloads and a behavior-inference script for each system.

3.2.1 System Workloads

To study how a distributed storage system reacts to local file-system faults, we need to exercise its code paths that lead to interaction with its local file system. We crafted a workload suite, *errbench*, for this purpose; our suite consists of two workloads per system: read an existing data item, and insert or update a data item.

3.2.2 Fault Injection

We initialize the system under study to a known state by inserting a few data items and ensuring that they are safely replicated and persisted on disk. Our workloads either read or update the items inserted as part of the initialization. Next, we configure the application to run atop *errfs* by specifying its mount point as the `data-directory` of the application. Thus, all reads and writes performed by the application flow through *errfs* which can then inject faults. We run the application workload multiple times, each time injecting a single fault for a single file-system block through *errfs*.

errfs can inject two types of corruptions: corrupted with *zeros* or *junk*. For corruptions, *errfs* performs the read and changes the contents of the block that is marked for corruption, before returning to the application. *errfs* can inject three types of errors: `EIO` on reads (*read errors*), `EIO` on writes (*write errors*) or `ENOSPC` and `EDQUOT` on writes that require additional space (*space errors*). To emulate errors, *errfs* does not perform the operation but simply returns an appropriate error code.

3.2.3 Behavior Inference

For each run of the workload where a single fault is injected, we observe how the system behaves. Our system-specific behavior-inference scripts glean system behavior from the system's log files and client-visible outputs such as server status, return codes, errors (`stderr`), and output messages (`stdout`). Once the system behavior for an injected fault is known, we compare the observed behavior against expected behaviors. The following are the expected behaviors we test for:

- *Committed data should not be lost*
- *Queries should not silently return corrupted data*
- *Cluster should be available for reads and writes*
- *Queries should not fail after retries*

We believe our expectations are reasonable since a single fault in a single node of a distributed system should ideally not result in any undesirable behavior. If we find that an observed behavior does not match expectations, we flag that particular run (a combination of the workload and the fault injected) as erroneous, analyze relevant application code, contact developers, and file bugs.

Local Behavior and Global Effect. In a distributed system, multiple nodes work with their local file system to store user data. When a fault is injected in a node, we need to observe two things: local behavior of the node where the fault is injected and global effect of the fault.

In most cases, a node locally reacts to an injected fault. As shown in the legend of Figure 1, a node can *crash* or *partially crash* (only a few threads of the process are killed) due to an injected fault. In some cases, the node can fix the problem by *retrying* any failed operation or by using *internally redundant* data (cases where the same data is redundant across files within a replica). Alternatively, the node can detect and *ignore* the corrupted data or just *log an error message*. Finally, the node may *not even detect* or take any measure against a fault.

The global effect of a fault is the result that is externally visible. The global effect is determined by how distributed protocols (such as leader election, consensus, recovery, repair) react in response to the local behavior of

the faulty node. For example, even though a node can locally ignore corrupted data and lose it, the global recovery protocol can potentially fix the problem, leading to a *correct* externally observable behavior. Sometimes, because of how distributed protocols react, a global *corruption*, *data loss*, *read-unavailability*, *write-unavailability*, *unavailability*, or *query failure* might be possible. When a node simply crashes as a local reaction, the system runs with *reduced redundancy* until manual intervention.

These local behaviors and global effects for a given workload and a fault might vary depending on the role played (leader or follower) by the node where the fault is injected. For simplicity, we uniformly use the terms *leader* and *follower* instead of *master* and *slave*.

We note here that our workload suite and model are *not complete*. First, our suite consists only of simple read and write workloads while more complex workloads may yield additional insights. Second, our model does not inject all possible file-system faults; rather, it injects only a subset of faults such as corruptions, read, write, and space errors. However, even our simple workloads and fault model drive systems into corner cases, leading to interesting behaviors. Our framework can be extended to incorporate more complex faults and our workload suite can be augmented with more complex workloads; we leave this as an avenue for future work.

4 Results and Observations

We studied eight widely used distributed storage systems: Redis (v3.0.4), ZooKeeper (v3.4.8), Cassandra (v3.7), Kafka (v0.9), RethinkDB (v2.3.4), MongoDB (v3.2.0), LogCabin (v1.0), and CockroachDB (beta-20160714). We configured all systems to provide the highest safety guarantees possible; we enabled checksums, synchronous replication, and synchronous disk writes. We configured all systems to form a cluster of three nodes and set the replication factor as three.

We present our results in four parts. First, we present our detailed behavioral analysis and a qualitative summary for each system (§4.1). Second, we derive and present a set of observations related to data integrity and error handling *across* all eight systems (§4.2). Next, we discuss features of current file systems that can impact the problems we found (§4.3). Finally, we discuss why modern distributed storage systems are not tolerant of single file-system faults and describe our experience interacting with developers (§4.4).

4.1 System Behavior Analysis

Figure 1 shows the behaviors for all systems when faults are injected into different on-disk structures. The on-disk structure names shown on the right take the form: *file_name.logical_entity*. We derive the logical entity name from our understanding of the on-disk format of the file. If a file can be contained in a single file-system block, we do not show the logical entity name.

Interpreting Figure 1: We guide the reader to relevant portions of the figure for a few structures for one system (Redis). When there are *corruptions* in metadata structures in the appendonly file or *errors* in accessing the same, the node simply crashes (first row of local behavior boxes for both workloads in Redis). If the leader crashes, then the cluster becomes unavailable and if the followers crash, the cluster runs with reduced redundancy (first row of global effect for both workloads). *Corruptions* in user data in the appendonly file are undetected (second row of local behavior for both workloads). If the leader is corrupted, it leads to a global user-visible corruption, and if the followers are corrupted, there is no harmful global effect (second row of global effect for read workload). In contrast, *errors* in appendonly file user data lead to crashes (second row of local behavior for both workloads); crashes of leader and followers lead to cluster unavailability and reduced redundancy, respectively (second row of global effect for both workloads).

We next qualitatively summarize the results in Figure 1 for each system.

4.1.1 Redis

Redis is a popular data structure store, used as database, cache, and message broker. Redis uses a simple appendonly file (*aof*) to log user data. Periodic snapshots are taken from the *aof* to create a redis database file (*rdb*). During startup, the followers re-synchronize the *rdb* file from the leader. Redis does not elect a leader automatically when the current leader fails.

Summary and Bugs: Redis does not use checksums for *aof* user data; thus, it does not detect corruptions. Figure 2(a) shows how the re-synchronization protocol propagates corrupted user data in *aof* from the leader to the followers leading to a global user-visible corruption. If the followers are corrupted, the same protocol unintentionally fixes the corruption by fetching the data from the leader. Corruptions in metadata structures in *aof* and errors in *aof* in leader causes it to crash, making the cluster unavailable. Since the leader sends the *rdb* file during re-synchronization, corruption in the same causes both the followers to crash. These crashes ultimately make the cluster unavailable for writes.

4.1.2 ZooKeeper

ZooKeeper is a popular service for storing configuration information, naming, and distributed synchronization. It uses *log* files to append user data; the first block of the log contains a header, the second contains the transaction body, and the third contains the transaction tail along with ACLs and other information.

Summary and Bugs: ZooKeeper can detect corruptions in the log using checksums but reacts by simply crash-

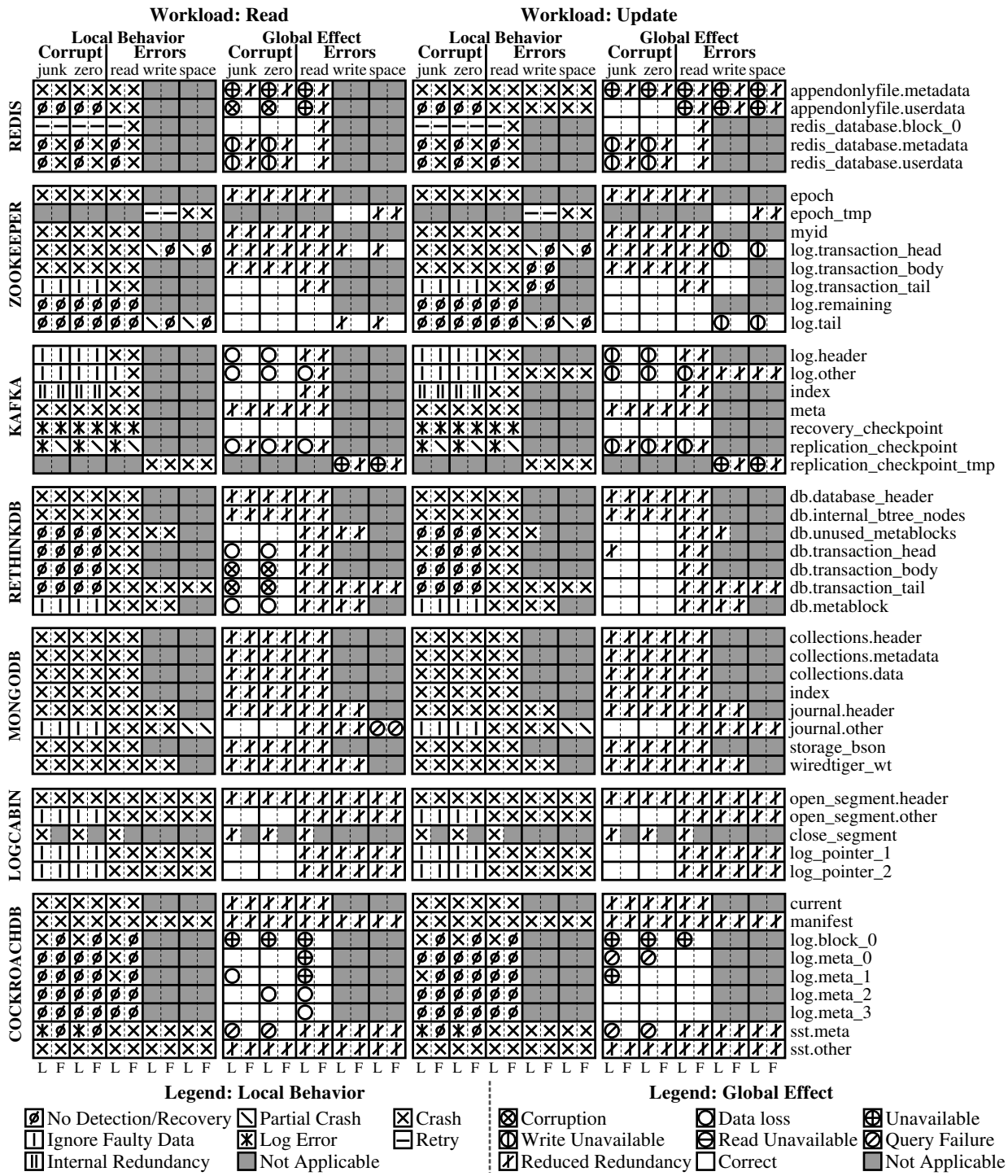


Figure 1: **System Behaviors.** *The figure shows system behaviors when corruptions (corrupted with either junk or zeros), read errors, write errors, and space errors are injected in various on-disk logical structures. The leftmost label shows the system name. Within each system workload (read and update), there are two boxes – first, local behavior of the node where the fault is injected and second, cluster-wide global effect of the injected fault. The rightmost annotation shows the on-disk logical structure in which the fault is injected. It takes the following form: file_name.logical_entity. If a file can be contained in a single file-system block, we do not show the logical entity name. Annotations on the bottom show where a particular fault is injected (L - leader/master, F - follower/slave). A gray box for a fault and a logical structure combination indicates that the fault is not applicable for that logical structure. For example, write errors are not applicable for the epoch structure in ZooKeeper as it is not written and hence shown as a gray box.*

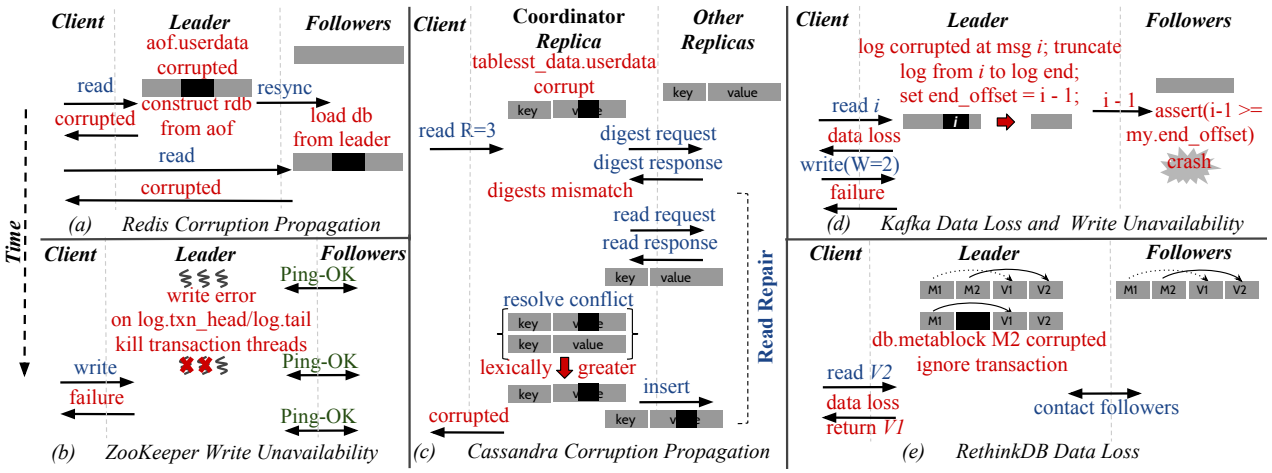


Figure 2: **Example Bugs.** *The figure depicts some of the bugs we discovered in Redis, ZooKeeper, Cassandra, Kafka, and RethinkDB. Time flows downwards as shown on the left. The black portions denote corruption.*

ing. Similarly, it crashes in most error cases, leading to reduced redundancy. In all crash scenarios, ZooKeeper can reliably elect a new leader, thus ensuring availability. ZooKeeper ignores a transaction locally when its tail is corrupted; the leader election protocol prevents that node from becoming the leader, avoiding undesirable behaviors. Eventually, the corrupted node repairs its log by contacting the leader, leading to correct behavior.

Unfortunately, ZooKeeper does not recover from write errors to the transaction head and log tail (Figure 1 – rows four and eight in ZooKeeper). Figure 2(b) depicts this scenario. On write errors during log initialization, the error handling code tries to gracefully shutdown the node but kills only the transaction processing threads; the quorum thread remains alive (partial crash). Consequently, other nodes believe that the leader is healthy and do not elect a new leader. However, since the leader has partially crashed, it cannot propose any transactions, leading to an indefinite write unavailability.

4.1.3 Cassandra

Cassandra is a Dynamo-like [23] NoSQL store. Both user data tables (tablesst) and system schema (schemasst) are stored using a variation of Log Structured Merge Trees [59]. Unlike other systems we study, Cassandra does not have a leader and followers; instead, the nodes form a ring. Hence, we show its behaviors separately in Figure 3.

Summary and Bugs: Cassandra enables checksum verification on user data only as a side effect of enabling compression. When compression is turned off, corruptions are not detected on user data (tablesst_data). On a read query, a coordinator node collects and compares digests (hash) of the data from R replicas [20]. If the digests mismatch, conflicts in the values are resolved using a *latest timestamp wins* policy. If there is a tie between timestamps, the lexically greatest value is chosen and installed on other replicas [38]. As shown in Figure 2(c), on $R = 3$, if the corrupted value is lexically greater than the original value, the corrupted value is returned to the user and the corruption is propagated to other intact replicas. On the other hand, if the corrupted value is lexically lesser, it fixes the corrupted node. Reads to a corrupted node with $R = 1$ always return corrupted data.

Faults in tablesst_index cause query failures. Faults in schema data and schema index cause the node to crash, making it unavailable for reads and writes with $R = 3$ and $W = 3$, respectively. Faults in other schema files result in query failure. In most cases, user-visible problems that are observed in $R = 1$ configuration are not fixed even when run with $R = 3$.

4.1.4 Kafka

Kafka is a distributed persistent message queue in which clients can publish and subscribe for messages. It uses a *log* to append new messages and each message is checksummed. It maintains an *index* file which indexes messages to byte offsets within the log. The *replication_checkpoint* and *recovery_checkpoint* indicate how many messages are replicated to followers so far and how many messages are flushed to disk so far, respectively.

Summary and Bugs: On read and write errors, Kafka mostly crashes. Figure 2(d) shows the scenario where Kafka can lose data and become unavailable for writes. When a log entry is corrupted on the leader (Figure 1 – rows one and two in Kafka), it locally ignores that entry and all subsequent entries in the log. The leader then instructs the followers to do the same. On receiving this instruction, the followers hit a fatal assertion and simply crash. Once the followers crash, the cluster becomes unavailable for writes and the data is also lost. Corruption in index is fixed using internal redundancy. Faults in the *replication_checkpoint* of the leader results in a data loss as the leader is unable to record the replication offsets

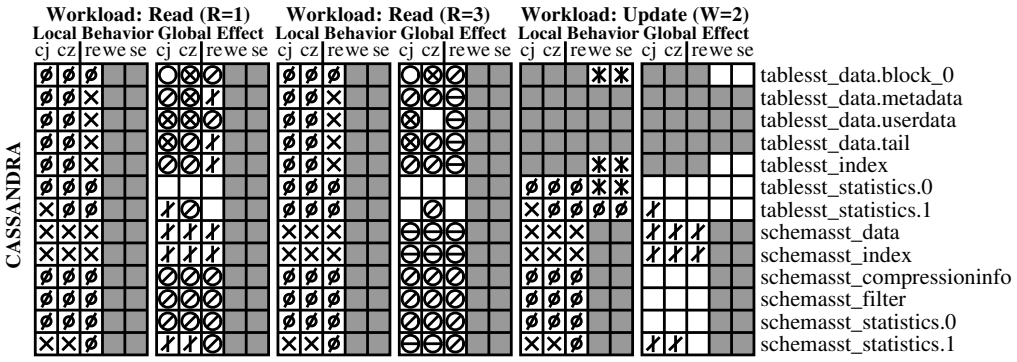


Figure 3: **System Behavior: Cassandra.** *The figure shows system behaviors when corruptions (corrupted with either junk (cj) or zeros(cz)), read errors (re), write errors (we), and space errors (se) are injected in various on-disk logical structures for Cassandra. The legend for local behaviors and global effects is the same as shown in Figure 1.*

of the followers. Kafka becomes unavailable when the leader cannot read or write *replication_checkpoint* and *replication_checkpoint_tmp*, respectively.

4.1.5 RethinkDB

RethinkDB is a distributed database suited for pushing query results to real-time web applications. It uses a persistent B-tree to store all data. *metablocks* in the B-tree point to the data blocks that constitute the current and the previous version of the database. During an update, new data blocks are carefully first written and then the *metablock* with checksums is updated to point to the new blocks, thus enabling atomic updates.

Summary and Bugs: On any fault in database header and internal B-tree nodes, RethinkDB simply crashes. If the leader crashes, a new leader is automatically elected. RethinkDB relies on the file system to ensure the integrity of data blocks; hence, it does not detect corruptions in transaction body and tail (Figure 1 – rows five and six in RethinkDB). When these blocks of the leader are corrupted, RethinkDB silently returns corrupted data.

Figure 2(e) depicts how data is silently lost when the transaction head or the metablock pointing to the transaction is corrupted on the leader. Even though there are intact copies of the same data on the followers, the leader does not fix its corrupted or lost data, even when we perform the reads with *majority* option. When the followers are corrupted, they are not fixed by contacting the leader. Although this does not lead to an immediate user-visible corruption or loss (because the leader's data is the one finally returned), it does so when the corrupted follower becomes the leader in the future.

4.1.6 MongoDB

MongoDB is a popular replicated document store that uses WiredTiger [53] underneath for storage. When an item is inserted or updated, it is added to the *journal* first; then, it is checkpointed to the *collections* file.

Summary and Bugs: MongoDB simply crashes on most errors, leading to reduced redundancy. A new leader is automatically elected if the current leader crashes. MongoDB employs checksums for all files; corruption in any block of any file causes a checksum mismatch and an eventual crash. One exception to the above is when blocks other than journal header are corrupted (Figure 1 – the sixth row in MongoDB). In this case, MongoDB detects and ignores the corrupted blocks; then, the corrupted node truncates its corrupted journal, descends to become a follower, and finally repairs its journal by contacting the leader. In a corner case where there are space errors while appending to the journal, queries fail.

4.1.7 LogCabin

LogCabin uses the Raft consensus protocol [56] to provide a replicated and consistent data store for other systems to store their core metadata. It implements a segmented-log [77] and each segment is a file on the file system. When the current *open* segment is fully utilized, it is *closed* and a new segment is opened. Two pointer files point to the latest two versions of the log. They are updated alternately; when a pointer file is partially updated, LogCabin uses the other pointer file that points to a slightly older but consistent version of the log.

Summary and Bugs: LogCabin crashes on all read, write, and space errors. Similarly, if an *open* segment file header or blocks in a closed segment are corrupted, LogCabin simply crashes. LogCabin recognizes corruption in any other blocks in an *open* segment using checksums, and reacts by simply discarding and ignoring the corrupted entry and all subsequent entries in that segment (Figure 1 – second row in LogCabin). If a log pointer file is corrupted, LogCabin ignores that pointer file and uses the other pointer file.

In the above two scenarios, the leader election protocol ensures that the corrupted node does not become the leader; the corrupted node becomes a follower and fixes its log by contacting the new leader. This ensures that in any fault scenario, LogCabin would not globally corrupt or lose user data.

Technique	Redis	ZooKeeper	Cassandra	Kafka	RethinkDB	MongoDB	LogCabin	CockroachDB
Metadata Checksums	*P*	√	√	√	*P*	√	√	√
Data Checksums	*P*	√[a]	√[$]	√		√	√	√
Background Scrubbing								√
External Repair Tools	√		√			√		√
Snapshot Redundancy	*P*∗	*P*∗				*P*∗		

P - applicable only for some on-disk structures; a - Adler32 checksum
∗ - only for certain amount of time; $ - unused when compression is off

Table 2: **Data Integrity Strategies.** *The table shows techniques employed by modern systems to ensure data integrity of user-level application data.*

4.1.8 CockroachDB

CockroachDB is a SQL database built to survive disk, machine, and data-center failures. It uses a tuned version of RocksDB underneath for storage; the storage engine is an LSM tree that appends incoming data to a persistent *log*; the in-memory data is then periodically compacted to create the *sst* files. The *manifest* and the *current* files point to the current version of the database.

Summary and Bugs: Most of the time, CockroachDB simply crashes on corruptions and errors on any data structure, resulting in reduced redundancy. Faults in the log file on the leader can sometimes lead to total cluster unavailability as some followers also crash following the crash of the leader. Corruptions and errors in a few other log metadata can cause a data loss where CockroachDB silently returns zero rows. Corruptions in sst files and few blocks of log metadata cause queries to fail with error messages such as *table does not exist* or *db does not exist*. Overall, we found that CockroachDB has many problems in fault handling. However, the reliability may improve in future since CockroachDB is still under active development.

4.2 Observations across Systems

We now present a set of observations with respect to data integrity and error handling *across* all eight systems.

#1: Systems employ diverse data integrity strategies. Table 2 shows different strategies employed by modern distributed storage systems to ensure data integrity. As shown, systems employ an array of techniques to detect and recover from corruption. The table also shows the diversity across systems. On one end of the spectrum, there are systems that try to protect against data corruption in the storage stack by using checksums (e.g., ZooKeeper, MongoDB, CockroachDB) while the other end of spectrum includes systems that completely trust and rely upon the lower layers in the storage stack to handle data integrity problems (e.g., RethinkDB and Redis). Despite employing numerous data integrity strategies, all systems exhibit undesired behaviors.

Sometimes, *seemingly unrelated configuration settings affect data integrity*. For example, in Cassandra, checksums are verified only as a side effect of enabling compression. Due to this behavior, corruptions are not detected or fixed when compression is turned off, leading to user-visible silent corruption.

We also find that a few systems use *inappropriate checksum algorithms*. For example, ZooKeeper uses Adler32 which is suited only for error detection after decompression and can have collisions for very short strings [47]. In our experiments, we were able to inject corruptions that caused checksum collisions, driving ZooKeeper to serve corrupted data. We believe that it is not unusual to expect metadata stores like ZooKeeper to store small entities such as configuration settings reliably. In general, we believe that more care is needed to understand the robustness of possible checksum choices.

#2: Local Behavior: Faults are often undetected; even if detected, crashing is the most common local reaction. We find that faults are often locally undetected. Sometimes, this leads to an immediate harmful global effect. For instance, in Redis, corruptions in the appendonly file of the leader are undetected, leading to global silent corruption. Also, corruptions in the rdb of the leader are also undetected and, when sent to followers, causes them to crash, leading to unavailability. Similarly, in Cassandra, corruption of tablesst_data is undetected which leads to returning corrupted data to users and sometimes propagating it to intact replicas. Likewise, RethinkDB does not detect corruptions in the transaction head on the leader which leads to a global user-visible data loss. Similarly, corruption in the transaction body is undetected leading to global silent corruption. The same faults are undetected also on the followers; a global data loss or corruption is possible if a corrupted follower becomes the leader in future.

While some systems detect and react to faults purposefully, some react to faults only as a side effect. For instance, ZooKeeper, MongoDB, and LogCabin carefully detect and react to corruptions. On the other hand, Redis, Kafka, and RethinkDB sometimes react to a corruption only as a side effect of a failed deserialization.

We observe that *crashing is the most common local reaction to faults*. When systems detect corruption or encounter an error, they simply crash, as is evident from the abundance of crash symbols in local behaviors of Figure 1. Although crashing of a single node does not immediately affect cluster availability, total unavailability becomes imminent as other nodes also can fail subsequently. Also, workloads that require writing to or reading from all replicas will not succeed even if one node crashes. After a crash, simply restarting does not help if the fault is sticky; the node would repeatedly crash until manual intervention fixes the underlying problem. We

Structures	Fault Injected	Scope Affected
Redis:		
appendonlyfile.metadata	any	All[#]
appendonlyfile.userdata	read, write errors	All[#]
Cassandra:		
tablesst_data.block_0	corruptions (junk)	First Entry[$]
tablesst_index	corruptions	SSTable[#]
schemasst_compressioninfo	corruptions, read error	Table[#]
schemasst_filter	corruptions, read error	Table[#]
schemasst_statistics.0	corruptions, read error	Table[#]
Kafka:		
log.header	corruptions	Entire Log[$]
log.other	corruptions, read error	Entire Log[$*]
replication_checkpoint	corruptions, read error	All[$]
replication_checkpoint_tmp	write errors	All[#]
RethinkDB:		
db.transaction_head	corruptions	Transaction[$]
db.metablock	corruptions	Transaction[$]

[$]- data loss [#] -inaccessible [*]- starting from corrupted entry

Table 3: **Scope Affected.** *The table shows the scope of data (third column) that becomes lost or inaccessible when only a small portion of data (first column) is faulty.*

also observe that nodes are more prone to crashes on errors than corruptions.

We observe that failed operations are rarely retried. While retries help in several cases where they are used, we observe that sometimes *indefinitely retrying operations may lead to more problems*. For instance, when ZooKeeper is unable to write new epoch information (to epoch_tmp) due to space errors, it deletes and creates a new file keeping the old file descriptor open. Since ZooKeeper blindly retries this sequence and given that space errors are sticky, the node soon runs out of descriptors and crashes, reducing availability.

#3: Redundancy is underutilized: A single fault can have disastrous cluster-wide effects. Contrary to the widespread expectation that redundancy in distributed systems can help recover from single faults, we observe that even a single error or corruption can cause adverse cluster-wide problems such as total unavailability, silent corruption, and loss or inaccessibility of inordinate amount of data. Almost all systems in many cases do not use redundancy as a source of recovery and miss opportunities of using other intact replicas for recovering. Notice that all the bugs and undesirable behaviors that we discover in our study are due to injecting only a single fault in a single node at a time. Given that the data and functionality are replicated, ideally, none of the undesirable behaviors should manifest.

A few systems (MongoDB and LogCabin) automatically recover from some (not all) data corruptions by utilizing other replicas. This recovery involves synergy between the local and the distributed recovery actions. Specifically, on encountering a corrupted entry, these systems locally ignore faulty data (local recovery policy). Then, the leader election algorithm ensures that the node where a data item has been corrupted and hence ignored does not become the leader (global recovery policy). As a result, the corrupted node eventually recovers the corrupted data by fetching it from the current leader. In many situations, even these systems do not automatically recover by utilizing redundancy. For instance, LogCabin and MongoDB simply crash when closed segment or collections are corrupted, respectively.

We also find that *an inordinate amount of data can be affected when only a small portion of data is faulty*. Table 3 shows different scopes that are affected when a small portion of the data is faulty. The affected portions can be silently lost or become inaccessible. For example, in Redis, all of user data can become inaccessible when metadata in the appendonly file is faulty or when there are read and write errors in appendonly file data. Similarly, in Cassandra, an entire table can become inaccessible when small portions of data are faulty. Kafka can sometimes lose an entire log or all entries starting from the corrupted entry until the end of the log. RethinkDB loses all the data updated as part of a transaction when a small portion of it is corrupted or when the metablock pointing to that transaction is corrupted.

In summary, we find that redundancy is not effectively used as a source of recovery and the general expectation that redundancy can help availability of functionality and data is not a reality.

#4: Crash and corruption handling are entangled. We find that detection and recovery code of many systems often inadvertently try to detect and fix two fundamentally distinct problems: *crashes* and *data corruption*.

Storage systems implement crash-consistent update protocols (i.e., even in the presence of crashes during an update, data should always be recoverable and should not be corrupt or lost) [7, 61, 62]. To do this, systems carefully order writes and use checksums to detect partially updated data or corruptions that can occur due to crashes. On detecting a checksum mismatch due to corruption, all systems invariably run the crash recovery code (even if the corruption was *not* actually due to crash but rather due to a real corruption in the storage stack), ultimately leading to undesirable effects such as data loss.

One typical example of this problem is RethinkDB. RethinkDB does not use application-level checksums to handle corruption. However, it does use checksums for its metablocks to recover from *crashes*. Whenever a metablock is corrupted, RethinkDB detects the mismatch in metablock checksum and invokes its crash recovery code. The crash recovery code believes that the system crashed when the last transaction was committing. Consequently, it rolls back the committed and already-acknowledged transaction, leading to a data loss. Similarly, when the log is corrupted in Kafka, the recovery code treats the corruption as a signal of a crash; hence, it

truncates and loses all further data in the log instead of fixing only the corrupted entry. The underlying reason for this problem is the inability to differentiate corruptions due to crashes from real storage stack corruptions.

LogCabin tries to distinguish crashes from corruption using the following logic: If a block in a closed segment (a segment that is full) is corrupted, it correctly flags that problem as a corruption and reacts by simply crashing. On the other hand, if a block in an open segment (still in use to persist transactions) is corrupted, it detects it as a crash and invokes its usual crash recovery procedure. MongoDB also differentiates corruptions in collections from journal corruptions in a similar fashion. Even systems that attempt to discern crashes from corruption do not always do so correctly.

There is an important consequence of entanglement of detection and recovery of crashes and corruptions. *During corruption (crash) recovery, some systems fetch inordinate amount of data to fix the problem*. For instance, when a log entry is corrupted in LogCabin and MongoDB, they can fix the corrupted log by contacting other replicas. Unfortunately, they do so by ignoring the corrupted entry and all subsequent entries until the end of the log and subsequently fetching all the ignored data, instead of simply fetching only the corrupted entry. Since a corruption is identified as a crash during the last committing transaction, these systems assume that the corrupted entry is the last entry in the log. Similarly, Kafka followers also fetch additional data from the leader instead of only the corrupted entry.

#5: Nuances in commonly used distributed protocols can spread corruption or data loss. We find that subtleties in the implementation of commonly used distributed protocols such as leader election, read-repair [23], and re-synchronization can propagate corruption or data loss.

For instance, in Kafka, a local data loss in one node can lead to a global data loss due to the subtleties in its leader election protocol. Kafka maintains a set of *in-sync-replicas* (ISR) and any node in this set can become the leader. When a log entry is corrupted on a Kafka node, it ignores the current and all subsequent entries in the log and truncates the log until the last correct entry. Logically, now this node should not be part of the ISR as it has lost some log entries. However, this node is not removed from the ISR and so eventually can still become the leader and silently lose data. This behavior is in contrast with leader election protocols of ZooKeeper, MongoDB, and LogCabin where a node that has ignored log entries do *not* become the leader.

Read-repair protocols are used in Dynamo-style quorum systems to fix any replica that has stale data. On a read request, the coordinator collects the digest of the data being read from a configured number of replicas. If all digests match, then the local data from the coordinator is simply returned. If the digests do not match, an internal conflict resolution policy is applied, and the resolved value is installed on replicas. In Cassandra, which implements read-repair, the conflict resolution resolves to the lexically greater value; if the injected corrupted bytes are lexically greater than the original value, the corrupted value is propagated to all other intact replicas.

Similarly, in Redis, when a data item is corrupted on the leader, it is not detected. Subsequently, the re-synchronization protocol propagates the corrupted data to the followers from the leader, overriding the correct version of data present on the followers.

4.3 File System Implications

All the bugs that we find can occur on XFS and all ext file systems including ext4, the default Linux file system. Given that these file systems are commonly used as local file systems in replicas of large distributed storage deployments and recommended by developers [50, 55, 64, 76], our findings have important implications for such real-world deployments.

File systems such as btrfs and ZFS employ checksums for user data; on detecting a corruption, they return an error instead of letting applications silently access corrupted data. Hence, bugs that occur due to an injected block corruption will not manifest on these file systems. We also find that applications that use end-to-end checksums when deployed on such file systems, surprisingly, lead to poor interactions. Specifically, applications crash more often due to errors than corruptions. In the case of corruption, a few applications (e.g., LogCabin, ZooKeeper) can use checksums and redundancy to recover, leading to a correct behavior; however, when the corruption is transformed into an error, these applications crash, resulting in reduced availability.

4.4 Discussion

We now consider why distributed storage systems are not tolerant of single file-system faults. In a few systems (e.g., RethinkDB and Redis), we find that the primary reason is that they expect the underlying storage stack layers to reliably store data. As more deployments move to the cloud where reliable storage hardware, firmware, and software might not be the reality, storage systems need to start employing end-to-end integrity strategies.

Next, we believe that recovery code in distributed systems is not rigorously tested, contributing to undesirable behaviors. Although many systems employ checksums and other techniques, recovery code that exercises such machinery is not carefully tested. We advocate future distributed systems need to rigorously test failure recovery code using fault injection frameworks such as ours.

Third, although a body of research work [25, 79, 83, 84, 94] and enterprise storage systems [49, 57, 58] pro-

vide software guidelines to tackle partial faults, such wisdom has not filtered down to commodity distributed storage systems. Our findings provide motivation for distributed systems to build on existing research work to practically tolerate faults other than crashes [17, 44, 97].

Finally, although redundancy is effectively used to provide improved availability, it remains underutilized as a source of recovery from file-system and other partial faults. To effectively use redundancy, first, the on-disk data structures have to be carefully designed so that corrupted or inaccessible parts of data can be identified. Next, corruption recovery has to be decoupled from crash recovery to fix only the corrupted or inaccessible portions of data. Sometimes, recovering the corrupted data might be impossible if the intact replicas are not reachable. In such cases, the outcome should be defined by design rather than left as an implementation detail.

We contacted developers of the systems regarding the behaviors we found. RethinkDB and Redis rely on the underlying storage layers to ensure data integrity [68, 69]. RethinkDB intends to change the design to include application-level checksums in the future and updated the documentation to reflect the bugs we reported [71, 72] until this is fixed. They also confirmed the entanglement in corruption and crash handling [73].

The write unavailability bug in ZooKeeper discovered by CORDS was encountered by real-world users and has been fixed recently [99, 101]. ZooKeeper developers mentioned that crashing on detecting corruption was not a conscious design decision [100]. LogCabin developers also confirmed the entanglement in corruption and crash handling in open segments; they added that it is hard to distinguish a partial write from corruption in open segments [46]. Developers of CockroachDB and Kafka have also responded to our bug reports [15, 16, 39].

5 Related Work

Our work builds on four bodies of related work.

Corruptions and errors in storage stack: As discussed in §2, detailed studies on storage errors and corruptions [8, 9, 48, 54, 79, 81] motivated our work.

Fault injection: Our work is related to efforts that inject faults into systems and test their robustness [11, 32, 82, 89]. Several efforts have built generic fault injectors for distributed systems [21, 36, 86]. A few studies have shown how file systems [10, 63, 98] and applications running atop them [87, 97] react specifically to storage and memory faults. Our work draws from both bodies of work but is unique in its focus on testing behaviors of distributed systems to storage faults. We believe our work is the first to comprehensively examine the effects of storage faults across many distributed storage systems.

Testing Distributed Systems: Several distributed model checkers have succeeded in uncovering bugs in distributed systems [34, 43, 95]. CORDS exposes bugs that cannot be discovered by model checkers. Model checkers typically reorder network messages and inject crashes to find bugs; they do not inject storage-related faults. Similar to model checkers, tools such as Jepsen [42] that test distributed systems under faulty networks are complementary to CORDS. Our previous work [3] studies how file-system crash behaviors affect distributed systems. However, these faults occur only on a crash unlike block corruption and errors introduced by CORDS.

Bug Studies: A few recent bug studies [33, 96] have given insights into common problems found in distributed systems. Yuan et al. show that 34% of catastrophic failures in their study are due to unanticipated error conditions. Our results also show that systems do not handle read and write errors well; this poor error handling leads to harmful global effects in many cases. We believe that bug studies and fault injection studies are complementary to each other; while bug studies suggest constructing test cases by examining sequences of events that have led to bugs encountered in the wild, fault injection studies like ours concentrate on injecting one type of fault and uncovering new bugs and design flaws.

6 Conclusions

We show that tolerance to file-system faults is not ingrained in modern distributed storage systems. These systems are not equipped to effectively use redundancy across replicas to recover from local file-system faults; user-visible problems such as data loss, corruption, and unavailability can manifest due to a single local file-system fault. As distributed storage systems are emerging as the primary choice for storing critical user data, carefully testing them for all types of faults is important. Our study is a step in this direction and we hope our work will lead to more work on building next generation fault-resilient distributed systems.

Acknowledgments

We thank the anonymous reviewers and Hakim Weatherspoon (our shepherd) for their insightful comments. We thank the members of the ADSL and the developers of CockroachDB, LogCabin, Redis, RethinkDB, and ZooKeeper for their valuable discussions. This material was supported by funding from NSF grants CNS-1419199, CNS-1421033, CNS-1319405, and CNS-1218405, DOE grant DE-SC0014935, as well as donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Samsung, Seagate, Veritas, and VMware. Finally, we thank CloudLab [74] for providing a great environment for running our experiments. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] Cords Tool and Results. http://research.cs.wisc.edu/adsl/Software/cords/.

[2] Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Aws Albarghouthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Beyond Storage APIs: Provable Semantics for Storage Stacks. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*, Kartause Ittingen, Switzerland, May 2015.

[3] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[4] Apache. Cassandra. http://cassandra.apache.org/.

[5] Apache. Kakfa. http://kafka.apache.org/.

[6] Apache. ZooKeeper. https://zookeeper.apache.org/.

[7] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[8] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008.

[9] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, CA, June 2007.

[10] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Analyzing the Effects of Disk-Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

[11] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault Injection Experiments Using FIAT. *IEEE Transactions on Computers*, 39(4), April 1990.

[12] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. Disks for Data Centers. Technical report, Google, 2016.

[13] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[14] CockroachDB. CockroachDB. https://www.cockroachlabs.com/.

[15] CockroachDB. Disk corruptions and read-/write error handling in CockroachDB. https://forum.cockroachlabs.com/t/disk-corruptions-and-read-write-error-handling-in-cockroachdb/258.

[16] CockroachDB. Resiliency to disk corruption and storage errors. https://github.com/cockroachdb/cockroach/issues/7882.

[17] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. Practical Hardening of Crash-Tolerant Systems. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, June 2012.

[18] Data Center Knowledge. Ma.gnolia data is gone for good. http://www.datacenterknowledge.com/archives/2009/02/19/magnolia-data-is-gone-for-good/.

[19] Datastax. Netflix Cassandra Use Case. http://www.datastax.com/resources/casestudies/netflix.

[20] DataStax. Read Repair: Repair during Read Path. http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesReadRepair.html.

[21] S. Dawson, F. Jahanian, and T. Mitton. ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementations. In *Proceedings of the 2nd International Computer Performance and Dependability Symposium (IPDS '96)*, 1996.

[22] Jeff Dean. Building Large-Scale Internet Services. http://static.googleusercontent.com/media/research.google.com/en//people/jeff/SOCC2010-keynote-slides.pdf.

[23] 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 '07)*, Stevenson, WA, October 2007.

[24] Jon Elerath. Hard-disk Drives: The Good, the Bad, and the Ugly. *Commun. ACM*, 52(6), June 2009.

[25] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*, Salt Lake City, Utah, 2012.

[26] Daniel Fryer, Dai Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. Checking the Integrity of Transactional Mechanisms. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, CA, February 2014.

[27] Daniel Fryer, Kuei Sun, Rahat Mahmood, Ting-Hao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying File System Consistency at Runtime. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012.

[28] FUSE. Linux FUSE (Filesystem in Userspace) interface. https://github.com/libfuse/libfuse.

[29] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[30] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, Toronto, Ontario, Canada, August 2011.

[31] Jim Gray. Why Do Computers Stop and What Can Be Done About It? Technical Report PN87614, Tandem, June 1985.

[32] Weining Gu, Z. Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of Linux Kernel Behavior Under Errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '03)*, San Francisco, CA, June 2003.

[33] 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 ACM Symposium on Cloud Computing (SOCC '14)*, Seattle, WA, November 2014.

[34] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[35] James R Hamilton et al. On Designing and Deploying Internet-Scale Services. In *Proceedings of the 21st Annual Large Installation System Administration Conference (LISA '07)*, Dallas, Texas, November 2007.

[36] Seungjae Han, Kang G Shin, and Harold A Rosenberg. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-time Systems. In *Proceedings of the International Computer Performance and Dependability Symposium (IPDS '95)*, 1995.

[37] James Myers. Data Integrity in Solid State Drives. http://intel.ly/2cF0dTT.

[38] Jerome Verstrynge. Timestamps in Cassandra. http://docs.oracle.com/cd/B12037_01/server.101/b10726/apphard.htm.

[39] Kafka. Data corruption or EIO leads to data loss. https://issues.apache.org/jira/browse/KAFKA-4009.

[40] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for

Disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, CA, April 2004.

[41] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrisha Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Cambridge, MA, November 2000.

[42] Kyle Kingsbury. Jepsen. `http://jepsen.io/`.

[43] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[44] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: Practical Fault Tolerance Beyond Crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[45] LogCabin. LogCabin. `https://github.com/logcabin/logcabin`.

[46] LogCabin. Reaction to disk errors and corruptions. `https://groups.google.com/forum/#!topic/logcabin-dev/wqNcdj0IHe4`.

[47] Mark Adler. Adler32 Collisions. `http://stackoverflow.com/questions/13455067/horrific-collisions-of-adler32-hash`.

[48] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '15)*, Portland, Oregon, June 2015.

[49] Ningfang Mi, A. Riska, E. Smirni, and E. Riedel. Enhancing Data Availability in Disk Drives through Background Activities. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

[50] Michael Rubin. Google moves from ext2 to ext4. `http://lists.openwall.net/linux-ext4/2010/01/04/8`.

[51] MongoDB. MongoDB. `https://www.mongodb.org/`.

[52] MongoDB. MongoDB at eBay. `https://www.mongodb.com/presentations/mongodb-ebay`.

[53] MongoDB. MongoDB WiredTiger. `https://docs.mongodb.org/manual/core/wiredtiger/`.

[54] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? And Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16)*, Haifa, Israel, June 2016.

[55] Netflix. Cassandra at Netflix. `http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html`.

[56] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, June 2014.

[57] Oracle. Fusion-IO Data Integrity. `https://blogs.oracle.com/linux/entry/fusion_io_showcases_data_integrity`.

[58] Oracle. Preventing Data Corruptions with HARD. `http://docs.oracle.com/cd/B12037_01/server.101/b10726/apphard.htm`.

[59] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4), 1996.

[60] Bernd Panzer-Steindel. Data integrity. *CERN/IT*, 2007.

[61] 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, CO, October 2014.

[62] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *The Proceedings of the International Conference on Dependable Systems and Networks (DSN-2005)*, Yokohama, Japan, June 2005.

[63] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, UK, October 2005.

[64] Rahul Bhartia. MongoDB on AWS Guidelines and Best Practices. http://media.amazonwebservices.com/AWS_NoSQL_MongoDB.pdf.

[65] Redis. Instagram Architecture. http://highscalability.com/blog/2012/4/9/the-instagram-architecture-facebook-bought-for-a-cool-billio.html.

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

[67] Redis. Redis at Flickr. http://code.flickr.net/2014/07/31/redis-sentinel-at-flickr/.

[68] Redis. Silent data corruption in Redis. https://github.com/antirez/redis/issues/3730.

[69] RethinkDB. Integrity of read results. https://github.com/rethinkdb/rethinkdb/issues/5925.

[70] RethinkDB. RethinkDB. https://www.rethinkdb.com/.

[71] RethinkDB. RethinkDB Data Storage. https://www.rethinkdb.com/docs/architecture/#data-storage.

[72] RethinkDB. RethinkDB Doc Issues. https://github.com/rethinkdb/docs/issues/1167.

[73] RethinkDB. Silent data loss on metablock corruptions. https://github.com/rethinkdb/rethinkdb/issues/6034.

[74] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), 2014.

[75] Robert Harris. Data corruption is worse than you know. http://www.zdnet.com/article/data-corruption-is-worse-than-you-know/.

[76] Ron Kuris. Cassandra From tarball to production. http://www.slideshare.net/planetcassandra/cassandra-from-tarball-to-production-2.

[77] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[78] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4), 1984.

[79] 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 '10)*, San Jose, CA, February 2010.

[80] 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 '07)*, San Jose, CA, February 2007.

[81] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16)*, Santa Clara, CA, February 2016.

[82] D.P. Siewiorek, J.J. Hudak, B.H. Suh, and Z.Z. Segal. Development of a Benchmark to Measure System Robustness. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, June 1993.

[83] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. Ensuring Data Integrity in Storage: Techniques and Applications. In *The 1st International Workshop on Storage Security and Survivability (StorageSS '05)*, FairFax County, Virginia, November 2005.

[84] Mike J. Spreitzer, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Douglas B. Terry. Dealing with Server Corruption in Weakly Consistent Replicated Data Systems. *Wirel. Netw.*, 5(5), October 1999.

[85] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey, March 2015.

[86] David T. Stott, Benjamin Floering, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *Proceedings of the 4th International Computer Performance and Dependability Symposium (IPDS '00)*, Chicago, IL, 2000.

[87] Sriram Subramanian, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S Gunawi, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Jeffrey F Naughton. Impact of Disk Corruption on Open-Source DBMS. In *Proceedings of the 26th International Conference on Data Engineering (ICDE '10)*, Long Beach, CA, March 2010.

[88] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[89] T. K. Tsai and R. K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation: Quantitative Evaluation of Computing and Communication Systems (MMB '95)*, London, UK, September 1995.

[90] Twitter. Kafka at Twitter. `https://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time`.

[91] Uber. The Uber Engineering Tech Stack, Part I: The Foundation. `https://eng.uber.com/tech-stack-part-one/`.

[92] Uber. The Uber Engineering Tech Stack, Part II: The Edge And Beyond. `https://eng.uber.com/tech-stack-part-two/`.

[93] Voldemort. Project Voldemort. `http://www.project-voldemort.com/voldemort/`.

[94] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the Salus Scalable Block Store. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, April 2013.

[95] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, Boston, MA, April 2009.

[96] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[97] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. ViewBox: Integrating Local File Systems with Cloud Storage Services. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, CA, February 2014.

[98] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, CA, February 2010.

[99] ZooKeeper. Cluster unavailable on space and write errors. `https://issues.apache.org/jira/browse/ZOOKEEPER-2495`.

[100] ZooKeeper. Crash on detecting a corruption. `http://mail-archives.apache.org/mod_mbox/zookeeper-dev/201701.mbox/browser`.

[101] ZooKeeper. Zookeeper service becomes unavailable when leader fails to write transaction log. `https://issues.apache.org/jira/browse/ZOOKEEPER-2247`.

Omid, Reloaded: Scalable and Highly-Available Transaction Processing

Ohad Shacham
Yahoo Research

Francisco Perez-Sorrosal
Yahoo

Edward Bortnikov
Yahoo Research

Eshcar Hillel
Yahoo Research

Idit Keidar
Technion and Yahoo Research

Ivan Kelly
Midokura

Matthieu Morel
Skyscanner

Sameer Paranjpye
Arimo

Abstract

We present Omid – a transaction processing service that powers web-scale production systems at Yahoo. Omid provides ACID transaction semantics on top of traditional key-value storage; its implementation over Apache HBase is open sourced as part of Apache Incubator. Omid can serve hundreds of thousands of transactions per second on standard mid-range hardware, while incurring minimal impact on the speed of data access in the underlying key-value store. Additionally, as expected from always-on production services, Omid is highly available.

1 Introduction

In recent years, there is an increased focus on supporting large-scale distributed transaction processing; examples include [6, 7, 11, 17, 18, 20, 28]. Transaction systems have many industrial applications, and the need for them is on the rise in the big data world. One prominent use case is Internet-scale data processing pipelines, for example, real-time indexing for web search [31]. Such systems process information in a streamed fashion, and use shared storage in order to facilitate communication between processing stages. Quite often, the different stages process data items in parallel, and their execution is subject to data races. Overcoming such race conditions at the application level is notoriously complex; the system design is greatly simplified by using the abstraction of transactions with well-defined *atomicity*, *consistency*, *isolation*, and *durability* (ACID) semantics [27].

We present Omid, an ACID transaction processing system for key-value stores. Omid has replaced an initial prototype bearing the same name, to which we refer here as Omid1 [25], as Yahoo's transaction processing engine; it has been entirely re-designed for scale and reliability, thereby bearing little resemblance with the origin (as discussed in Section 3 below). Omid's open source version recently became an Apache Incubator project[1].

Internally, Omid powers Sieve[2], Yahoo's web-scale content management platform for search and personalization products. Sieve employs thousands of tasks to digest billions of events per day from a variety of feeds and push them into a real-time index in a matter of seconds. In this use case, tasks need to execute as ACID transactions at a high throughput [31].

The system design has been driven by several important business and deployment considerations. First, guided by the principle of separation of concerns, Omid was designed to leverage battle-tested key-value store technology and support transactions over data stored therein, similar to other industrial efforts [6, 31, 17]. While Omid's design is compatible with multiple NoSQL key-value stores, the current implementation works with Apache HBase [1].

A second consideration was simplicity, in order to make the service easy to deploy, support, maintain, and monitor in production. This has led to a design based on a centralized *transaction manager* (*TM*)[3]. While its clients and data storage nodes are widely-distributed and fault-prone, Omid's centralized TM provides a single source of truth regarding the transaction history, and facilitates conflict resolution among updating transactions (read-only transactions never cause aborts).

Within these constraints, it then became necessary to find novel ways to make the service scalable for throughput-oriented workloads, and to ensure its continued availability following failures of clients, storage nodes, and the TM. Omid's main contribution is in providing these features:

Scalability Omid runs hundreds of thousands of transactions per second over multi-petabyte shared stor-

[1] http://omid.incubator.apache.org

[2] http://yahoohadoop.tumblr.com/post/129089878751

[3] The TM is referred to as Transaction Status Oracle (TSO) in the open source code and documentation.

age. As in other industrial systems [31, 25, 6], scalability is improved by providing *snapshot isolation* (SI) rather than serializability [27] and separating data management from control. Additionally, Omid employs a unique combination of design choices in the control plane: (i) synchronization-free transaction processing by a single TM, (ii) scale-up of the TM's in-memory conflict detection (deciding which transactions may commit) on multi-core hardware, and (iii) scale-out of metadata (HBase).

High availability The data tier is available by virtue of HBase's reliability, and the TM is implemented as a primary-backup process pair with shared access to critical metadata. Our solution is unique in tolerating a potential overlap period when two processes act as primaries, and at the same time avoiding costly synchronization (consensus), as long as a single primary is active. Note that, being generic, the data tier is not aware of the choice of primary and hence serves operations of both TMs in case of such overlap.

We discuss Omid's design considerations in Section 2 and related transaction processing systems in Section 3. We detail the system guarantees in Section 4. Section 5 describes Omid's transaction protocol, and Section 6 discusses high-availability. An empirical evaluation is given in Section 7. We conclude, in Section 8, by discussing lessons learned from Omid's production deployment and our interaction with the open source community, as well as future developments these lessons point to.

2 Design Principles and Architecture

Omid was incepted with the goal of adding transactional access on top of HBase, though it can work with any strongly consistent key-value store that provides multi-versioning with version manipulation and atomic *putIfAbsent* insertions as we now describe.

The underlying data store offers *persistence* (using a write-ahead-log), *scalability* (via sharding), and *high availability* (via replication) of the data plane, relieving Omid to manage only the transaction control plane. Omid further relies on the underlying data store for fault-tolerant and persistent storage of transaction-related metadata. This metadata includes a dedicated table that holds a single record per committing transaction, and in addition, per-row metadata for items accessed transactionally. The Omid architecture is illustrated in Figure 1.

Omid leverages *multi-versioning* in the underlying key-value store in order to allow transactions to read consistent snapshots of changing data as needed for snapshot isolation. The store's API allows users to manipulate versions explicitly. It supports atomic *put(key, val, ver)* and *putIfAbsent(key, val, ver)* operations for updating or inserting a new item with a specific version, and an atomic *get(key, ver)* operation for retrieving the item's value with highest version not exceeding *ver*. Specifically, when the item associated with an existing key is overwritten, the new version (holding the key, its new value, and a new version number) is created, while the previous version persists. An old version might be required as long as there is some active transaction that had begun before the transaction that overwrote this version has committed. Though this may take a while, overwritten versions eventually become obsolete. A cleaning process, (in HBase, implemented as a coprocessor [2]), frees up the disk space taken up by obsolete versions.

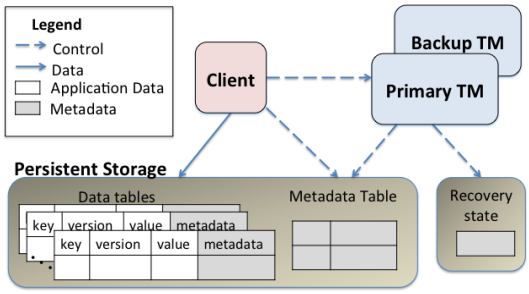


Figure 1: **Omid architecture**. Clients manipulate data that resides in data tables in the underlying data store (for example, HBase) and use the TM for conflict detection. Only the primary TM is active, and the backup is in hot standby mode. The TM maintains persistent metadata in the data store as well as separately managed recovery state (for example, using Zookeeper).

The transaction control plane is implemented by a centralized transaction manager. The TM has three roles: (i) *version (timestamp) allocation*; (ii) *conflict detection* in order to determine which transactions may commit; and (iii) *persistent logging of the commits*. The TM provides high availability via a primary-backup approach— if the primary TM becomes unresponsive, then the backup becomes the new primary and takes over. This design offers durability and high availability; it further facilitates scalability of storage and compute resources separately – metadata storage access scales out on the underlying distributed data store, whereas conflict management is done entirely in RAM, and scales up on a shared-memory multi-core server.

Our high availability solution tolerates "false" failovers, where a new primary replaces one that is simply slow, (for example, due to a garbage collection stall), leading to a period with two active primaries. Synchronization between the two is based on shared persis-

tent metadata storage, and induces overhead only in rare cases when more than one TM acts as primary. Omid uses time-based leases in order to minimize potential overlap among primaries. The implementation employs Apache Zookeeper [4] for lease management and synchronization between primary and backup.

3 Related Work

Distributed transaction processing has been the focus of much interest in recent years. Most academic-oriented papers [7, 8, 11, 18, 20, 32, 36] build full-stack solutions, which include transaction processing as well as a data tier. Some new protocols exploit advanced hardware trends like RDMA and HTM [19, 20, 33]. Generally speaking, these solutions do not attempt to maintain separation of concerns between different layers of the software stack, neither in terms of backward compatibility nor in terms of development efforts. They mostly provide strong consistency properties such as serializability.

On the other hand, production systems such as Google's Spanner [17], Megastore [9] and Percolator [31], Yahoo's Omid1 [25], Cask's Tephra [6], and more [22, 30, 5], are inclined towards separating the responsibilities of each layer. These systems, like the current work, reuse an existing persistent highly-available data-tier; for example, Megastore is layered on top of Bigtable [16], Warp [22] uses HyperDex [21], and CockroachDB [5] uses RocksDB.

Omid most closely resembles Tephra [6] and Omid1 [25], which also run on top of a distributed key-value store and leverage a centralized TM (sometimes called *oracle*) for timestamp allocation and conflict resolution. However, Omid1 and Tephra store all the information about committed and aborted transactions in the TM's RAM, and proactively duplicate it to every client that begins a transaction (in order to allow the client to determine locally which non-committed data should be excluded from its reads). This approach is not scalable, as the information sent to clients can consist of many megabytes. Omid avoids such bandwidth overhead by storing pertinent information in a metadata table that clients can access as needed. Our performance measurements in Section 7 below show that Omid significantly out-performs Omid1, whose design is very close to Tephra's. For high availability, Tephra and Omid1 use a write-ahead log, which entails long recovery times for replaying the log; Omid, instead, reuses the inherent availability of the underlying key-value store, and hence recovers quickly from failures.

Percolator also uses a centralized "oracle" for timestamp allocation but resolves conflicts via two-phase commit, whereby clients lock database records rendering them inaccessible to other transactions; the Percolator paper does not discuss high availability. Other systems like Spanner and CockroachDB allot globally increasing timestamps using a (somewhat) synchronized clock service. Spanner also uses two-phase commit whereas CockroachDB uses distributed conflict resolution where read-only transactions can cause concurrent update transactions to abort. In contrast, Omid never locks (or prevents access to) a database record, and never aborts due to conflicts with read-only transactions.

The use cases production systems serve allow them to provide SI [31, 25, 6, 5], at least for read-only transactions [17]. It is nevertheless straightforward to extend Omid to provide serializability, similarly to a serializable extension of Omid1 [35] and Spanner [17]; it is merely a matter of extending the conflict analysis to cover read-sets [24, 14], which may degrade performance.

A number of other recent efforts avoid the complexity of two-phase commit [26] by serializing transactions using a global serialization service such as highly-available log [11, 23, 13] or totally-ordered multicast [15]. Omid is unique in utilizing a single transaction manager to resolve conflicts in a scalable way.

4 Service Semantics and Interface

Omid provides transactional access to a large collection of persistent data items identified by unique keys. The service is highly available, whereas its clients are ephemeral, i.e., they are alive only when performing operations and may fail at any time.

Semantics. A *transaction* is a sequence of put and get operations on different objects that ensures the so-called ACID properties: *atomicity* (all-or-nothing execution), *consistency* (preserving each object's semantics), *isolation* (in that concurrent transactions do not see each other's partial updates), and *durability* (whereby updates survive crashes).

Different isolation levels can be considered for the third property. Omid opts for snapshot isolation [12], which is provided by popular database technologies such as Oracle, PostgreSQL, and SQL Server. Note that under SI, concurrent transactions conflict only if they *update* the same item, whereas with serializability, a transaction that updates an item conflicts with transactions that *get* that item. Thus, for read-dominated workloads, SI is amenable to implementations (using multi-versioned concurrency control) that allow more concurrency than serializable ones, and hence scale better.

API. Omid's client API offers abstractions both for control (*begin, commit*, and *abort*) and for data access (*get* and *put*). Following a commit call, the transaction

may successfully *commit*, whereby all of its operations take effect; in case of conflicts, (i.e., when two concurrent transactions attempt to update the same item), the transaction may *abort*, in which case none of its changes take effect. An abort may also be initiated by the programmer.

5 Transaction Processing

We now explain how Omidmanages transactions so as to guarantee SI semantics. For clarity of the exposition, we defer discussion of the TM's reliability to the next section; for now, let us assume that this component never fails. We describe Omid's data model in Section 5.1, then proceed to describe the client operation in Section 5.2 and the TM's operation in Section 5.3.

5.1 Data and metadata

Omid employs optimistic concurrency control with commit-time conflict resolution. Intuitively, with SI, a transaction's reads all appear to occur the time when it begins, while its writes appear to execute when it commits. Omid therefore associates two timestamps with each transaction: a *read timestamp* ts_r when it begins, and a *commit timestamp* ts_c upon commit. Both are provided by the TM using a logical clock it maintains. In addition, each transaction has a unique transaction id *txid*, for which we use the read timestamp; in order to ensure its uniqueness, the TM increments the clock whenever a transaction begins.

The data store is multi-versioned. A write operation by a transaction starting at some time t needs to be associated with a version number that exceeds all those written by transactions that committed before time t. However, the version order among concurrent transactions that attempt to update the same key is immaterial, since at least one of these transactions is doomed to abort. To ensure the former, we use the writing transaction's *txid*, which exceeds those of all previously committed transactions, as the version number.

Since transaction commit needs to be an atomic step, Omid tracks the list of committed transactions in a persistent *Commit Table (CT)*, as shown in Table 1, which in our implementation is also stored in HBase. Each entry in the CT maps a committed transaction's *txid* to its respective ts_c. To commit a transaction, the TM writes the $(txid, ts_c)$ pair to the CT, which makes the transaction durable, and is considered its commit point. Gets refer to the CT using the *txid* in the data record in order to find out whether a read value has been committed. In case it has, they use the commit timestamp to decide whether the value appears in their snapshot.

Data Table

key	value	version (*txid*)	commit (cf)
k_1	a	**5**	7
k_1	b	**8**	nil

Commit Table

txid	commit *ts*
5	7

Table 1: **Omid data and metadata**. Data is multi-versioned, with the *txid* as the version number. The *commit field* indicates whether the data is committed, and if so, its commit timestamp. The commit table (CT) maps incomplete committed transaction ids to their respective commit timestamps. Transaction 5 has already committed and updated cf for k_1, but has not yet removed itself from CT; transaction 8 is still pending.

While checking the CT for every read ensures correctness, it imposes communication costs and contention on the CT. To avoid this overhead, Omid augments each record in the data store with a *commit field (cf)*, indicating whether the data is committed, and if it is, its commit timestamp. Initially the commit field is *nil*, indicating that the write is *tentative*, i.e., potentially uncommitted. Following a commit, the transaction updates the commit fields of its written data items with its ts_c, and then removes itself from the CT. Only then, the transaction is considered complete. A background cleanup process helps old (crashed or otherwise slow) committed transactions complete.

Table 1 shows an example of a key k_1 with two versions, the second of which is tentative. A transaction that encounters a tentative write during a read still refers to the CT in order to find out whether the value has been committed. In case it has, it helps complete the transaction that wrote it by copying its ts_c to the commit field. The latter is an optimization that might reduce accesses to the commit table by ensuing transactions.

5.2 Client-side operation

Transactions proceed optimistically and are validated at commit time. In the course of a transaction, a client's get operations read a snapshot reflecting the data store state at their read timestamp, while put operations write tentative values with *txid*. Since SI needs to detect only write-write conflicts, only the transaction's *write-set* is tracked. The operations, described in pseudocode in Algorithm 1, execute as follows:

Begin. The client obtains from the TM a read timestamp ts_r, which also becomes its transaction id (*txid*). The TM ensures that this timestamp exceeds all the commit timestamps of committed transactions and precedes all commit timestamps that will be assigned to committing transactions in the future.

Algorithm 1 Omid's client-side code.

```
1:  local variables txid, write-set
2:  procedure BEGIN
3:      txid ← TM.BEGIN()
4:      write-set ← ∅
5:  procedure PUT(key, value)
6:      ds.put(key, value, txid, nil)
7:      add 64-bit hash of key to write-set
8:  procedure GET(key)
9:      for rec ← ds.get(key, versions down from ts_r) do
10:         if rec.commit≠nil then                    ▷ not tentative
11:             if rec.commit < ts_r then
12:                 return rec.value
13:         else                                      ▷ tentative
14:             value ← GETTENTATIVEVALUE(rec, key)
15:             if value ≠nil then
16:                 return value
17:     return nil
18: procedure GETTENTATIVEVALUE(rec,key)
19:     lookup rec.version in CT
20:     if present then                               ▷ committed
21:         update rec.commit                         ▷ helping
22:         if rec.commit < ts_r then return rec.value
23:     else                          ▷ re-read version not found in CT
24:         rec ← ds.get(key, rec.version)
25:         if rec.commit≠nil ∧ rec.commit < ts_r then
26:             return rec.value
27:     return nil
28: procedure COMMIT
29:     ts_c ← TM.COMMIT(txid, write-set)
30:     for all key in write-set do
31:         rec ← ds.get(key, txid)
32:         if ts_c = ⊥ then                          ▷ abort
33:             remove rec
34:         else
35:             rec.cf ← ts_c
36:     remove record with txid from CT
```

Put(key,val). The client adds the tentative record to the data store (line 6) and tracks the key in its local *write-set*. To reduce memory and communication overheads, we track 64-bit hashes rather than full keys.

Get(key). A get reads from the data store (via ds.get()) records pertaining to *key* with versions smaller than ts_r, latest to earliest (line 9), in search of the value written for this key by the latest transaction whose ts_c does not exceed ts_r (i.e., the latest version written by a transaction that committed before the current transaction began).

If the read value is committed with a commit timestamp lower than ts_r, it is returned (line 12). Upon encountering a tentative record (with cf=*nil*), the algorithm calls GETTENTATIVEVALUE (line 18) in order to search its ts_c in the CT. If this *txid* was not yet written, then it can safely be ignored, since it did not commit. However, a subtle race could happen if the transaction has updated the commit timestamp in the data store and then removed itself from the CT between the time the record was read and the time when the CT was checked. In order to discover this race, a record is re-read after its version is not found in the CT (line 23). In all cases, the first value encountered in the backward traversal with a commit timestamp lower than ts_r is returned.

Commit. The client requests *commit(txid, write-set)* from the TM. The TM assigns it a commit timestamp ts_c and checks for conflicts. If there are none, it commits the transaction by writing $(txid, ts_c)$ to the CT and returns a response. Following a successful commit, the client writes ts_c to the commit fields of all the data items it wrote to (indicating that they are no longer tentative), and finally deletes its record from the CT. Whether the commit is successful or not a background process helps transactions to complete or cleans their uncommitted records from the data store, thereby overcoming client failures.

5.3 TM operation

The TM uses an internal (thread-safe) clock to assign read and commit timestamps. Pseudocode for the TM's begin and commit functions is given in Algorithm 2; both operations increment the clock and return its new value. Thus, read timestamps are unique and can serve as transaction ids. Begin returns once all transactions with smaller commit timestamps are finalized, (i.e., written to the CT or aborted).

Commit involves compute and I/O aspects for conflict detection and CT update, resp. The TM uses a pipelined SEDA architecture [34] that scales each of these stages separately using multiple threads. Note that the I/O stage also benefits from such parallelism since the CT can be sharded across multiple storage nodes and yield higher throughput when accessed in parallel.

In order to increase throughput, writes to the commit table are batched. Both begin and commit operations need to wait for batched writes to complete before they can return – begin waits for all smaller-timestamped transactions to be persisted, while commit waits for the committing transaction. Thus, batching introduces a tradeoff between I/O efficiency, (i.e., throughput), and begin/commit latency.

The CONFLICTDETECT function checks for conflicts using a hash table in main memory. (The TM's compute aspect is scaled by running multiple instances of this function for different transactions, accessing the same table in separate threads.) For the sake of conflict detection, every entry in the write-set is considered a *key*, (though in practice it is a 64-bit hash of the appropriate key). Each *bucket* in the hash table holds an array of pairs, each consisting of a key hashed to this bucket and the ts_c of the transaction that last wrote to this key.

CONFLICTDETECT needs to (i) validate that none of the keys in the write-set have versions larger than *txid* in the table, and (ii) if validation is successful, update the table entries pertaining to the write-set to the transaction's newly assigned ts_c. However, this needs to be done atomically, so two transactions committing in parallel won't miss each other's updates. Since holding a lock on the entire table for the duration of the conflict detection procedure would severely limit concurrency, we instead limit the granularity of atomicity to a single bucket: for each key in the write-set, we lock the corresponding bucket (line 52), check for conflicts in that bucket (line 54), and if none are found, optimistically add the key with the new ts_c to the bucket (lines 56–61). The latter might prove redundant in case the transaction ends up aborting due to a conflict it discovers later. However, since our abort rates are low, such spurious additions rarely induce additional aborts.

Algorithm 2 TM functions.

```
37: procedure BEGIN
38:     txid = Clock.FetchAndIncrement()
39:     wait until there are no pending commit operations
40:           with ts_c < txid
41:     return txid

42: procedure COMMIT(txid, write-set)
43:     ts_c ← Clock.FetchAndIncrement()
44:     if ConflictDetect(txid, write-set) = COMMIT then
45:         UpdateCT(txid, ts_c)            ▷ proceed to I/O stage
46:         return ts_c
47:     else
48:         return ABORT

49: procedure CONFLICTDETECT(txid,write-set)
50:     for all key ∈ write-set do
51:         b ← key's bucket
52:         lock b
53:         small ← entry with smallest ts in b
54:         if ∃ (key, t) ∈ b s.t. t > txid then        ▷ conflict
55:             unlock b; return ABORT
                ▷ no conflict on key found – update hash table
56:         if ∃ (key, t) ∈ b s.t. t < txid then
57:             overwrite (key, t) with (key, ts_c)
58:         else if ∃ empty slot s ∈ b then
59:             write (key, ts_c) to s
60:         else if small.t ≤ txid then
61:             overwrite small with (key, ts_c)
62:         else                                        ▷ possible conflict
63:             unlock b; return ABORT
64:         unlock b
65:     return COMMIT
```

A second challenge is to limit the table size and garbage-collect information pertaining to old commits. Since a transaction need only check for conflicts with transactions whose ts_c exceeds its *txid*, it is safe to remove all entries that have smaller commit times than the *txid* of the oldest active transaction. Unfortunately, this observation does not give rise to a feasible garbage collection rule: though transactions usually last few tens of milliseconds, there is no upper bound on a transaction's life span, and no way to know whether a given outstanding transaction will ever attempt to commit or has failed. Instead, we use the much simpler policy of restricting the number of entries in a bucket. Each bucket holds a fixed array of the most recent (key, ts_c) pairs. In order to account for potential conflicts with older transactions, a transaction also aborts in case the minimal ts_c in the bucket exceeds its *txid* (line 62). In other words, a transaction expects to find, in every bucket it checks, at least one commit timestamp older than its start time or one empty slot, and if it does not, it aborts.

The size of the hash table is chosen so as to reduce the probability for spurious aborts, which is the probability of all keys in a given bucket being replaced during a transaction's life span. If the throughput is T transactional updates per second, a bucket in a table with e entries will overflow after e/T seconds on average. For example, if 10 million keys are updated per second, a bucket in a one-million-entry table will overflow only after 100ms on average, which is much longer than most transactions. We further discuss the impact of the table size in Section 7.

Garbage collection. A dedicated background procedure (*co-processor*) cleans up old versions. To this end, the TM maintains a *low water mark*, which is used in two ways: (1) the co-processor scans data store entries, and keeps, for each key, the biggest version that is smaller than the low water mark along with all later versions. Lower versions are removed. (2) When a transaction attempts to commit, if its *txid* is smaller than the low water mark, it aborts because the co-processor may have removed versions that ought to have been included in its snapshot. The TM attempts to increase the low water mark when the probability of such aborts is small.

6 High Availability

Very-high-end Omid-powered applications are expected to work around the clock, with a mean-time-to-recover of just a few seconds. Omid therefore needs to provide *high availability (HA)*. Given that the underlying data store is already highly available and that client failures are tolerated by Omid's basic transaction processing protocol, Omid's HA solution only needs to address TM failures. This is achieved via the primary-backup paradigm: during normal operation, a single *primary* TM handles

client requests, while a *backup* TM runs in hot standby mode. Upon detecting the primary's failure, the backup performs a *failover* and becomes the new primary.

The backup TM may falsely suspect that the primary has failed. The resulting potential simultaneous operation of more than one TM creates challenges, which we discuss in Section 6.1. We address these in Section 6.2 by adding synchronization to the transaction commit step. While such synchronization ensures correctness, it also introduces substantial overhead. We then optimize the solution in Section 6.3 to forgo synchronization during normal (failure-free) operation.

Our approach thus resembles many popular protocols, such as Multi-Paxos [29] and its variants, which expedite normal mode operation as long as an agreed leader remains operational and unsuspected. However, by relying on shared persistent state in the underlying highly available data store, we obtain a simpler solution, eliminating the need to synchronize with a quorum in normal mode or to realign state during recovery.

6.1 Failover and concurrent TMs

The backup TM constantly monitors the primary's liveness. Failure detection is timeout-based, namely, if the primary TM does not re-assert its existence within a configured period, it is deemed failed, and the backup starts acting as primary. Note that the primary and backup run independently on different machines, and the time it takes the primary to inform the backup that it is alive can be unpredictable due to network failures and processing delays, (e.g., garbage-collection stalls or long I/O operations). But in order to provide fast recovery, it is undesirable to set the timeout conservatively so as to ensure that a live primary is never detected as faulty.

We therefore have to account for the case that the backup performs failover and takes over the service while the primary is operational. Though such simultaneous operation of the two TMs is a necessary evil if one wants to ensure high availability, our design strives to reduce such overlap to a minimum. To this end, the primary TM actively checks if a backup has replaced it, and if so, "commits suicide", i.e., halts. However, it is still possible to have a (short) window between the failover and the primary's discovery of the existence of a new primary when two primary TMs are active.

When a TM fails, all the transactions that began with it and did not commit (i.e., were not logged in the CT) are deemed aborted. However, this clear separation is challenged by the potential simultaneous existence of two TMs. For example, if the TM fails while a write it issued to the CT is still pending, the new TM may begin handling new transactions before the pending write takes effect. Thus, an old transaction may end up committing

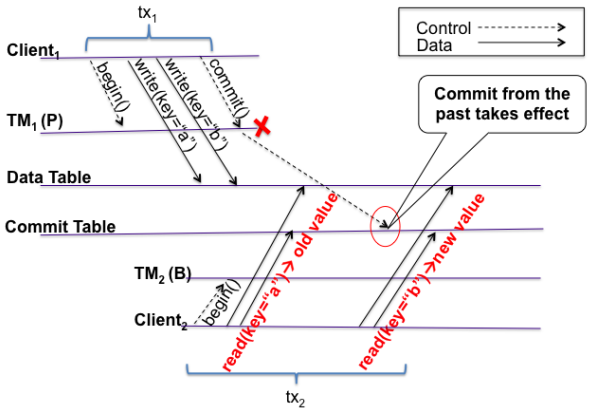


Figure 2: **The challenge with two concurrent TMs.** An old transaction, tx_1, commits while a new one tx_2 is processed, causing tx_2 to see an inconsistent snapshot.

after the new TM has begun handling new ones. Unless handled carefully, this can cause a new transaction to see partial updates of old ones, as illustrated in Figure 2. To avoid this, we must ensure that once a new transaction obtains a read timestamp, the status of all transactions with smaller commit timestamps does not change.

A straightforward way to address the above challenge is via mutual exclusion, i.e., making sure that at most one TM commits operations at a time. However, this solution would entail synchronization upon each commit, not only at failover times, which would adversely affect performance. We therefore forgo this option.

6.2 Basic HA algorithm

Upon failover from TM_1 (the old primary) to TM_2 (the new one), we strive to ensure the following properties:

P1 all timestamps assigned by TM_2 exceed all those assigned by TM_1;

P2 after a transaction tx_2 with read timestamp $ts2_r$ begins, no transaction tx_1 that will end up with a commit timestamp $ts1_c < ts2_r$ can update any additional data items (though it may still commit); and

P3 when a transaction reads a tentative update, it can determine whether this update will be committed with a timestamp smaller than its read timestamp or not.

Properties P1–P3 are sufficient for SI: P1 implies that commit timestamps continue to be totally ordered by commit time, P2 ensures that a transaction encounters every update that must be included in its snapshot, and P3 stipulates that the transaction can determine whether to return any read value.

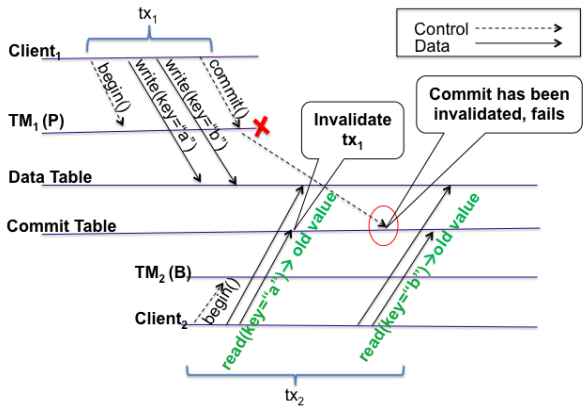


Figure 3: **Addressing the challenge of two concurrent TMs.** The old transaction is invalidated by the new one and therefore cannot commit.

To ensure the first two properties, the TMs publish the read timestamps they allot as part of initiating a transaction in a persistent shared object, *maxTS*. Before committing, the TM checks maxTS. If it finds a timestamp greater than its last committed one, it deduces that a new TM is active, aborts the transaction attempting to commit, and halts.

In Figure 2 we saw a scenario where the third property, P3, is violated— when tx_2 reads key a it cannot tell that tx_1, which wrote it, will end up committing with a smaller $ts1_c$ than $ts2_r$. This leads to an inconsistent snapshot at tx_2, as it sees the value of key b written by tx_1.

To enforce P3, tx_2 cannot wait for TM_1, because the latter might have failed. Instead, we have tx_2 proactively abort tx_1, as illustrated in Figure 3. More generally, when a read encounters a tentative update whose *txid* is not present in the CT, it forces the transaction that wrote it to abort. We call this *invalidation*, and extend the CT's schema to include an *invalid* field to this end. Invalidation is performed via an atomic put-if-absent (supported by HBase's *checkAndMutate* API) to the CT, which adds a record marking that tx_1 has "invalid" status. The use of an atomic put-if-absent achieves *consensus* regarding the state of the transaction.

Commits, in turn, read the CT after adding the commit record in order to check whether an invalidation record also exists, and if so, halt without returning a commit response to the client. In addition, every read of a tentative update checks its invalid field in the CT, and ignores the commit record if the transaction has been invalidated.

While this solution satisfies the three required properties, it also induces a large number of synchronization steps: (i) writing allotted read timestamps to maxTS to ensure P1; (ii) checking maxTS at commit time to ensure P2; and (iii) checking the CT for invalidation at the end of every commit to ensure P3. The next section presents an optimization that reduces the cost of synchronization.

6.3 Synchronization-free normal operation

In order to eliminate the synchronization overhead most of the time, Omid's HA solution uses two mechanisms. First, to reduce the overheads (i) and (ii) associated with timestamp synchronization, it allocates timestamp ranges in large chunks, called *epochs*. That is, instead of incrementing maxTS by one timestamp at a time, the TM increments it by a certain *range*, and is then free to allot timestamps in this range without further synchronization. Second, to reduce cost (iii) of checking for invalidations, it uses locally-checkable *leases*, which are essentially locks that live for a limited time. As with locks, at most one TM may hold the lease at a given time (this requires the TMs' clocks to advance roughly at the same rate). Omid manages epochs and leases as shared objects in Zookeeper, and accesses them infrequently.

Algorithm 3 summarizes the changes to support HA. On the TM side, CHECKRENEW is called at the start of every commit and begin. It first renews the lease every δ time, for some parameter δ (lines 68–70). This parameter defines the tradeoff between synchronization frequency and recovery time: the system can remain unavailable for up to δ time following a TM failure. Since clocks may be loosely synchronized, Omid defines a *guard period* of $\delta' < \delta$, so that the lease must be renewed at least δ' time before it expires. The production default for δ' is $\delta/4$. The primary TM fails itself (halts) if it cannot renew the lease prior to that time. From the clients' perspective, this is equivalent to a TM crash (line 70). Second, CHECKRENEW allocates a new epoch if needed (lines 71–74).

The backup (not shown in pseudocode) regularly checks the shared lease, and if it finds that it has expired, it immediately sets its clock to exceed maxTS, allocates a new epoch for itself (by increasing maxTS), and begins serving requests, *without any special recovery procedure*. Since the epoch claimed by a new TM always exceeds the one owned by the old one, Property P1 holds.

Property P2 is enforced by having the TM (locally) check that its lease is valid before committing a transaction (lines 68–70). Since at most one TM can hold the lease at a given time, and since the commit is initiated after all writes to items that are part of the transaction complete, Property P2 holds.

Nevertheless, the lease does not ensure Property P3, since the lease may expire while the commit record is in flight, as in the scenario of Figures 2 and 3. To this end, we use the invalidation mechanism described above. However, we limit its scope as follows: (1) A commit needs to check whether the transaction has been invali-

Algorithm 3 Omid's HA algorithm.

```
66: procedure CHECKRENEW
67:         ▷ called by the TM at start of BEGIN and COMMIT
68:     if lease < now + δ′ then
69:         renew lease for δ time                    ▷ atomic operation
70:         if failed then halt
71:     if Clock = epoch then
72:         epoch ← Clock + range
73:         extend maxTS from Clock to epoch
74:         if failed then halt

75: procedure TMCHECKINVALIDATE(txid)
76:                 ▷ called by the TM before COMMIT returns
77:     if lease < now + δ′ then
78:         if txid invalid in CT then halt

79: procedure GETTENTATIVEVALUE(REC)
80:                 ▷ replaces same function from Algorithm 1
81:     lookup rec.version in CT
82:     if present then
83:         if invalidated then return nil
84:         update rec.commit                         ▷ helping
85:         if rec.commit < ts_r then return rec.value
86:     else                ▷ new code – check if need to invalidate
87:         if rec.version ∈ old epoch by an old TM then
88:             invalidate t in CT                    ▷ try to invalidate
89:             if failed then
90:                 lookup rec.version in CT
91:                 if invalidated then return nil
92:                 update rec.commit                 ▷ helping
93:                 if rec.commit < ts_r then
94:                     return rec.value
95:             else                                  ▷ invalidated
96:                 return nil
97:         else                ▷ original code – no invalidation
98:             rec ← ds.get(key, rec.version)
99:             if rec.commit≠nil ∧ rec.commit < ts_r then
100:                return rec.value
101:    return nil
```

dated only if the TM's lease has expired. This is done in the TMCHECKINVALIDATE function. (2) A read needs to invalidate a transaction only if it pertains to an earlier epoch of a different TM. We extend client's GETTENTATIVEVALUE function to perform such invalidation in Algorithm 3 lines 83, 87–96. Note that a transaction reading a tentative update still checks its validity status regardless of the epoch, in order to avoid "helping" invalidated transactions complete their tentative updates.

Finally ,we note that on TM failover, some clients may still be communicating with the old TM. While the old TM may end up committing some of their requests, a problem arises if the client times out on the old TM before getting the commit response, since the client might unnecessarily retry a committed transaction. To avoid this problem, a client that times out on its TM checks the CT for the status of its transaction before connecting to a new TM. If the status is still undetermined, the client tries to invalidate the CT entry, thus either forcing the transaction to abort or learning that it was committed (in case the invalidation fails).

7 Evaluation

Omid's implementation complements Apache HBase with transaction processing. It exploits HBase to store both application data and the CT metadata. HBase, the TMs, and Zookeeper are all deployed on separate dedicated machines.

In large-scale deployments, HBase tables are sharded (partitioned) into multiple *regions*. Each region is managed by a *region server*; one server may serve multiple regions. HBase is deployed on top of Hadoop Distributed Filesystem (HDFS), which provides the basic abstraction of scalable and reliable storage. HDFS is replicated 3-fold in all the settings described below.

Section 7.1 presents performance statistics obtained in Omid's production deployment, focusing on the end-to-end application-level overhead introduced by transaction processing. Section 7.2 further zooms in on the TM scalability under very high loads.

7.1 End-to-end performance in production

We present statistics of Omid's use in a production deployment of Sieve – Yahoo's content management system. Sieve digests streams of documents from multiple sources, processes them, and indexes the results for use in search and personalization applications. Each document traverses a pipeline of tasks, either independently or as part of a mini-batch. A task is an ACID processing unit, framed as a transaction. It typically reads one or more data items generated by preceding tasks, performs some computation, and writes one or more artifacts back.

Sieve scales across task pipelines that serve multiple products, performing tens of thousands of tasks per second on multi-petabyte storage. All are powered by a single Omid service, with the CT sharded across 10 regions managed by 5 region servers. Sieve is throughput-oriented, and favors scalability over transaction latency.

Figure 4 presents statistics gathered for five selected Sieve tasks. For each task, we present its average latency broken down to components – HBase access (two bottom components in each bar), compute time, and the TM's begin and commit (top two components). In this deployment, Omid updates the commit fields synchronously upon commit, that is, commit returns only after the commit fields of the transaction's write-set have been updated. Note that since a begin request waits for all transactions with smaller *txids* to commit, its processing

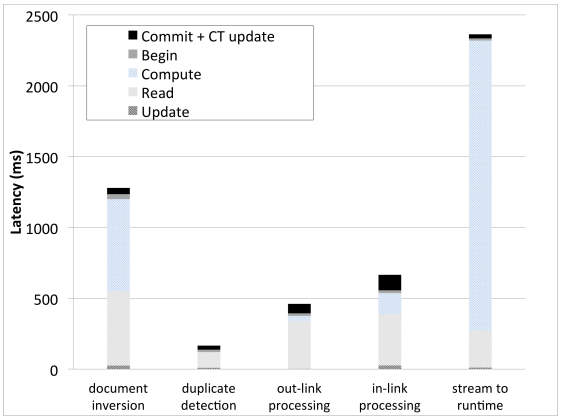


Figure 4: **Transaction latency breakdown in production deployment of Omid in Sieve.** The top two components represent transaction management overhead.

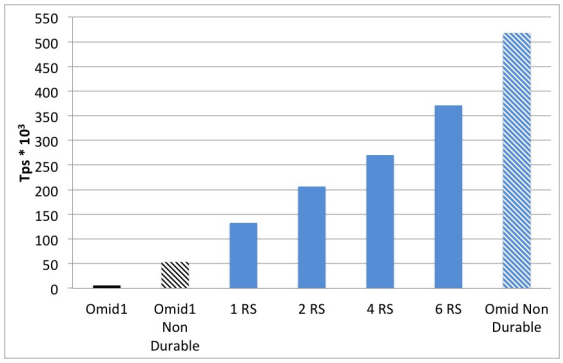


Figure 5: **Scalability of Omid's CT updates with the number of HBase region servers, and comparison with Omid1.** Non-durable versions do not persist transactions and thus provide upper bounds on throughput under perfect storage scaling.

latency is similar to that of a commit operation, minus the time commit takes to update the commit fields.

We see that for tasks that perform significant processing and I/O, like document inversion and streaming to index, Omid's latency overhead (for processing begin and commit) is negligible – 2–6% of the total transaction duration. In very short tasks such as duplicate detection and out-link processing, Omid accounts for up to roughly one third of the transaction latency.

The transaction abort rates observed in Sieve are negligible (around 0.002%). They stem from either transient HBase outages or write-write conflicts, e.g., concurrent in-link updates of extremely popular web pages.

7.2 TM microbenchmarks

We now focus on TM performance. To this end, our microbenchmarks invoke only the TM's begin and commit APIs, and do not access actual data. We run both the TM and HBase (holding the CT) on industry-standard 8-core Intel Xeon E5620 servers with 24GB RAM and 1TB magnetic drive. The interconnects are 1Gbps Ethernet.

We generate workloads in which transaction write-set sizes are distributed Zipf, i.e., follow a power-law ($Pr[X \geq x] = x^{-\alpha}$) with exponent values of $\alpha = 1.2$, $\alpha = 1.6$, and $\alpha = 2$ (the smaller the heavier-tailed), cut-off at 256 keys. Each transaction's latency, (i.e., the time we wait after invoking its begin and before invoking its commit), is set to 5ms per write. Note that read-sets are not sent to the TM and hence their size is immaterial.

We note that key selection affects *real* conflicts: if the written keys are drawn from a heavy-tailed distribution, then two concurrent transactions are likely to update the same key, necessitating one of them to abort. Since this is an artifact of the workload, which is unaffected by our system design, we attempt to minimize this phenomenon in our experiments. We therefore uniformly sample 64-bit integers for the key hashes. Recall that our experience in production shows that real conflicts are indeed rare.

We begin by evaluating scalability, which is our principal design goal. The TM throughput is constrained by two distinct resources – the storage access required for persisting commits in the CT, and the compute resources used for conflict detection. These resources scale independently: the former, evaluated in Section 7.2.1, scales out across multiple HBase region servers, whereas the latter scales up on multi-core hardware, and is studied in Section 7.2.2. Section 7.2.3 then evaluates the throughput-latency tradeoff that Omid exhibits when using a single region server. Finally, in Section 7.2.4, we exercise Omid's high-availability mechanism.

7.2.1 Commit table scalability

Since the commit records are fixed-length (two 64-bit integers), the CT performance does not depend on transaction sizes, and so we experiment only with $\alpha = 1.6$. Recall that in order to optimize throughput, the TM batches writes to the CT and issues multiple batches in parallel. Experimentally, we found that the optimal number of concurrent CT writer threads is 4, and the batch size that yields the best throughput is 2K transactions per writer.

Figure 5 depicts Omid's commit rate as function of the number of HBase region servers, which scales to almost 400K tps. It further compares Omid's throughput to that of Omid1 [25], which, similarly to Omid, runs atop HBase, and uses a centralized TM. It is worth noting that even in the single-server configuration, Omid outperforms Omid1 by more than 25x. This happens because upon each begin request, Omid1 sends to the client a large amount of information (equivalent to the combination of Omid's CT and the in-memory conflict detection

table). This saturates the CPU and network resources.

The "non-durable" bars – leftmost and second from the right – represent experiments where commits are not persisted to stable storage. In Omid this means forgoing the write to the CT, whereas in Omid1 it means disabling the write to BookKeeper in which the system stores its commit log. These results provide upper bounds on the throughput that can be obtained with perfect storage scaling in both systems. Omid peaks at 518K transactions per second, whereas Omid1 peaks at 50K.

7.2.2 Conflict detection scalability

In the experiment reported above, the conflict detection algorithm is evaluated as part of the system. There, the commit table I/O is the bottleneck, and the conflict detection process can keep up with the pace of four I/O threads even when running sequentially, i.e., in a single thread.

We next focus on scale-up of this component running by itself using 1 to 8 threads, in order to study its potential scalability in even larger configurations. The experiment employs a conflict table of 128M 64-bit integers (1G total size). The bucket size is 32 integers, i.e., the table is 4M buckets big.

Figure 6(a) illustrates the processing rate. As expected processing shorter transactions (a bigger α) is faster. The rate scales to 2.6M transactions per second for $\alpha = 1.2$, and to 5M for $\alpha = 2$. Note that exercising such high throughput in a complete system would require an order of magnitude faster network to sustain the request/response packet rate. Clearly the TM's compute aspect is far from being a bottleneck.

Finally, we analyze the false abort rate. (The uniform sampling of key hashes and relatively short transaction latencies render real collisions unlikely, hence all aborts are deemed false). The overall abort rate is negligibly small. In Figure 6(b) we zoom-in on transactions clustered into three buckets: shorter than 8 writes, 8 to 63 writes, and 64+ writes. The worst abort rate is below 0.01%. It occurs, as expected, for long transactions in the most heavy-tailed distribution. Further reduction of the false abort rate would require increasing the table size or using multiple hashes (similarly to Bloom filters).

7.2.3 Latency-throughput tradeoff

We now examine the impact of load on TM access latency with a single region server managing the CT. We use here $\alpha = 1.6$. For every given system load, the batch size is tuned for optimal latency: under light load, no batching is employed, (i.e., commits are written one at a time), whereas under high load, we use batches of 10K.

Figure 7 reports the average client-side latency of commit operations, broken down to three components: (1) network round-trip delay and conflict detection, which are negligible, and do not vary with the load or batch size; (2) HBase CT write latency, which increases with the batch size; and (3) queueing delay at the TM, which increases with the load. Begin latency is similar, and is therefore omitted. We increase the load up to 70K transactions per second, after which the latency becomes excessive; to exceed this throughput, one may use multiple region servers as in the experiment of Section 7.2.1.

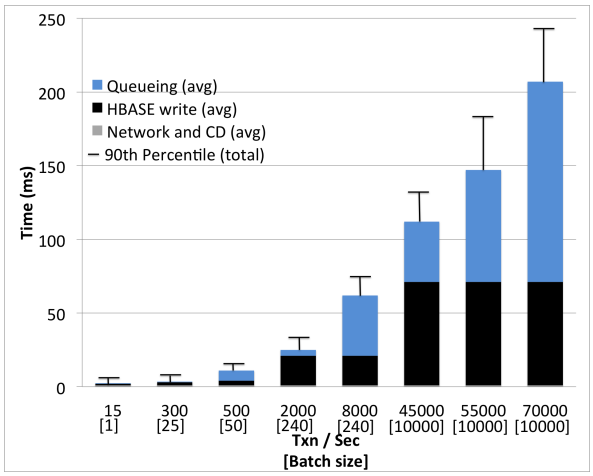


Figure 7: **Omid throughput vs. latency.** Client-perceived commit latency (average broken down and 90% of total); single region server; power-law transaction sizes with $\alpha = 1.6$; batch sizes optimized for minimum latency (in square brackets below each bar).

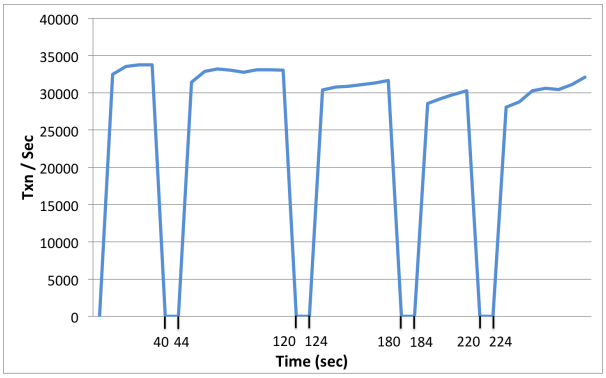
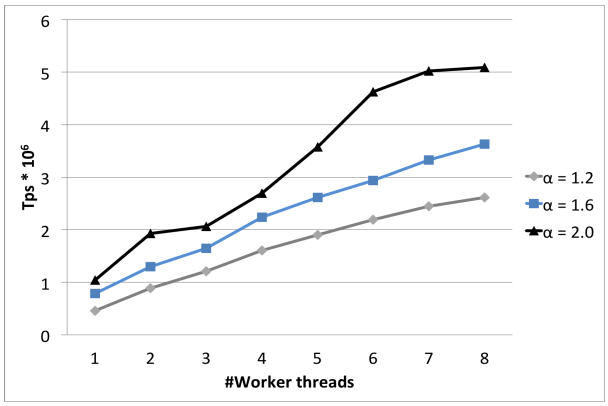


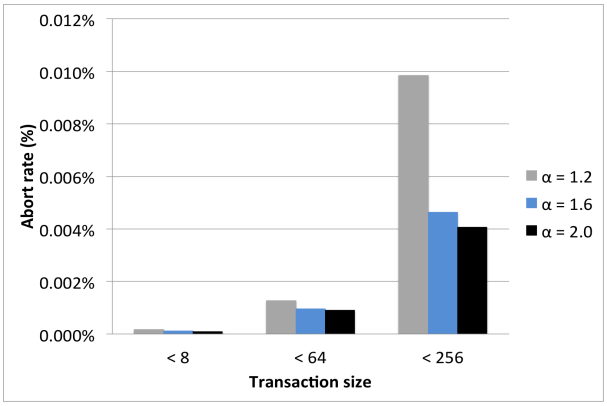
Figure 8: **Omid throughput with four failovers**; recovery takes around 4 seconds.

7.2.4 High availability

Finally, we exercise the high-availability mechanism. As long as the primary TM does not fail, HA induces negligible overhead. We now examine the system's recovery following a primary TM failure. The failure detec-



(a) Conflict detection scalability



(b) Conflict detection false abort rate

Figure 6: **Conflict detection scalability and false abort rate.** Transaction write-set sizes are distributed power-law ($Pr[X \geq x] = x^{-\alpha}$) with exponent values of $\alpha = 1.2$, $\alpha = 1.6$, and $\alpha = 2$ (the smaller the heavier-tailed); the key hashes are 64-bit integers, uniformly sampled to avoid real conflicts whp; transaction latency is 5ms per write.

tion timeout is $\delta = 1$ sec. Figure 8 depicts the system throughput over time, where the primary TM is forcefully shut down after 40 sec, is then allowed to recover, and the new primary (original backup) is shut down after 120 sec. The primary is shut down two more times at 180 and 220 sec; the failover completes within 4 sec.

8 Lessons Learned and Future Work

Omid was originally designed as a foundational building block for Sieve – Yahoo's next-generation content management platform. The need for transactions emerges in scenarios similar to Percolator [31]. Analogously to other data pipelines, Sieve is more throughput-sensitive than latency-sensitive. This has led to a design that trades off latency for throughput via batching. The original design of Omid1 [25] did not employ a CT, but instead had the TM send clients information about all pending transactions. This design was abandoned due to limited scalability in the number of clients, and was replaced by Omid, which uses the CT to track transaction states. The CT may be sharded for I/O scalability, but its update rate is bounded by the resources of the single (albeit multi-threaded) TM; this is mitigated by batching.

Since becoming an Apache Incubator project, Omid is witnessing increased interest, in a variety of use cases. Together with Tephra, it is being considered for use by Apache Phoenix – an emerging OLTP SQL engine over HBase storage [3]. In that context, latency has increased importance. We are therefore developing a low-latency version of Omid that has clients update the CT instead of the TM, which eliminates the need for batching and allows throughput scaling without sacrificing latency. Similar approaches have been used in Percolator [31], Corfu [10], and CockroachDB [5]. We note, however, that such decentralization induces extra synchronization oerhead at commit time and may increase aborts (in particular, reads may induce aborts); the original design may be preferable for throughput-oriented systems.

Another development is using application semantics to reduce conflict detection. Specifically, some applications can identify scenarios where conflicts need not be checked because the use case ensures that they won't happen. Consider, e.g., a massive table load, where records are inserted sequentially, hence no conflicts can arise. Another example is a secondary index update, which is guaranteed to induce no conflict given that the primary table update by the same transaction has been successful. To reduce overhead in such cases, we plan to extend the write API to indicate which written keys need to be tracked for conflict detection.

On the scalability side, faster technologies may be considered to maintain Omid's commit metadata. In particular, since Omid's commit table is usually written sequentially and infrequently read, it might be more efficient to use log-structured storage that is better optimized for the above scenario.

Acknowledgments

We acknowledge the many contributions to Omid, as concept and code, since its early days. Our thanks go to Aran Bergman, Daniel Gomez Ferro, Yonatan Gottesman, Flavio Junqueira, Igor Katkov, Francis Christopher Liu, Ralph Rabbat, Benjamin (Ben) Reed, Kostas Tsioutsiouliklis, Maysam Yabandeh, and the Sieve team at Yahoo. We also thank Dahlia Malkhi and the FAST reviewers for insightful comments.

References

[1] Apache HBase. http://hbase.apache.org.

[2] Apache HBase Coprocessor Introduction. https://blogs.apache.org/hbase/entry/coprocessor_introduction.

[3] Apache Phoenix. https://phoenix.apache.org.

[4] Apache Zookeeper. http://zookeeper.apache.org.

[5] CockroachDB. https://github.com/cockroachdb/cockroach/blob/master/docs/design.md.

[6] Tephra: Transactions for Apache HBase. https://tephra.io.

[7] AGUILERA, M. K., LENERS, J. B., AND WALFISH, M. Yesquel: Scalable sql storage for web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 245–262.

[8] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev. 41*, 6 (Oct. 2007), 159–174.

[9] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.

[10] BALAKRISHNAN, M., MALKHI, D., DAVIS, J. D., PRABHAKARAN, V., WEI, M., AND WOBBER, T. CORFU: A distributed shared log. *ACM Trans. Comput. Syst. 31*, 4 (2013), 10.

[11] BALAKRISHNAN, M., MALKHI, D., WOBBER, T., WU, M., PRABHAKARAN, V., WEI, M., DAVIS, J. D., RAO, S., ZOU, T., AND ZUCK, A. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13, pp. 325–340.

[12] BERENSON, H., BERNSTEIN, P. A., GRAY, J., MELTON, J., O'NEIL, E. J., AND O'NEIL, P. E. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.* (1995), pp. 1–10.

[13] BERNSTEIN, P. A., REID, C. W., AND DAS, S. Hyder - a transactional record manager for shared flash. In *CIDR* (January 2011), pp. 9–20.

[14] CAHILL, M. J., ROHM, U., AND FEKETE, A. Serializable isolation for snapshot databases. In *SIGMOD* (2008), pp. 729–738.

[15] CAMARGOS, L. Sprint: a middleware for high-performance transaction processing. In *In EuroSys 07: Proceedings of the ACM SIGOPS/EuroSys Eu Conference on Computer Systems 2007* (2007), ACM, pp. 385–398.

[16] 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 Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[17] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012), pp. 261–264.

[18] COWLING, J., AND LISKOV, B. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (2012), USENIX ATC'12, pp. 21–21.

[19] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Apr. 2014), pp. 401–414.

[20] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 54–70.

[21] ESCRIVA, R., WONG, B., AND SIRER, E. G. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2012), SIGCOMM '12, pp. 25–36.

[22] ESCRIVA, R., WONG, B., AND SIRER, E. G. Warp: Lightweight multi-key transactions for key-value stores. *CoRR abs/1509.07815* (2015).

[23] EYAL, I., BIRMAN, K., KEIDAR, I., AND VAN RENESSE, R. Ordering transactions with prediction in distributed object stores. In *LADIS* (2013).

[24] FEKETE, A., LIAROKAPIS, D., O'NEIL, E., NEIL, P. O., AND SHASHA, D. Making snapshot isolation serializable. *ACM TODS 30* (2005), 492–528.

[25] FERRO, D. G., JUNQUEIRA, F., KELLY, I., REED, B., AND YABANDEH, M. Omid: Lock-free transactional support for distributed data stores. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014* (2014), pp. 676–687.

[26] GRAY, J., HELLAND, P., O'NEIL, P. E., AND SHASHA, D. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.* (1996), pp. 173–182.

[27] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[28] KRASKA, T., PANG, G., FRANKLIN, M. J., MADDEN, S., AND FEKETE, A. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, pp. 113–126.

[29] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[30] PATTERSON, S., ELMORE, A. J., NAWAB, F., AGRAWAL, D., AND ABBADI, A. E. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. In *PVLDB* (2012), vol. 5, pp. 1459–1470.

[31] PENG, D., AND DABEK, F. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)* (2010).

[32] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD '12, pp. 1–12.

[33] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 87–104.

[34] WELSH, M. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, University of California, Berkeley, 2002.

[35] YABANDEH, M., AND GOMEZ-FERRO, D. A critique of snapshot isolation. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys '12, pp. 155–168.

[36] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), E. L. Miller and S. Hand, Eds., ACM, pp. 263–278.

Application Crash Consistency and Performance with CCFS

Thanumalayan Sankaranarayana Pillai [1] Ramnatthan Alagappan[1] Lanyue Lu[1]
Vijay Chidambaram[2] Andrea C. Arpaci-Dusseau[1] Remzi H. Arpaci-Dusseau[1]

[1]*University of Wisconsin-Madison* [2]*The University of Texas at Austin*

Abstract. Recent research has shown that applications often incorrectly implement crash consistency. We present ccfs, a file system that improves the correctness of application-level crash consistency protocols while maintaining high performance. A key idea in ccfs is the abstraction of a *stream*. Within a stream, updates are committed in program order, thus helping correctness; across streams, there are no ordering restrictions, thus enabling scheduling flexibility and high performance. We empirically demonstrate that applications running atop ccfs achieve high levels of crash consistency. Further, we show that ccfs performance under standard file-system benchmarks is excellent, in the worst case on par with the highest performing modes of Linux ext4, and in some cases notably better. Overall, we demonstrate that both application correctness and high performance can be realized in a modern file system.

1 Introduction

"*Filesystem people should aim to make 'badly written' code 'just work'*" – Linus Torvalds [52]

The constraint of ordering is a common technique applied throughout all levels of computer systems to ease the construction of correct programs. For example, locks and condition variables limit how multi-threaded programs run, making concurrent programming simpler [2]; memory consistency models with stricter constraints (e.g., sequential consistency) generally make reasoning about program behavior easier [47]; `fsync` calls in data management applications ensure preceding I/O operations complete before later operations [5, 35].

Unfortunately, constraining ordering imposes a fundamental cost: poor performance. Adding synchronization primitives to concurrent programs adds overhead and reduces performance [19, 21]; stronger multiprocessor memory models are known to yield lower throughput [16]; forcing writes to a disk or SSD can radically reduce I/O performance [5, 6]. While in rare cases we can achieve both correctness and performance [39], in most cases we must make an unsavory choice to sacrifice one.

Within modern storage systems, this same tension arises. A file system, for example, could commit all updates in order, adding constraints to ease the construction of applications (and their crash-recovery protocols) atop them [3, 35]. Many file system developers have determined that such ordering is performance prohibitive; as a result, most modern file systems reduce internal ordering constraints. For example, many file systems (including ext4, xfs, btrfs, and the 4.4BSD fast file system) re-order application writes [1], and some file systems commit directory operations out of order (e.g., btrfs [35]). Lower levels of the storage stack also re-order aggressively, to reduce seeks and obtain grouping benefits [22,23,41,43].

However, research has shown that user-level applications are often incorrect because of re-ordering [35, 56]. Many applications use a specialized write protocol to maintain crash consistency of their persistent data structures. The protocols, by design or accident, frequently require all writes to commit in their issued order [36].

The main hypothesis in this paper is that a carefully designed and implemented file system can achieve *both* ordering and high performance. We explore this hypothesis in the context of the Crash-Consistent File System (ccfs), a new file system that enables crash-consistent applications while delivering excellent performance.

The key new abstraction provided by ccfs, which enables the goals of high performance and correctness to be simultaneously met, is the *stream*. Each application's file-system updates are logically grouped into a stream; updates within a stream, including file data writes, are guaranteed to commit to disk in order. Streams thus enable an application to ensure that commits are ordered (making recovery simple); separating updates between streams prevents *false write dependencies* and enables the file system to re-order sufficiently for performance.

Underneath this abstraction, ccfs contains numerous mechanisms for high performance. Critically, while ordering updates would seem to overly restrict file-system implementations, we show that the journaling machinery found in many modern systems can be adopted to yield high performance while maintaining order. More specifically, ccfs uses a novel hybrid-granularity journaling approach that separately preserves the order of each stream; hybrid-granularity further enables other needed optimizations, including delta journaling and pointer-less metadata structures. Ccfs takes enough care to retain optimizations in modern file systems (like ext4) that appear at first to be incompatible with strict ordering, with new techniques such as order-preserving delayed allocation.

We show that the ordering maintained by ccfs im-

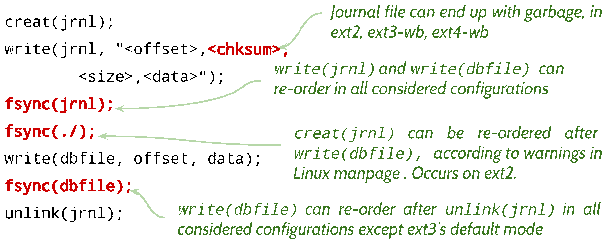


Figure 1: **Journaling Update Protocol.** *Pseudo-code for a simple version of write-ahead journaling; each statement is a system call. The normal text correspond directly to the protocol's logic, while the bold parts are additional measures needed for portability. Italicized comments show which measures are needed under the default modes of ext2, ext3, ext4, xfs, and btrfs, and the writeback mode of ext3/4 (ext3-wb, ext4-wb).*

proves correctness by testing five widely-used applications, including Git and LevelDB (both of which are inconsistent on many modern file systems [35]). We also show that most applications and standard benchmarks perform excellently with only a single stream. Thus, ccfs makes it straightforward to achieve crash consistency efficiently in practice without much developer overhead.

The paper is structured as follows. We provide motivation and background (§2), present ccfs (§3) and evaluate it (§4). We discuss related work (§5) and conclude (§6).

2 Motivation and Background

In this section, we first explain the extent to which current data-intensive applications are vulnerable during a crash. We then describe why a file system that preserves the order of application updates will automatically improve the state of application-level crash consistency. Finally, we discuss the performance overheads of preserving order, and how the overheads can be addressed.

2.1 State of Crash Consistency

To maintain the consistency of their user-level data structures in the event of a crash, many applications [20, 24, 32] modify the data they store in the file system via a carefully implemented *update protocol*. The update protocol is a sequence of system calls (such as file writes and renames) that updates underlying files and directories in a recoverable way. As an example, consider a simple DBMS that stores its user data in a single database file. To maintain transactional atomicity across a system crash, the DBMS can use an update protocol called *journaling* (or *write-ahead logging*): before updating the database file, the DBMS simply records the updates in a separate journal file. The pseudocode for the update protocol is shown in Figure 1. If a crash happens, the DBMS executes a *recovery protocol* when restarted: if the database file was only partially updated, the full update from the journal is replayed.

Correctly implementing crash-consistency protocols has proven to be difficult for a variety of reasons. First, the correctness inherently depends on the exact semantics of the system calls in the update protocol with respect to a system crash. Because file systems buffer writes in memory and send them to disk later, from the perspective of an application the effects of system calls can get re-ordered before they are persisted on disk. For example, in a naive version of the journaling update protocol, the unlink of the journal file can be re-ordered before the update of the database file. In Figure 1, an explicit `fsync` system call is used to force the update to disk, before issuing the `unlink`. Also, the semantics of system calls can differ between file systems; for example, the aforementioned re-ordering occurs in the default configurations of ext2, ext4, xfs, and btrfs, but not in ext3.

Second, the recovery protocol must correctly consider and recover from the multitude of states that are possible when a crash happens during the update protocol. Application developers strive for update protocols to be efficient, since the protocols are invoked during each modification to the data store; more efficient update protocols often result in more possible states to be reasoned about during recovery. For example, the journal protocol in Figure 1 is often extended to batch multiple transactions onto the journal before the actual update to the database file, so as to avoid performance-intensive `fsync` calls.

Finally, crash-consistency protocols are hard to test, much like concurrency mechanisms, because the states that might occur on a crash are non-deterministic. Since efficient protocol implementations are inherently tied to the format used by the application's data structures and concurrency mechanisms, it is impractical to re-use a single, verified implementation across applications.

Unsurprisingly, past research [35, 56, 57] has found many vulnerabilities in the implementations of crash consistency protocols in widely used applications written by experienced developers, such as Google's LevelDB and Linus Torvalds's Git. However, in this paper, we argue that it is practical to construct a file system that automatically improves application crash consistency. We base our arguments on the following hypotheses:

The Ordering Hypothesis: Existing update and recovery protocols (mostly) work correctly on an ordered and weakly-atomic file system (the exact definition of these terms is explained subsequently).

The Efficiency Hypothesis: An ordered and weakly-atomic file system can be as efficient as a file system that does not provide these properties, with the proper design, implementation, and realistic application workloads.

2.2 Weak Atomicity and Order

We hypothesize that most vulnerabilities that exist in application-level update protocols are caused because the related application code depends on two specific file-system guarantees. File systems that provide these guar-

	Time (s)	Seeks	Median seek distance (sectors)
Re-ordered	25.82	23762	120
FIFO	192.56	38201	2002112

Table 1: **Seeks and Order.** *The table shows the number of disk seeks incurred and the total time taken when 25600 writes are issued to random positions within a 2GB file in a HDD. Two different settings are investigated: the writes can be re-ordered or the order of writes is maintained using the FIFO strategy. The number of seeks incurred in each setting and the LBA seek distance shown are determined from a block-level I/O trace.*

antees, therefore, automatically mask application vulnerabilities. The first guarantee, and the major focus of our work, is that the effect of system calls should be persisted on disk in the order they were issued by applications; a system crash should not produce a state where the system calls appear re-ordered. The second (minor) guarantee is that, when an application issues certain types of system calls, the effect of the system call should be atomic across a system crash. The second guarantee, which we term *weak atomicity*, is specifically required for system calls that perform directory operations (including the creation, deletion, and renaming of files and hard links). Weak atomicity also includes stipulations about writes to files, but only at sector granularity (i.e., there is generally no need to guarantee that arbitrarily large writes are atomic). If a system call appends data to the end of a file, both increasing the file size and the writing of data to the newly appended portion of the file should be atomic together.

The fundamental reason that order simplifies the creation of update protocols is that it drastically reduces the number of possible states that can arise in the event of a crash, i.e., the number of states that the recovery protocol has to handle. For example, consider an update protocol that simply overwrites n sectors in a file; if the file system maintains order and weak atomicity, only n crash states are possible, whereas 2^n states are possible if the file system can re-order. Maintaining order makes it easier to reason about the correctness of recovery for both humans and automated tools [35].

The effectiveness of maintaining weak atomicity and order can be understood by considering the application-level crash-consistency vulnerabilities discovered recently [35]. Among 60 vulnerabilities in the study, the authors state that 16 are masked by maintaining weak atomicity alone. They also state that 27 vulnerabilities are masked by guaranteeing order. Of the remaining vulnerabilities, 12 are attributed to durability; however, the authors observe that 8 of these 12 will be masked if the file system guarantees order. Thus, in all, 50 of the 60 vulnerabilities are addressed by maintaining order and weak atomicity; the remaining 10 have minor consequences and are readily masked or fixed [36].

2.3 Order: Bad for Performance

Most real-world deployed file systems (such as btrfs) already maintain the weak atomicity required to mask application-level crash-consistency vulnerabilities. However, all commonly deployed file-system configurations (including ext4 in metadata-journaling mode, btrfs, and xfs) re-order updates, and the re-ordering only seems to increase with each new version of a file system (e.g., ext4 re-orders more than ext3 [35]; newer versions of ext4 re-order even more [53], as do newer systems like btrfs [35]). While maintaining update order is important for application crash consistency, it has traditionally been considered bad for performance, as we now discuss.

At low levels in the storage stack, re-ordering is a fundamental technique that improves performance. To make this case concrete, we created a workload that issues writes to random locations over a disk. Forcing these writes to commit in issue order takes roughly eight times longer than a seek-optimized order (Table 1). Re-ordering is important for hard drives [43] and SSDs [23]; approaches that constrict write ordering are insufficient.

Higher up the stack, ordering can induce negative (and sometimes surprising) performance degradations. Consider the following code sequence:

`write(`f_1`); write(`f_2`); fsync(`f_2`); truncate(`f_1`);`

In this code, without mandated order, the forced writes to f_2 can move ahead of the writes to f_1; by doing so, the truncate obviates the need for any writes to f_1 at all. Similarly, if the user overwrites f_1 instead of truncating it, only the newer data needs to be written to disk.

We call this effect *write avoidance*: not all user-level writes need to be sent to the disk, but can instead be either forgotten due to future truncates or coalesced due to future overwrites. Re-ordering allows write avoidance across `fsync` calls. Global write ordering, in contrast, implies that if writes to f_2 are being forced to disk, so must writes to f_1. Instead of skipping the writes to f_1, the file system must now both write out its contents (and related metadata), and then, just moments later, free said blocks. If the write to f_1 is large, this cost can be high.

We call this situation, where `fsync` calls or cache eviction reduce write avoidance in an ordered file system, a *write dependence*. Write dependence is not limited to writes by a single application; any application that forces writes to disk could cause large amounts of other (potentially unneeded) I/O to occur. When write dependence does not improve crash consistency, as when it occurs between independent applications, we term it a *false dependence*, an expected high-cost of global order.

Apart from removing the chance for write avoidance, write dependence also worsens application performance in surprising ways. For example, the `fsync(`f_2`)` becomes a high-latency operation, as it must wait for all previous writes to commit, not just the writes to f_2. The overheads associated with write dependence can be further exacerbated by various optimizations found in modern file systems. For example, the ext4 file system uses a tech-

nique known as *delayed allocation*, wherein it batches together multiple file writes and then subsequently allocates blocks to files. This important optimization is defeated by forced write ordering.

2.4 Order with Good Performance

We believe it is possible to address the overheads associated with maintaining order in practice. To reduce disk-level scheduling overheads, a variety of techniques have been developed that preserve the *appearance* of ordered updates in the event of a crash while forcing few constraints on disk scheduling.

For example, in ext4 data journaling, all file-system updates (metadata and data) are first written to a journal. Once committed there, the writes can be propagated (checkpointed) to their in-place final locations. Note that there are no ordering constraints placed upon the checkpoint writes; they can be re-ordered as necessary by lower layers in the storage stack to realize the benefits of low-level I/O scheduling. Further, by grouping all writes into a single, large transaction, writes are effectively committed in program order: if a write to f_1 occurs before a write to f_2, they will either be committed together (in the same transaction), or the write to f_2 will commit later; never will f_2 commit before f_1. We discuss ext4 journaling in more detail in the next section.

Unfortunately, total write ordering, as provided with data journaling, exacts a high performance cost: each data item must be written twice, thus halving disk bandwidth for some workloads. For this reason, most journaling file systems only journal metadata, maintaining file-system crash consistency but losing ordering among application writes. What would be ideal is the performance of metadata-only journaling combined with the ordering guarantees provided by full data journaling.

However, even if an efficient journaling mechanism is used, it does not avoid overheads due to false dependence. To address this problem, we believe a new abstraction is needed, which enables the file system to separate update orderings across different applications. Within an application, we believe that false dependence is rare and does not typically arise.

Thus, we are left with two open questions. Can a metadata-only journaling approach be adopted that maintains order but with high performance? Second, can a new abstraction eliminate false dependence? We answer these questions in the affirmative with the design of ccfs.

3 Crash-Consistent File System

In this section, we describe ccfs, a file system that embraces application-level crash consistency. Ccfs has two goals: preserving the program order of updates and weak atomicity, and performance similar to widely-used reordering file systems. So as to satisfy these goals, we derive ccfs from the ext4 file system. Ext4 is widely used, includes many optimizations that allow it to perform efficiently in real deployments, and includes a journaling mechanism for internal file-system consistency. In ccfs, we extend ext4's journaling to preserve the required order and atomicity in an efficient manner without affecting the optimizations already present in ext4.

The key idea in ccfs is to separate each application into a *stream*, and maintain order only within each stream; writes from different streams are re-ordered for performance. This idea has two challenges: metadata structures and the journaling mechanism need to be separated between streams, and order needs to be maintained within each stream efficiently. Ccfs should solve both without affecting existing file-system optimizations. In this section, we first explain ext4's journaling mechanism (§3.1), then the streams abstraction (§3.2), how streams are separated (§3.3) and how order is maintained within a stream (§3.4), and our implementation (§3.5). We finally discuss how applications can practically start using the streams abstraction (§3.6).

3.1 Journaling in Ext4

To maintain internal file-system metadata consistency, ext4 requires the atomicity of sets of metadata updates (e.g., all metadata updates involved in creating a file) and an order between these sets of updates. Ext4 uses an optimized journaling technique for this purpose. Specifically, the journaling occurs at block granularity, batches multiple sets of atomic metadata updates (*delayed logging* [11]), uses a circular journal, and delays forced checkpointing until necessary. The block-granularity and circular aspects prove to be a challenge for adoption in ccfs, while delayed logging and checkpointing are important optimizations that ccfs needs to retain. We now briefly explain these techniques of ext4 journaling.

Assume the user performs a metadata operation (such as creating a file), causing ext4 to modify metadata structures in the file-system blocks b_1, b_2, b_3. Ext4 associates b_1, b_2, b_3 with an in-memory data structure called the *running transaction*, T_i. Instead of immediately persisting T_i when the metadata operation completes, ext4 waits for the user to perform more operations; when this happens, the resulting set of block modifications are also associated with T_i (i.e., *delayed logging*). Periodically, ext4 *commits* the running transaction, i.e., writes the updated contents of all the associated blocks of T_i and some bookkeeping information to an on-disk journal. When T_i starts committing, a new running transaction (T_{i+1}) is created to deal with future metadata operations. Thus, ext4 always has one running transaction, and at most one committing transaction. Once T_i finishes committing, its blocks can be written to their actual locations on disk in any order; this is usually done by Linux's page-flushing

daemon in an optimized manner.

If a crash happens, after rebooting, ext4 scans each transaction written in the journal file sequentially. If a transaction is fully written, the blocks recorded in that transaction are propagated to their actual locations on disk; if not, ext4 stops scanning the journal. Thus, the atomicity of all block updates within each transaction are maintained. Maintaining atomicity implicitly also maintains order within a transaction, while the sequential scan of the journal maintains order across transactions.

The on-disk journal file is circular: after the file reaches a maximum size, committed transactions in the tail of the journal are freed (i.e., *checkpointed*) and that space is reused for recording future transactions. Ext4 ensures that before a transaction's space is reused, the blocks contained in it are first propagated to their actual locations (if the page-flushing mechanism had not yet propagated them). Ext4 employs techniques that coalesce such writebacks. For example, consider that a block recorded in T_i is modified again in T_j; instead of writing back the version of the block recorded in T_i and T_j separately, ext4 simply ensures that T_j is committed before T_i's space is reused. Since the more recent version (in T_j) of the block will be recovered on a crash without violating atomicity, the earlier version of the block will not matter. Similar optimizations also handle situations where committed blocks are later unreferenced, such as when a directory gets truncated.

For circular journaling to work correctly, ext4 requires a few invariants. One invariant is of specific interest in ccfs: the number of blocks that can be associated with a transaction is limited by a threshold. To enforce the limit, before modifying each atomic set of metadata structures, ext4 first verifies that the current running transaction (say, T_i) has sufficient capacity left; if not, ext4 starts committing T_i and uses T_{i+1} for the modifications.

3.2 Streams

Ccfs introduces a new abstraction called the *stream*; each application usually corresponds to a single stream. Writes from different streams are re-ordered for performance, while order is preserved within streams for crash consistency. We define the stream abstraction such that it can be easily used in common workflows; as an example, consider a text file $f1$ that is modified by a text editor while a binary file $f2$ is downloaded from the network, and they are both later added to a VCS repository. Initially, the text editor and the downloader must be able to operate on their own streams (say, A and B, respectively), associating $f1$ with A and $f2$ with B. Note that there can be no constraints on the location of $f1$ and $f2$: the user might place them on the same directory. Moreover, the VCS should then be able to operate on another stream C, using C for modifying both $f1$ and $f2$. In such a scenario, the stream abstraction should guarantee the order required for crash consistency, while allowing enough re-ordering for the best performance possible.

Hence, in ccfs, streams are transient and are not uniquely associated with specific files or directories: a file that is modified in one stream might be later modified in another stream. However, because of such flexibility, while each stream can be committed independently without being affected by other streams, it is convenient if the stream abstraction takes special care when two streams perform operations that affect logically related data. For example, consider a directory that is created by stream A, and a file that is created within the directory by stream B; allowing the file creation to be re-ordered after the directory creation (and recovering the file in a *lost+found* directory on a crash) might not make logical sense from an application's perspective. Hence, when multiple streams perform logically related operations, the file system takes sufficient care so that the temporal order between those operations is maintained on a crash.

We loosely define the term *related* such that related operations do not commonly occur in separate streams within a short period of time; if they do, the file system might perform inefficiently. For example, separate directory entries in a directory are not considered related (since it is usual for two applications to create files in the same directory), but the creation of a file is considered related to the creation of its parent. Section 3.5 further describes which operations are considered logically related and how their temporal order is maintained.

Our stream interface allows all processes and threads belonging to an application to easily share a single stream, but also allows a single thread to switch between different streams if necessary. Specifically, we provide a `setstream(s)` call that creates (if not already existing) and associates the current thread with the stream `s`. All future updates in that thread will be assigned to stream `s`; when forking (a process or thread), a child will adopt the stream of its parent. The API is further explained in Section 3.5 and its usage is discussed in Section 3.6.

3.3 Separating Multiple Streams

In ccfs, the basic idea used to separately preserve the order of each stream is simple: ccfs extends the journaling technique to maintain multiple in-memory running transactions, one corresponding to each stream. Whenever a synchronization system call (such as `fsync`) is issued, only the corresponding stream's running transaction is committed. All modifications in a particular stream are associated with that stream's running transaction, thus maintaining order within the stream (optimizations regarding this are discussed in the next section).

Using multiple running transactions poses a challenge: committing one transaction without committing others

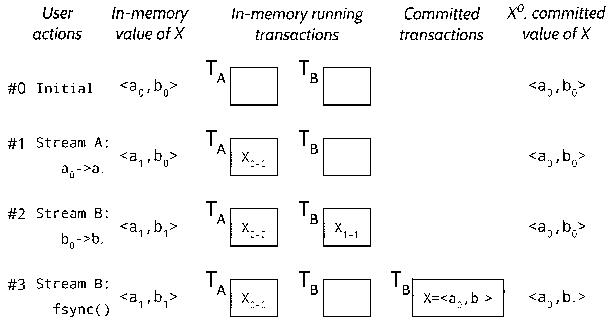


Figure 2: **Hybrid-granularity Journaling.** *Timeline showing hybrid-granularity journaling in ccfs. Block X initially contains the value $\langle a_0, b_0 \rangle$, T_A and T_B are the running transactions of streams A and B; when B commits, X is recorded at the block level on disk .*

(i.e., re-ordering between streams) inherently re-orders the metadata modified across streams. However, internal file-system consistency relies on maintaining a global order between metadata operations; indeed, this is the original purpose of ext4's journaling mechanism. It is hence important that metadata modifications in different streams be logically independent and be separately associated with their running transactions. We now describe the various techniques that ccfs uses to address this challenge while retaining the existing optimizations in ext4.

3.3.1 Hybrid-granularity Journaling

The journaling mechanism described previously (§3.1) works at block-granularity: entire blocks are associated with running transactions, and committing a transaction records the modified contents of entire blocks. Ccfs uses *hybrid-granularity journaling*, where byte-ranges (instead of entire blocks) are associated with the running transaction, but transactional commits and checkpointing still happen at block-granularity.

Ccfs requires byte-granularity journaling because separate metadata structures modified by different streams might exist in the same file-system block. For example, a single block can contain the inode structure for two files used by different applications; in block-granularity journaling, it is not possible to associate the inodes with the separate running transactions of two different streams.

Block-granularity journaling allows many optimizations that are not easily retained in byte-granularity. A major optimization affected in ext4 is data coalescing during checkpoints: even if multiple versions of a block are committed, only the final version is sent to its in-place location. Since the Linux buffer cache and storage devices manage data at block granularity, such coalescing becomes complicated with a byte-granularity journal.

To understand hybrid-granularity journaling, consider the example illustrated in Figure 2. In this example, block X initially contains the bytes $\langle a_0 b_0 \rangle$. Before allowing any writes, ccfs makes an in-memory copy (say, X^0) of the initial version of the block. Let the first byte of X be modified by stream A into a_1; ccfs will associate the byte range X_{0-0} with the running transaction T_A of stream A (X_{i-j} denotes the ith to jth bytes of block X), thus following byte-granularity. Let stream B then modify the second byte into b_1, associating X_{1-1} with T_B; the final in-memory state of X will be $\langle a_1 b_1 \rangle$. Now, assume the user calls `fsync` in stream B, causing T_B to commit (T_A is still running). Ccfs converts T_B into block-granularity for the commit, by super-imposing the contents of T_B (i.e., X_{1-1} with the content b_1) on the initial versions of their blocks (i.e, X^0 with content $\langle a_0 b_0 \rangle$), and committing the result (i.e., $\langle a_0 b_1 \rangle$). When T_B starts committing, it updates X^0 with the value of X that it is committing. If the user then calls `fsync` in A, X_{0-0} is super-imposed on X^0 ($\langle a_0 b_1 \rangle$), committing $\langle a_1 b_1 \rangle$.

Thus, hybrid-granularity journaling performs in-memory logging at byte-granularity, allowing streams to be separated; the delayed-logging optimization of ext4 is unaffected. Commits and checkpoints are block-granular, thus preserving delayed checkpointing.

3.3.2 Delta Journaling

In addition to simply associating byte ranges with running transactions, ccfs allows associating the exact changes performed on a specific byte range (i.e., the *deltas*). This technique, which we call *delta journaling*, is required when metadata structures are actually shared between different streams (as opposed to independent structures sharing the same block). For example, consider a metadata tracking the total free space in the file system: all streams need to update this metadata.

Delta journaling in ccfs works as follows. Assume that the byte range X_{1-2} is a shared metadata field storing an integer, and that stream A adds i to the field and stream B subtracts j from the field. Ccfs associates the delta $\langle X_{1-2} \colon +i \rangle$ to the running transaction T_A and the delta $\langle X_{1-2} \colon -j \rangle$ to the running T_B. When a transaction commits, the deltas in the committing transaction are imposed on the initial values of their corresponding byte ranges, and then the results are used for performing the commit. In our example, if X_{1-2} initially had the value k, and stream B committed, the value $(k - j)$ will be recorded for the byte range during the commit; note that hybrid-granularity journaling is still employed, i.e.., the commit will happen at block-granularity.

In ext4, shared metadata structures requiring delta journaling are the *free inode count* and the *free block count*, which concern the global state across the file system. Delta journaling is also needed for the *nlink* and the modification time fields of directory inodes, since multiple streams can modify the same directory.

3.3.3 Pointer-less Data Structures

Metadata in file systems often use data structures such as linked lists and trees that contain internal pointers, and these cause metadata operations in one stream to update

pointers in structures already associated with another stream. For example, deleting an entry in a linked list will require updating the *next* pointer of the previous entry, which might be associated with another stream. Ccfs eliminates the need to update pointers across streams by adopting alternative data structures for such metadata.

Ext4 has two metadata structures that are of concern: directories and the *orphan list*. Directories in ext4 have a structure similar to linked lists, where each entry contains the relative byte-offset for the next entry. Usually, the relative offset recorded in a directory entry is simply the size of the entry. However, to delete a directory entry d_i, ext4 adds the size of d_i to the offset in the previous entry (d_{i-1}), thus making the previous entry point to the next entry (d_{i+1}) in the list. To make directories pointer-less, ccfs replaces the offset in each entry with a *deleted* bit: deleting an entry sets the bit. The insert and scan procedures are modified appropriately; for example, the insert procedure recognizes previously deleted entries in the directory and uses them for new entries if possible.

The orphan list in ext4 is a standard linked list containing recently freed inodes and is used for garbage collecting free blocks. The order of entries in the list does not matter for its purposes in ext4. We convert the orphan list into a pointer-less structure by substituting it with an orphan directory, thus reusing the same data structure.

3.3.4 Order-less Space Reuse

Ccfs carefully manages the allocation of space in the file system such that re-ordering deallocations between streams does not affect file-system consistency. For example, assume stream A deletes a file and frees its inode, and stream B tries to create a file. The allocation routines in ext4 might allocate to B the inode that was just freed by A. However, if B commits before A, and then a crash occurs, the recovered state of the file system will contain two unrelated files assigned the same inode.

Ext4 already handles the situation for block allocation (for reasons of security) by reusing blocks only after the transaction that frees those blocks has fully committed. In ccfs, we extend this solution to both inode and directory-entry reuse. Thus, in our example, B will reuse A's freed inode only if A has already been committed.

3.4 Maintaining Order Within Streams

We saw in the previous section how to separate dependencies across independent streams; we now focus on ordering the updates within the same stream. Ext4 uses metadata-only journaling: ext4 can re-order file appends and overwrites. Data journaling, i.e., journaling all updates, preserves application order for both metadata and file data, but significantly reduces performance because it often writes data twice. A hybrid approach, selective data journaling (SDJ) [5], preserves order of both data and metadata by journaling only overwritten file data; it only journals the block pointers for file appends. Since modern workloads are mostly composed of appends, SDJ is significantly more efficient than journaling all updates.

System calls	No delayed allocation	Order-violating delayed alloc	Order-preserving delayed alloc
write(f1, 1);	alloc(f1, 1);		
write(f2, 1);	alloc(f2, 1);		
write(f1, 1);	alloc(f1, 1);		
write(f2, 1);	alloc(f2, 1);		
fsync(f2);		alloc(f2, 2);	atomic{alloc(f1, 2) alloc(f2, 2)};

Figure 3: **Order-preserving Delayed Allocation.** *Timeline of allocations performed, corresponding to a system-call sequence.*

We adopt the hybrid SDJ approach in ccfs. However, the approach still incurs noticeable overhead compared to ext4's default journaling under practical workloads because it disables a significant optimization, *delayed allocation*. In our experiments, the createfiles benchmark results in 8795 ops/s on ext4 with delayed allocation on a HDD, and 7730 ops/s without (12% overhead).

Without delayed allocation, whenever an application appends to files, data blocks are allocated and block pointers are assigned to the files immediately, as shown in the second column of Figure 3. With delayed allocation (third column), the file system does not immediately allocate blocks; instead, allocations for multiple appends are delayed and done together. For order to be maintained within a stream, block pointers need to be assigned immediately (for example, with SDJ, only the order of allocations is preserved across system crashes): naive delayed allocation inherently violates order.

Ccfs uses a technique that we call *order-preserving delayed allocation* to maintain program order while allowing delayed allocations. Whenever a transaction T_i is about to commit, all allocations (in the current stream) that have been delayed so far are performed and added to T_i before the commit; further allocations from future appends by the application are assigned to T_{i+1}. Thus, allocations are delayed until the next transaction commit, but not across commits. Since order is maintained within T_i via the atomicity of all operations in T_i, the exact sequence in which updates are added to T_i does not matter, and thus the program order of allocations is preserved.

However, the running transaction's size threshold poses a challenge: at commit time, what if we cannot add all batched allocations to T_i? Ccfs solves this challenge by reserving the space required for allocations when the application issues the appends. Order-preserving delayed allocation thus helps ccfs achieve ext4's performance while maintaining order. For the createfiles benchmark, the technique achieves 8717 ops/s in ccfs, and thus performs similar to the default configuration of ext4 (8795 ops/s).

3.5 Implementation

Ccfs changes 4,500 lines of source code (ext4 total: 50,000 lines). We now describe our implementation.

Stream API. The `setstream()` call takes a *flags* parameter along with the stream. One flag is currently supported: `IGNORE_FSYNC` (ignore any `fsync` calls in this stream). We provide a `getstream()` call that is used, for example, to find if the current process is operating on the *init* stream (explained in §3.6) or a more specific stream. A `streamsync()` call flushes all updates in the current stream.

Related Operations Across Streams. The current version of ccfs considers the following operations as logically related: modifying the same regular file, explicitly modifying the same inode attributes (such as the owner attribute), updating (creating, deleting, or modifying) directory entries of the same name within a directory, and creating a directory and any directory entries within that directory. To understand how ccfs maintains temporal ordering between related operations from different streams, consider that stream A first performs operation O_A at time t_1 and stream B then performs a related operation O_B at t_2. If stream A gets committed between t_1 and t_2 (either due to an `fsync` or a periodic background flush), the required temporal order is already maintained, since O_A is already on disk before O_B is performed. If not, ccfs temporarily merges the streams together and treats them as one, until the merged streams get committed to disk; the streams are then separated and allowed to proceed independently.

Maintaining Order Within Streams. An implementation challenge for order-preserving delayed allocation is that the allocations need to be performed when a transaction is about to commit, but before the actual committing starts. We satisfy these requirements without much complexity by performing the allocations in the `T_LOCKED` state of the transaction, a transient state in the beginning of every commit when all file-system updates are blocked. A more efficient implementation can carefully perform these allocations before the `T_LOCKED` state.

To correctly maintain the order of file updates, SDJ requires careful handling when data is both appended and overwritten on the same block. For example, consider an append when T_i was running and an overwrite when T_i is committing (when T_{i+1} is running); to maintain order, two versions of the block must be created in memory: the old version (that does not contain the overwrite) must be used as part of T_i's commit, and the new version must be journaled in T_{i+1}. Ccfs handles these cases correctly.

3.6 Discussion

We now discuss how we expect applications to use streams. Overall, the abstraction is flexible: while we expect most applications to use a single stream, if needed, applications can also use separate streams for individual tasks, or multiple applications can share a single stream. In the current version of ccfs, the *init* process is assigned an *init* stream; hence, all applications inherit this stream by default. We expect most applications whose write performance are user visible to issue a single `setstream()` call in the beginning of the application (but to not make any other code changes). Thus, applications by default will have improved crash consistency, and applications issuing `setstream()` will have both improved consistency and high performance. If so desired, applications can also significantly improve their performance (while maintaining consistency) by first setting the `IGNORE_FSYNC` flag and removing any `O_SYNC` flags, and issuing `streamsync()` calls only when durability is actually desired.

Application	ext4	ccfs
LevelDB	1	0
SQLite-Roll	0	0
Git	2	0
Mercurial	5	2
ZooKeeper	1	0

(a) Vulnerabilities found

Application		ext4	ccfs
LevelDB	Images	158 / 465	427 / 427
	Time (s)	24.31 / 30	30 / 30
Git	Images	84 / 112	96 / 96
	Time (s)	9.95 / 40	40 / 40

(b) Consistent post-reboot disk states produced by BoB

Table 2: **Consistency Testing.** *The first table shows the results of model-based testing using Alice, and the second shows experimental testing with BoB. Each vulnerability reported in the first table is a location in the application source code that has to be fixed. The* Images *rows of the second table show the number of disk images reproduced by the BoB tool that the application correctly recovers from; the* Time *rows show the time window during which the application can recover correctly from a crash (x / y: x time window, y total workload runtime). For Git, we consider the default configuration instead of a safer configuration with bad performance (§4.4).*

4 Evaluation

In our evaluation, we answer the following questions:

- Does ccfs improve application crash consistency?
- Does ccfs effectively use streams to eliminate the overhead of write dependencies?
- How does ccfs perform in standard file system benchmarks run in a single stream?
- What is the performance effect of maintaining order on real applications?

We performed a set of experiments to answer these questions. For the experiments, we use an Intel Core 2 Quad Processor Q9300 with 4 GB of memory running Linux 3.13, with either an SSD (Samsung 840 EVO 500 GB) or a HDD (Toshiba MK1665GSX 160 GB).

4.1 Reliability

We first examine whether the in-order semantics provided by ccfs improves application crash consistency compared to the widely-used ext4 file system (which reorders writes). We follow a model-based testing strategy to check application consistency on both file systems using the Alice tool [35]. The tool records the system-call trace for a given application workload, and then uses a file-system model to reproduce the possible set of file-system states if a system crash occurs. We con-

figured Alice with the models of ext4 (model provided with the tool) and ccfs (system calls are weakly atomic and in-order). We tested five applications previously reported [35] to exhibit crash inconsistencies on ext4: LevelDB, SQLite, Git, Mercurial, and ZooKeeper. We use workloads similar to the previous study, but newer versions of the applications; we do not check durability in Git and Mercurial since they never call `fsync`.

The results of our testing are shown in Table 2(a). Ext4 results in multiple inconsistencies: LevelDB fails to maintain the order in which key-value pairs are inserted, Git and Mercurial can result in repository corruption, and ZooKeeper may become unavailable. With ccfs, the only inconsistencies were with Mercurial. These inconsistencies are exposed on a process crash with any file system, and therefore also occur during system crashes in ccfs; they result only in *dirstate corruption*, which can be manually recovered from and is considered to be of minor consequence [27]. Thus, our model-based testing reveals that applications are significantly more crash consistent on ccfs than ext4.

We used the BoB tool [35] to test whether our implementation of ccfs maintains weak atomicity and ordering, i.e., whether the implementation reflects the model used in the previous testing. BoB records the block-level trace for an application workload running on a file system, and reproduces a subset of disk images possible if a crash occurs. BoB generates disk images by persisting blocks in and out of order; each image corresponds to a time window during the runtime where a crash will result in the image. These windows are used to measure how much time the application remains consistent.

We used Git and LevelDB to test our implementation and compare it with ext4; both have crash vulnerabilities exposed easily on a re-ordering file system. Table 2(b) shows our results. With ext4, both applications can easily result in inconsistency. LevelDB on ext4 is consistent only on 158 of the 465 images reproduced; a system crash can result in being unable to open the datastore after reboot, or violate the order in which users inserted key-value pairs. Git will not recover properly on ext4 if a crash happens during 30.05 seconds of the 40 second runtime of the workload. With ccfs, we were unable to reproduce any disk state in which LevelDB or Git are inconsistent. We conclude that our implementation provides the desired properties for application consistency.

Thus, our results show that ccfs noticeably improves the state of application crash consistency. We next evaluate whether this is achieved with good performance.

4.2 Multi-stream Benefits

Maintaining order causes write dependence during `fsync` calls and imposes additional overheads, since each `fsync` call must flush all previous dirty data. In the simplest case, this results in additional `fsync` latency; it can also prevent writes from being coalesced across `fsync` calls when data is overwritten, and prevent writes from being entirely avoided when the previously written data is deleted. We now evaluate if using separate streams in ccfs prevents these overheads.

Micro-Benchmark	File system	`fsync` latency (s)	`fsync` written (MB)	Total written (MB)
Append	ext4	0.08	0.03	100.19
	ccfs-1	1.28	100.04	100.18
	ccfs-2	0.08	0.03	100.20
Truncate	ext4	0.07	0.03	0.18
	ccfs-1	1.28	100.04	100.21
	ccfs-2	0.05	0.03	0.20
Overwrite	ext4	0.08	0.03	100.19
	ccfs-1	1.27	100.04	300.72
	ccfs-2	0.07	0.03	100.20

Table 3: **Single-`fsync` Experiments.** *`fsync` latencies in the first column correspond to the data written by the `fsync` shown in the second column on HDD, while the total data shown in the third column affects the available device bandwidth and hence performance in more realistic workloads.*

We devised three microbenchmarks to study the performance effects of preserving order. The *append* microbenchmark appends a large amount of data to file A, then writes 1 byte to file B and calls `fsync` on B; this stresses the `fsync` call's latency. The *truncate* benchmark truncates file A after calling `fsync` while *overwrite* overwrites A after the `fsync`; these benchmarks stress whether or not writes are avoided or coalesced.

We use two versions of each benchmark. In the simpler version, we write 100 MB of data in file A and measure the latency of the `fsync` call and the total data sent to the device. In another version, a foreground thread repeatedly writes B and calls `fsync` every five seconds; a background thread continuously writes to A at 20 MB/s, and may truncate A or overwrite A every 100 MB, depending on the benchmark. The purpose of the multi-`fsync` version is to understand the distribution of `fsync` latencies observed in such a workload.

We ran the benchmarks on three file-system configurations: ext4, which re-orders writes and does not incur additional overheads, ccfs using a single stream (ccfs-1), and ccfs with modifications of A and B in separate streams (ccfs-2). Table 3 and Figure 4 show our results.

For the append benchmark, in ext4, the `fsync` completes quickly in 0.08 seconds since it flushes only B's data to the device. In ccfs-1, the `fsync` sends 100 MB and takes 1.28 seconds, but ccfs-2 behaves like ext4 since A and B are modified in different streams. Repeated `fsync` follows the same trend: most `fsync` calls are fast in ext4 and ccfs-2 but often take more than a second in ccfs-1. A few `fsync` calls in ext4 and ccfs-2 are slow due to interference from background activity by the page-flushing daemon and the periodic journal commit.

With truncates, ext4 and ccfs-2 never send file A's data to disk, but ccfs-1 sends the 100 MB during `fsync`, re-

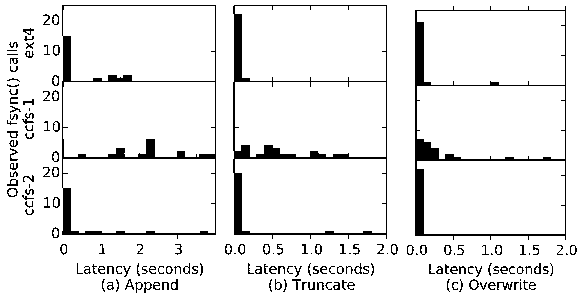


Figure 4: **Repeated `fsync` Experiments** *Histogram of user-observed foreground latencies in our multi-`fsync` experiments. Each experiment is run for two minutes on a HDD.*

sulting in higher latency and more disk writes. Most repeated `fsync` calls in ext4 and ccfs-2 are fast, as expected; they are slow in ccfs-1, but still quicker than the append benchmark because the background thread would have just truncated A before some of the `fsync`.

With overwrites, in both ext4 and ccfs-2, only the final version of A's data reaches the disk: in ccfs-2, SDJ considers the second modification of A an append because the first version of A is not yet on disk (this still maintains order). In ccfs-1, the first version is written during the `fsync`, and then the second version (overwrite) is both written to the journal and propagated to its actual location, resulting in 300 MB of total disk writes. Repeated `fsync` calls are slow in ccfs-1 but quicker than previous benchmarks because of fewer disk seeks: with this version of the benchmark, since A is constantly overwritten, data is only sent to the journal in ccfs-1 and is never propagated to its actual location.

These results show that ccfs is effective at avoiding write dependence overheads when multiple streams are used (in comparison to a file system providing global order). The results also show that, within a stream, write dependence can cause noticeable overhead. For certain applications, therefore, it is possible that dividing the application into multiple streams is necessary for performance. The subsequent sections show that the majority of the applications do not require such division.

4.3 Single-stream Overheads

The previous experiments show how ccfs avoids the performance overheads across streams; we now focus on performance within a stream. The performance effects of maintaining order within a stream are affected by false dependencies between updates within the stream, and hence depend significantly on the pattern of writes. We perform our evaluation using the Filebench [12, 51] suite that reflects real-world workload patterns and microbenchmarks, and compare performance between ext4 (false dependencies are not exposed) and ccfs (false dependencies are exposed because of ordering within streams). Another source of overhead within streams is the disk-level mechanism used to maintain order, i.e., the SDJ technique used in ccfs. Hence, we compare performance between ext4 (no order), ccfs (order-preserving delayed allocation and SDJ), and ext4 in the data=journal mode (*ext4-dj*, full data journaling). We compare performance both with a HDD (disk-level overheads dominated by seeks) and an SSD (seeks less pronounced).

The overall results are shown in Figure 5; performance is most impacted by overwrites and `fsync` calls. We now explain the results obtained on each benchmark.

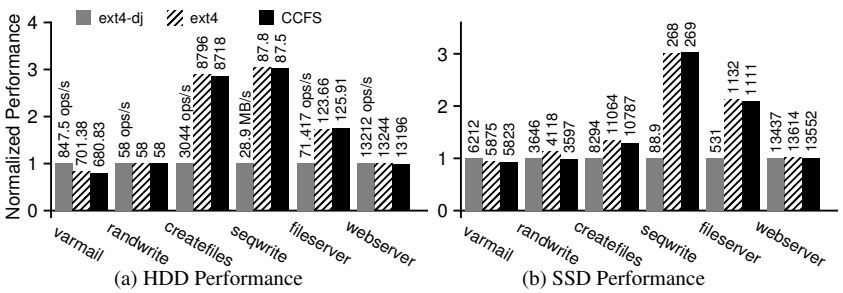
The *varmail* benchmark emulates a multithreaded mail server, performing file creates, appends, deletes, reads, and `fsync` calls in a single directory. Since each append is immediately followed by an `fsync`, there is no additional write dependence due to ordering. Performance is dominated by seek latency induced by the frequent `fsync` calls, resulting in similar performance across ext4 and ccfs. Ext4-dj issues more writes but incurs less seeks (since data is written to the journal rather than the in-place location during each `fsync`), and performs 20% better in the HDD and 5% better in the SSD.

Randwrite overwrites random locations in an existing file and calls `fsync` every 100 writes. Since the `fsync` calls always flush the entire file, there is no additional write dependence due to ordering. However, the overwrites cause both ccfs (SDJ) and ext4-dj (full journaling) to write twice as much data as ext4. In the HDD, all file systems perform similarly since seeks dominate performance; in the SSD, additional writes cause a 12% performance decrease for ccfs and ext4-dj.

Createfiles and *seqwrite* keep appending to files, while *fileserver* issues appends and deletes to multiple files; they do not perform any overwrites or issue any `fsync` calls. Since only appends are involved, ccfs writes the same amount of data as ext4. Under the HDD, similar performance is observed in ccfs and in ext4. Under SSDs, createfiles is 4% slower atop ccfs because of delayed allocation in the `T_LOCKED` state, which takes a noticeable amount of time (an average of 132 ms during each commit); this is an implementation artifact, and can be optimized. For all these benchmarks, ext4-dj writes data twice, and hence is significantly slower. *Webserver* involves mostly reads and a few appends; performance is dominated by reads, all file systems perform similarly.

Figure 5(c) compares the CPU usage of ccfs and ext4. For most workloads, our current implementation of ccfs has moderately higher CPU usage; the significant usage for *fileserver* and *seqwrite* is because the workloads are dominated by block allocations and de-allocations, which is especially CPU intensive for our implementation. This can be improved by adopting more optimized structures and lookup tables (§3.5). Thus, while it does not noticeably impact performance in our experiments, reducing CPU usage is an important future goal for ccfs.

Overall, our results show that maintaining order does



(a) HDD Performance (b) SSD Performance

	Writes (KB/op)			CPU ($\mu s/op$)	
	ext4-dj	**ext4**	**ccfs**	**ext4**	**ccfs**
varmail	3.42	2.91	2.98	59.9	67.2
randwrite	8.1	4.1	8.1	16.8	24.4
createfiles	12.22	5.53	5.49	89.1	94.4
seqwrite	2.0	1.0	1.0	0.9	2.4
fileserver	1093	321	327	1040	2937
webserver	0.49	0.24	0.15	74.4	75.5

(c) HDD Writes and CPU usage

Figure 5: **Imposing Order at Disk-Level: Performance, Data Written, and CPU usage.** *(a) and (b) show throughput under standard benchmarks for ccfs, ext4, and ext4 under the data=journal mode (ext4-dj), all normalized to ext4-dj. (c) shows the total writes and CPU usage with a HDD. Varmail emulates a multithreaded mail server, performing file creates, appends, deletes, reads, and* `fsync` *in a single directory. Randwrite does 200K random writes over a 10 GB file with an* `fsync` *every 100 writes. Webserver emulates a multithreaded web server performing open-read-close on multiple files and a log file append. Createfiles uses 64 threads to create 1M files. Seqwrite writes 32 GB to a new file (1 KB is considered an op in (c)). Fileserver emulates a file server, using 50 threads to perform creates, deletes, appends, and reads, on 80K files. The fileserver, varmail, and webserver workloads were run for 300 seconds. The numbers reported are the average over 10 runs.*

not incur any inherent performance overhead for standard workloads when the workload is run in one stream. False dependencies are rare and have little impact for common workloads, and the technique used to maintain order within streams in ccfs is efficient.

4.4 Case Studies

Our evaluation in the previous section shows the performance effects of maintaining order for standard benchmarks. We now consider three real-world applications: Git, LevelDB, and SQLite with rollback journaling; we focus on the effort required to maintain crash consistency with good performance for these applications in ccfs and the default mode (*data=ordered*) of ext4. For ext4, we ensure that the applications remain consistent by either modifying the application to introduce additional `fsync` calls or using safe application configuration options. All three applications are naturally consistent on ccfs when run on a single stream.

Single Application Performance. We first ran each application in its own stream in the absence of other applications, to examine if running the application in one stream is sufficient for good performance (as opposed to dividing a single application into multiple streams). Specifically, we try to understand if the applications have false dependencies. We also consider their performance when `fsync` calls are omitted without affecting consistency (including user-visible durability) on ccfs.

The results are shown in Table 4. For Git, we use a workload that adds and commits the Linux source code to an empty repository. While Git is naturally consistent atop ccfs, it requires a special option (*fsyncobjectfiles*) on ext4; this option causes Git to issue many `fsync` calls. Irrespective of this option, Git always issues 242 MB of appends and no overwrites. In ccfs, the 242 MB is sent directly to the device and the workload completes in 28.9 seconds. In ext4, the `fsync` calls needed for correctness prevent updates to metadata blocks from being coalesced; for example, a block bitmap that is repeatedly updated by the workload needs to be written to the journal on every `fsync`. Moreover, each `fsync` call forces a separate journal transaction, writing a separate descriptor block and commit block to the disk and causing two disk cache flushes. Thus, in ext4, the workload results in 1.4 GB of journal commits and takes 2294 seconds to complete (80× slower).

For SQLite, we insert 2000 rows of 120 bytes each into an empty table. SQLite issues `fsync` calls frequently, and there are no false dependencies in ccfs. However, SQLite issues file overwrites (31.83 MB during this workload), which causes data to be sent to the journal in ccfs. Sending the overwritten data to the journal improves the performance of ccfs in comparison to ext4 (1.28×). Because SQLite frequently issues an `fsync` after overwriting a small amount (4 KB) of data, ext4 incurs a seek during each `fsync` call, which ccfs avoids by writing the data to the journal. SQLite can also be heavily optimized when running atop ccfs by omitting unnecessary `fsync` calls; with our workload, this results in a 685× improvement.

For LevelDB, we use the fillrandom benchmark from the db_bench tool to insert 250K key-value pairs of 1000 bytes each to an empty database. Atop ext4, we needed to add additional `fsync` calls to improve the crash consistency of LevelDB. LevelDB on ccfs and the fixed version on ext4 have similar write avoidance, as can be seen from Table 4. Since LevelDB also does few file overwrites, it performs similarly on ccfs and ext4. With ccfs, existing `fsync` calls in LevelDB can be omitted since ccfs already guarantees ordering, increasing performance 5×.

Thus, the experiments suggest that false-dependency overheads are minimal within an application. In two of the applications, the ordering provided by ccfs can be used to omit `fsync` calls to improve performance.

Multiple Application Performance. We next test whether ccfs is effective in separating streams: Figure 6 shows the performance when running Git and SQLite

		Throu -ghput		User-level Metrics			Disk-level Metrics		
				fsync()	*Append (MB)*	*Overw- rite(kB)*	*Flushes*	Data (MB)	
								Journal	*Total*
Git	ext4	17	*files/s*	38599	242	0	77198	1423	1887
	ccfs	1351		0	242	0	10	18	243
	ccfs+	1351		0	242	0	10	18	243
SQLite	ext4	5.23	*ops/s*	6000	31.56	31.83	12000	70	170
	ccfs	6.71		6000	31.56	31.83	12000	117	176
	ccfs+	4598		0	0.32	0	0	0	0
LevelDB	ext4	5.25	*MB/s*	598	1087	0.01	1196	16.3	1131
	ccfs	5.1		523	1087	0	1046	16.2	1062
	ccfs+	25.5		0	199	0	2	0.074	157

Table 4: **Case Study: Single Application Performance.** *The table shows the performance and observed metrics of Git, LevelDB, and SQLite-rollback run separately under different file-system configurations on HDD. Ccfs+ denotes running ccfs with unnecessary* `fsync` *calls omitted; in both ccfs configurations, the application runs in a single stream. The user-level metrics characterize each workload; "appends" and "overwrites" show how much appended and overwritten data needs to be flushed by* `fsync` *calls (and also how much remain buffered when the workload ends). Overhead imposed by maintaining order will be observed by* `fsync` *calls in the ccfs configuration needing to flush more data. The disk-level metrics relate the characteristics to actual data written to the device.*

simultaneously. The situation in current real-world deployments is exemplified by the *ext4-bad* configuration in Figure 6: both applications are run on ext4, but Git runs without the *fsyncobjectfiles* option (i.e., consistency is sacrificed). The *ccfs-2* configuration is the intended use case for ccfs: Git and SQLite are in separate streams on ccfs, achieving consistency while performing similar to *ext4-bad*. (SQLite performs better under *ccfs-2* because ccfs sends some data to the journal and reduces seeks, as explained previously.) Thus, ccfs achieves real-world performance while improving correctness.

The *ccfs-1* configuration demonstrates the overhead of global order by running Git and SQLite in the same stream on ccfs; this is *not* the intended use case of ccfs. This configuration heavily impacts SQLite's performance because of (false) dependencies introduced from Git's writes. Running applications in separate streams can thus be necessary for acceptable performance.

The *ext4* configuration re-iterates previous findings: it maintains correctness using Git's *fsyncobjectfiles* on ext4, but Git is unacceptably slow due to `fsync` calls. The *ccfs+* configuration represents a secondary use case for ccfs: it runs the applications in separate streams on ccfs with unneeded `fsync` calls omitted, resulting in better SQLite performance (Git is moderately slower since SQLite uses more disk bandwidth).

Thus, running each application in its stream achieves correctness with good performance, while global order achieves correctness but reduces performance.

Developer Overhead. Achieving correctness atop ccfs (while maintaining performance) required negligible developer overhead: we added one `setstream()` call to the beginning of each application, without examining the applications any further. To omit unnecessary `fsync` calls in ccfs and improve performance (i.e., for the ccfs+ configuration), we used the `IGNORE_FSYNC` flag on the `setstream()` calls, and added `streamsync()` calls to places in the code where the user is guaranteed durability (one location in LevelDB and two in SQLite).

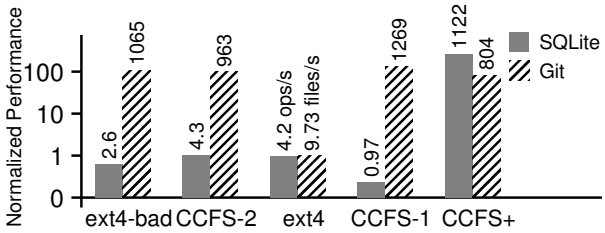


Figure 6: **Case Study: Multiple Application Performance.** *Performance of Git and SQLite-rollback run simultaneously under different configurations on HDD, normalized to performance under* `ext4` *configuration.* `Ext4-bad` *configuration runs the applications on ext4 with consistency sacrificed in Git.* `CCFS-2` *uses separate streams for each application on ccfs.* `Ext4` *uses ext4 with consistent Git.* `CCFS-1` *runs both applications in the same stream on ccfs.* `CCFS+` *runs applications in separate streams without unnecessary* `fsync`*. Workload: Git adds and commits a repository 25 times the size of Linux; SQLite repeatedly inserts 120-byte rows until Git completes.*

Correctness with ext4 required two additional `fsync` calls on LevelDB and the *fsyncobjectfiles* option on Git. The changes in ext4 both reduced performance and were complicated; we carefully used results from the Alice study to determine the additional `fsync` calls necessary for correctness. Note that, while we happened to find that Git's *fsyncobjectfiles* makes it correct on ext4, other changes are needed for other file systems (e.g., btrfs).

Thus, developer effort required to achieve correctness atop ccfs while maintaining performance is negligible; additional effort can improve performance significantly.

5 Related Work

We briefly describe how ccfs differs from previous work:

Atomicity interfaces. Transactional file-system interfaces have a long history [44] and allow applications to delegate most crash-consistency requirements to the file system. Recent work in this space includes file systems providing ACID semantics such as Amino [55], Valor [48], and Windows TxF [28], atomicity-only file systems as proposed by Vermat et al. [54], Park et al. [33], and CFS [29], and OS-level transaction support as advocated by TxOS [37]. Such interfaces allow adding crash consistency easily to applications which do not already implement them, and help heavily optimized applications that trade portability for performance [26].

For applications with existing consistency implementations, proponents of atomicity interfaces and transactional file systems advocate replacing the existing implementation with the interface provided by the file system. This is not trivial to achieve (though perhaps much easier than writing a new consistency implementation). For in-

stance, consider the SQLite database, and assume that we replace its consistency implementation using a straightforward *begin_atomic()–end_atomic()* interface provided by the file system. This does not work for two reasons. First, it does not offer SQLite's ROLLBACK command [50] (i.e., abort transaction) and the SAVEPOINT command (which allows an aborted transaction to continue from a previous point in the transaction). Second, unless the file system provides isolation (which recent research argues against [29]), it requires re-implementing isolation and concurrency control, since SQLite's isolation mechanism is inherently tied to its consistency mechanism [49]. With applications such as LevelDB, where the consistency mechanism is tightly coupled to query-efficient on-disk data structures [24, 32], adopting alternative consistency mechanisms will also cause unnecessary performance changes.

To summarize, adopting atomicity interfaces to overcome vulnerabilities is nonoptimal in applications with existing consistency implementations. One challenge is simply the changes required: CFS [29], with arguably the most user-friendly atomic interface, requires changing 38 lines in SQLite and 240 lines in MariaDB. Another challenge is portability: until the interfaces are widely available, the developer must maintain both the existing consistency protocol and a protocol using the atomic interface; this has deterred such interfaces in Linux [9]. Finally, the complexity of data structures and concurrency mechanisms in modern applications (e.g., LSM trees) are not directly compatible with a generic transactional interface; Windows TxF, a transactional interface to NTFS, is being considered for deprecation due to this [28]. In constrast, streams focus on masking vulnerabilities in existing application-level consistency implementations. Ccfs advocates a single change to the beginning of applications, and running them without more modification on both stream-enabled and stream-absent file systems.

Ordering interfaces. Fine-grained ordering interfaces [4, 5, 13] supplement the existing `fsync` call, making it less costly for applications to easily achieve crash consistency. They allow better performance, but require developers to specify the exact ordering required, and as such are not optimal for fixing existing protocol implementations. Ext4's data-journaled mode and LinLogFS [10] provide a globally ordered interface, but incur unacceptable disk-level ordering and false-dependence overhead. Xsyncfs [31] provides global order and improves performance by buffering user-visible outputs; this approach is complementary to our approach of reducing false dependencies. Other proposed ordering interfaces [8, 34] focus only on NVMs.

Implementation. Ccfs builds upon seminal work in database systems [17, 30] and file-system crash consistency [7, 11, 14, 15, 18, 40, 42, 45], but is unique in assembling different techniques needed for efficient implement of the stream API. Specifically, ccfs uses journaling [7, 18] for order within a stream, but applies techniques similar to soft updates [14, 15, 45] for separating streams. Such design is necessary: using soft updates directly for a long chain of dependent writes ordered one after the other (as ccfs promises within a stream) will result in excessive disk seeks. Block-level guarantees of atomicity and isolation, such as Isotope [46] and TxFlash [38], can simplify ccfs' separation of streams; however, techniques in Section 3 are still necessary. IceFS [25] extends ext3 to support multiple virtual journals, but requires data journaling within each journal to support ordered data writes, and hence cannot be directly used to improve application consistency without reducing performance. IceFS also does not use techniques similar to soft updates to separate the virtual journals, associating only a static and coarse-grained partition of the file-system namespace to each journal (compared to the dynamic and fine-grained stream abstraction).

In principle, one should be able to easily construct a stream-ordered file system atop a fine-grained ordering interface. However, the direct implementation of ordering in Featherstitch [13] uses the soft-updates approach, which is incompatible as described. OptFS' interface [5] is insufficient for implementing streams. Ccfs uses the SDJ technique from OptFS but optimizes it; the original relies on specialized hardware (durability notifications) and decreased guarantees (no durability) for efficiency.

6 Conclusion

In this paper, we present the stream abstraction as a practical solution for application-level crash consistency. We describe the stream API and the ccfs file system, an efficient implementation of the API. We use real applications to validate consistency atop the file system and compare performance with ext4, finding that ccfs maintains (and sometimes signficantly improves) performance while improving correctness. Our results suggest that developer effort for using the streams API is negligible and practical.

Acknowledgments. We thank the anonymous reviewers and Ashvin Goel (our shepherd) for their insightful comments. We thank the members of the ADSL for their valuable discussions. This material was supported by funding from NSF grants CNS-1419199, CNS-1421033, CNS-1319405, and CNS-1218405, DOE grant DE-SC0014935, as well as donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Samsung, Seagate, 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, DOE, or other institutions.

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.9 edition, 2014.

[2] Andrew D. Birrell. An Introduction to Programming with Threads. Technical Report SRC-RR-35, January 1989.

[3] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 21)*, Atlanta, Georgia, April 2016.

[4] Nathan C. Burnett. *Information and Control in File System Buffer Management*. PhD thesis, University of Wisconsin-Madison, October 2006.

[5] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[6] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, pages 101–116, San Jose, California, February 2012.

[7] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode File System. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, pages 43–60, San Francisco, California, January 1992.

[8] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[9] Jonathan Corbet. Better than POSIX?, March 2009. Retrieved April 2016 from https://lwn.net/Articles/323752/.

[10] Christian Czezatke and M. Anton Ertl. LinLogFS: A Log-structured Filesystem for Linux. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, San Diego, California, June 2000.

[11] Linux Documentation. XFS Delayed Logging Design. Retrieved April 2016 from https://www.kernel.org/doc/Documentation /filesystems/xfs-delayed-logging-design.txt.

[12] Filebench. Filebench. Retrieved March 2016 from https://github.com/filebench/filebench/wiki.

[13] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized File System Dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 307–320, Stevenson, Washington, October 2007.

[14] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2), May 2000.

[15] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[16] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, Seattle, Washington, May 1992.

[17] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[18] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[19] Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-lists. DISC, 2001.

[20] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi 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 Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[21] Maurice Herlihy. Wait-Free Synchronization. *Transactions on Programming Languages*, 11(1), January 1991.

[22] D. M. Jacobson and J. Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical Report HPL-CSP-91-7, Hewlett Packard Laboratories, 1991.

[23] Jaeho Kim, Jongmoo Choi, Yongseok Oh, Donghee Lee, Eunsam Kim, and Sam H. Noh. Disk Schedulers for Solid State Drives. In *EMSOFT*, Grenoble, France, October 2009.

[24] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, California, February 2016.

[25] 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, Colorado, October 2014.

[26] MariaDB. Fusion-io NVMFS Atomic Write Support. Retrieved April 2016 from https://mariadb.com/kb/en/mariadb/fusion-io-nvmfs-atomic-write-support/.

[27] Mercurial. Dealing with Repository and Dirstate Corruption. Retrieved April 2016 from https://www.mercurial-scm.org/wiki/RepositoryCorruption.

[28] Microsoft. Alternatives to using Transactional NTFS. Retrieved April 2016 from https://msdn.microsoft.com/en-us/library/hh802690.aspx.

[29] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '15)*, Santa Clara, CA, July 2015.

[30] 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*, 17(1):94–162, March 1992.

[31] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 1–16, Seattle, Washington, November 2006.

[32] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[33] Stan Park, Terence Kelly, and Kai Shen. Failure-Atomic Msync (): a Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the EuroSys Conference (EuroSys '13)*, Prague, Czech Republic, April 2013.

[34] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA '14)*, Minneapolis, MN, USA, June 2014.

[35] 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.

[36] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash Consistency. *Communications of the ACM*, 58(10), October 2015.

[37] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating Systems Transactions. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[38] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proceedings of the 8th Symposium on Operating Systems Design*

and Implementation (OSDI '08), San Diego, California, December 2008.

[39] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. Advances in Neural Information Processing, 2011.

[40] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[41] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[42] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '93)*, pages 307–326, San Diego, California, January 1993.

[43] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '90)*, pages 313–324, Washington, D.C, January 1990.

[44] Margo I. Seltzer. *File System Performance and Transaction Support*. PhD thesis, EECS Department, University of California, Berkeley, Jun 1993.

[45] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 71–84, San Diego, California, June 2000.

[46] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Isotope: Transactional Isolation for Block Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, California, February 2016.

[47] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A Primer on Memory Consistency and Cache Coherence. Synthesis Lectures on Computer Architecture, November 2011.

[48] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling Transactional File Access via Lightweight Kernel Extensions. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, San Francisco, California, February 2009.

[49] SQLite. Isolation In SQLite. Retrieved Dec 2016 from https://www.sqlite.org/isolation.html.

[50] SQLite. SQL As Understood By SQLite. Retrieved Dec 2016 from https://www.sqlite.org/lang.html.

[51] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *;login: The USENIX Magazine*, 41(1), June 2016.

[52] Linus Torvalds. Linux 2.6.29. Retrieved April 2016 from https://lkml.org/lkml/2009/3/25/632.

[53] Theodore Ts'o. ext4: remove calls to ext4_jbd2_file_inode() from delalloc write path. Retrieved April 2016 from http://lists.openwall.net/linux-ext4/2012/11/16/9.

[54] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Srivilliputtur Mannarswamy, Terence P. Kelly, and Charles B. Morrey III. Failure-Atomic Updates of Application Data in a Linux File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, California, February 2015.

[55] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID Semantics to the File System Via Ptrace. *ACM Transactions on Storage (TOS)*, 3(2):1–42, June 2007.

[56] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[57] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, October 2014.

High-Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System

Harendra Kumar*
Independent Researcher
harendra.kumar@gmail.com

Yuvraj Patel*
University of Wisconsin–Madison
yuvraj@cs.wisc.edu

Ram Kesavan & Sumith Makam
NetApp
{ram.kesavan, makam.sumith}@gmail.com

Abstract

We introduce a low-cost incremental checksum technique that protects metadata blocks against in-memory scribbles, and a lightweight digest-based transaction auditing mechanism that enforces file system consistency invariants. Compared with previous work, our techniques reduce performance overhead by an order of magnitude. They also help distinguish scribbles from logic bugs. We also present a mechanism to pinpoint the cause of scribbles on production systems. Our techniques have been productized in the NetApp® WAFL® (Write Anywhere File Layout) file system with negligible performance overhead, greatly reducing corruption-related incidents over the past five years, based on millions of runtime hours.

1 Introduction

Storage systems comprise unreliable hardware components such as disks [6, 51, 7], disk shelves, storage interconnect fabric, RAM [52], CPU [45, 56] and data transport buses. This hardware is driven by a software stack or by a dedicated storage operating system that is built around a file system such as ext3 [60], ext4 [42], ZFS [13], btrfs [49], or WAFL [32, 24]. The software is built from heterogeneous components that might also be unreliable because of inherent bugs that could affect other parts of the software ecosystem. Hardware failures and bugs in software can both corrupt data [5, 58, 61]. The file system must provide mechanisms to detect, avoid, and recover from such corruptions [6].

In general, data can be corrupted before it is written to persistent media, while it is residing on the media, or in the read path. Data at rest is protected from media failures by using detection techniques such as checksums [8, 13, 57] and scrubbing [53], and by using recovery techniques such as redundancy [47]. File system crash consistency is provided by techniques such as journaling [30, 59, 40, 50, 12], shadow paging [32, 13, 49], or soft updates [28]. However, memory scribbles that are caused by software bugs [3, 1, 2, 15, 62] or by hardware failures [9, 44, 46, 52, 39, 65], or logic bugs that are in the file system code can still introduce metadata inconsistencies in the write path.

The transaction auditing mechanism Recon [27] is a promising method to improve write integrity. However, with Recon the original version of metadata blocks must be cached and later compared with the modified copies for the audit. In copy-on-write file systems such as WAFL, this requirement can lead to heavy performance regression, especially under metadata-heavy workloads. A highly optimized WAFL implementation that used Recon-like auditing resulted in an unacceptable 30% throughput regression for critical workloads. Furthermore, Recon-like auditing cannot distinguish corruptions that are caused by memory scribbles from those that are caused by logic bugs. For an industrial-scale deployment, the ability to distinguish between the two causes is crucial for fast resolution of corruption bugs.

In this paper, we introduce two novel techniques that in combination can detect arbitrary inconsistencies more efficiently than Recon can. They also provide the crucial ability to distinguish between scribbles and logic bugs. First, we introduce a single rolling checksum through the lifetime of each metadata block – whether in-memory or on persistent media – to protect it against random scribbles. Next, we introduce a digest-based transaction auditing system to prevent logic bugs. Unlike Recon, digest-based auditing does not require caching the original versions of metadata blocks, and is therefore realized with negligible performance overhead. By having separate solutions for scribbles and logic bugs, we can prioritize implementation of auditing invariants based on the return on investment, which is invaluable for a large and complex file system implementation such as WAFL. In

*Research performed while working at NetApp

addition, the file system is doubly protected from scribbles by both techniques. Finally, we also introduce a low-overhead page protection technique that pin-points the root cause of software-caused scribbles, thereby providing quick resolution to corruption bugs.

These solutions have been field-tested across almost a quarter million NetApp ONTAP® customer systems over the past five years, and the data shows that we have attained among the highest levels of data integrity. The overall performance penalty that these data integrity mechanisms incur is almost unnoticeable. They have led to a more than three times reduction in inconsistencies due to software caused corruptions. To the best of our knowledge, we have not encountered a single corruption bug, across our entire customer deployment, that managed to get past the mechanisms that we propose in this paper. Reported corruptions have affected only metadata that were not protected by these mechanisms in the corresponding releases. In addition, by detecting problems early in the development cycle, the solutions have significantly improved engineering productivity.

The primary contributions of this paper are: (1) end-to-end metadata checksums, (2) lightweight digest-based transaction auditing, and (3) fine-grained page protection to pin-point the scribbling code-path.

2 Motivation

NetApp is a storage and data management company that offers software, systems, and services to manage and store data, including its flagship ONTAP operating system [34]. ONTAP implements a proprietary file system called WAFL (Write Anywhere File Layout) [32]. Numerous ONTAP systems are deployed across the world, and various hardware and software bugs have been incurred by them over the past two decades. Some of these incidents have resulted in latent file system inconsistencies that were discovered much later.

When these inconsistencies are discovered, the file system is marked inconsistent and a file system check is initiated. WAFL provides both online [35] and offline [33] file system consistency checks. Offline checking involves significant downtime, which is proportional to the dataset size. Online checking causes less downtime but affects system performance until it has finished. On many occasions, the checks end up discovering data loss, identifying damaged files, or even suggesting recovery from backups.

Although it is extremely rare for a given WAFL file system to end up inconsistent, the sheer number of customer systems that collectively log millions of runtime hours every day make it a more likely occurrence for NetApp's technical support staff. Our main goal was to fortify WAFL against inconsistencies to reduce both disruption for our customers and support costs for NetApp.

2.1 Write Path Metadata Corruptions

Two types of problems can cause file system recovery runs: (1) inconsistencies due to in-memory corruptions of metadata in the write path, and (2) inconsistencies due to the loss of metadata because of media failures that are beyond the redundancy threshold of the underlying RAID mechanism. This paper addresses problem 1. Note that persistent block checksums cannot protect the metadata from in-memory corruptions because the checksum computation occurs after the block is scribbled [64].

Metadata corruption in the write path can result from three primary causes: (1) logic bugs in metadata computation and updates, (2) scribbles of metadata blocks that are used as input for new metadata computation, and (3) scribbles of metadata blocks before they are written to persistent storage.

2.2 Scribbles Versus Logic Bugs

Let us first distinguish between two types of corruption bugs: scribbles and logic bugs.

A scribble (Heisenbug) [29] overwrites an arbitrary data element. It usually occurs randomly and is equally likely to corrupt any data or metadata, producing unpredictable results. Because of its unpredictable nature, a scribble is difficult to reproduce with systematic testing. It is difficult to diagnose because the observed symptoms are far removed from the original location and time of its occurrence. Mistaking a scribble for a logic bug can result in a futile bug chase, and wastage of engineering resources. Often, scribbles are the cause of known but unfixed bugs that adversely affect product quality.

Most hardware scribbles go undetected. ECC memory [16] can detect failures but with some limitation [25], and it might not always be used because of its cost [31, 36]. Software-induced scribbles can be detected using the processor's page protection mechanism, but again with significant costs and limitations.

In contrast, a logic bug (Bohrbug) [29] is inherent to the metadata computation and update logic. A computation with a logic bug generates incorrect outcome but stores it to its own memory location. Therefore, a logic bug is more predictable and limited in its impact. The observed symptoms are predictably confined to a particular behavioral aspect, which makes it easier to diagnose.

2.3 Causes of Corruptions

Scribbles can be caused by software bugs such as buffer overflow, incorrect pointer computation, and dangling pointers in a shared address space [11]. In WAFL, the file system buffer cache memory can be allocated away for other uses and then be recycled back, which makes file system buffers, including metadata buffers, likely victims of software scribbles.

Scribbles are also caused by hardware failures [55] such as memory errors, DMA errors, or CPU register bit flips [10] that remain undetected. Scribble bugs are not that rare [52], and memory scribbles due to hardware failures are expected to be more common in the future [14, 55]. It has been shown that scribbles can be induced by external attacks in a controlled way on a shared infrastructure. [38, 48].

A scribble of an intermediate result or a key data structure can also produce a second order corruption. In our experience, this class of bugs is extremely rare.

Logic bugs are typically found in insufficiently tested code; mature, field-tested code is less likely to have them.

2.4 WAFL File System Overview

Next, we briefly introduce WAFL before we evaluate an existing auditing solution for copy-on-write (COW) file systems. WAFL is a UNIX-style file system with a collection of inodes that represent its files [32]. The file system is written out as a tree of blocks that are rooted at a superblock. Every file system object in WAFL, including metadata, is a file. WAFL is a COW file system, in which every modified block is written to a new location on disk. Only the superblock is ever written in place.

As buffers and inodes are modified (or dirtied) by client operations, they are written out in batches for performance and consistency. Every mutable client operation is also recorded to a log in nonvolatile memory (NVRAM) before it is acknowledged; the operations in the log are replayed to recover data if a crash occurs. WAFL collects the resultant dirty buffers and inodes from hundreds of thousands of logged operations, and uses a checkpoint mechanism called a *consistency point* (CP) to flush them to persistent media as one large transaction. Each CP is an atomic transaction that succeeds only if all of its state is successfully written to persistent storage. Updates to in-memory data structures are isolated and targeted for a specific CP so that each CP represents a consistent and complete state of the file system. When the entire set of new blocks that belong to a CP is persisted, a new file system superblock is atomically written in place that references this new file system tree [32, 24].

2.5 Recon for COW File Systems

Recon is a transaction auditing mechanism that verifies all file system consistency invariants before a transaction is committed. Recon examines physical blocks below the file system, and infers the types of metadata blocks when they are read or written. This allows parsing and interpretation of the blocks, similar to semantically smart disks [54]. However, because of increased metadata overhead per transaction, metadata caching and comparison in Recon's design becomes unsustainable for COW file systems such as WAFL.

In WAFL, each write to a user data block causes the file system to read in the corresponding parent indirect block, to free the old block pointer to this user data block, and to allocate a new block pointer. This process recurses up the tree of blocks that constitute the file system image, all the way up to the super block. Thus, writing to a single user data block might require reading and writing more than one metadata block. Even though WAFL amortizes this overhead by batching numerous operations in a single transaction, the overhead for a Recon-like audit is still significantly high.

Let us analyze the cost of verifying the block accounting metadata in WAFL. When a block pointer is inserted into or deleted from an indirect block, WAFL sets or clears the corresponding bit in its bitmaps. A Recon-like audit compares the persistent versions of indirect blocks with the modified versions that are being committed to generate a list of block pointers that are allocated or freed in the current transaction. This list is then tallied with the corresponding changes in bitmap bits, which are obtained by comparing the persistent and modified versions of the corresponding bitmap blocks. To enable this comparison, a copy of the persistent version of each metadata block is cached in memory before the block is modified.

The cost of a Recon-like audit primarily includes: (1) making a copy of the indirect block before modification, (2) making copies of the bitmap blocks to record the allocated and freed blocks, (3) comparing the unmodified and modified versions of the indirect blocks and the bitmap blocks, and (4) verifying that the changes in the bitmap blocks are consistent with the changes in the indirect blocks. This process involves significant CPU cost and memory bandwidth, and if the metadata blocks are to be read from persistent media, it also involves significant I/O cost.

The memory requirement of a Recon-like audit scales up with the number of modified metadata blocks that are in the transaction. Caching the pristine versions of modified metadata blocks becomes impractical. Many of those blocks might be evicted from the cache and therefore must be read in from persistent storage just in time for

the audit, and the additional I/O directly affects the cost of auditing. Moreover, the increased memory pressure can force us to trigger transactions early, which results in smaller batch sizes and increased metadata overhead, in turn reducing the overall write throughput.

The audit of a consistency invariant cannot begin until the corresponding metadata are finalized. Self-referential metadata, such as bitmaps, are not finalized until the very end of a CP [37], and only then can they be audited. This serialization reduces the I/O efficiency of the CP, which in turn negatively affects the achievable write throughput of the file system.

2.6 Drawbacks of Recon

Performance: The designs presented in this paper were conceptualized before the publication of Recon, and coincidentally, we first tried the same metadata caching approach. A highly optimized prototype of Recon-style auditing for bitmap and indirect block cross consistency checks showed a performance degradation of approximately 30% on our internal database/OLTP benchmark; the benchmark is described in Section 7. The authors of the original Recon paper observed a similar performance penalty in their research prototype implementation for btrfs [22].

Diagnostics: Although transaction auditing detects the presence of an inconsistency, it cannot distinguish between a scribble and a logic bug. When the inconsistency is due to a scribble, the audit cannot identify the affected metadata block; it can only point to the set of blocks that were used to verify the invariant that failed its check.

Implementation challenge: Because scribbles can affect any metadata without bias, a foolproof transaction auditing needs consistency invariants that cover the entire metadata. Building such an exhaustive set of invariants is quite expensive in terms of both system performance and engineering resources, and it has a low return on investment for mature, field-tested code with no logic bugs.

3 Approach

3.1 Goals

For ONTAP, performance and field diagnostics are both as important as data integrity.

Performance: A 30% throughput regression to protect against a relatively rare though disruptive event is considered too high a price by the business. To be competitive, we need a solution with excellent metadata integrity protection that costs an order of magnitude less in terms of performance.

Diagnostics: Corruptions in general and scribbles in particular, are the hardest to diagnose. For faster diagnosis, we need the ability to distinguish scribbles from logic bugs. As explained in section 2.2, they are different in nature and require different techniques to determine the root cause. The diagnostic capability is even more important during product development when the likelihood of scribbles is higher and many person-hours are wasted on corruption bugs. Worse, the product sometimes ships with known, but unreproducible bugs that remain undiagnosed.

3.2 Solution Overview

In-memory metadata checksums: To distinguish scribbles from logic bugs and to identify the scribbled block, we use a general checksum protection for in-memory metadata blocks. Before a block is written to persistent storage, a checksum verification determines whether the block was scribbled.

Metadata page protection: Even though checksum verification prevents scribbles from being persisted, it cannot implicate the culprit code. We use processor assisted page granular protection to catch the culprit code. Because this approach has a higher performance tax, we recommend enabling it on a customer's system only when a metadata block checksum failure is reproducible.

Transaction auditing: Consistency invariants are verified by comparing digests of changes that occurred in a transaction. It is very efficient to create and compare digests. Changes to a metadata block are added to a digest as and when it is modified. Thus, we do not need cached copies of the original blocks.

By having a separate solution for scribble detection, we have the flexibility to prioritize the implementation of auditing invariants that yield higher return on investment. We can start with code-paths that are prone to logic bugs, such as code-paths with higher complexity or code-paths that correspond to newer, untested features.

4 Scribble Protection

In this section, we describe a high-performance and multiprocessor-capable incremental checksum scheme to protect in-memory metadata against scribbles. We also present some limitations, after which we present the page protection mechanism that overcomes the most important limitation.

The goal is to detect any unauthorized change to metadata blocks that are written out in a transaction. Because we are trying to protect against scribblers that share the same address space, we cannot use address space iso-

lation as a protection technique. The proposed page-granular protection in Section 4.7 is quite expensive, and therefore is used only in diagnostic mode. Furthermore, none of these techniques can detect a corruption that is caused by hardware problems.

Checksumming is a well-known data integrity protection mechanism. Block checksums are widely used to protect the integrity of the blocks of the file system that reside on persistent storage. Z^{2}FS [63] even proposes using a single checksum as well as a checksum handover scheme to protect the data while in transit across heterogeneous software components.

However, none of the known schemes protect the data while it resides in memory. In this section, we describe a scheme that can be used effectively to protect in-memory metadata on a large scale, and with negligible performance overhead.

4.1 End-to-End Checksum

We use a single rolling or incremental checksum [4, 41] to protect each metadata block through its entire life cycle, whether it is in memory or on persistent storage. When a block is read in from persistent storage, the stored checksum of the block is also read and then is verified by the file system. While in memory, this verified checksum is used to protect the block and is updated incrementally by every legal update of the block.

When a block is written out to persistent storage as part of a transaction, a fresh checksum is computed by the file system. This freshly computed checksum is used to verify the incrementally updated in-memory checksum, and if the verification fails, we know that the block has been scribbled. That way, the block remains protected whether it is in memory or on persistent storage.

Note that the file system already pays for the cost of checksum computation during each block read and write. The only additional work that is needed is the incremental update of that checksum whenever the block is modified in memory.

4.2 Checksum Initialization

A metadata block that is written out by a transaction is either newly created or is an existing block that was read from persistent storage earlier but was modified during the transaction. We initialize the in-memory checksum for a block as soon as it comes into existence in memory. If newly created, the checksum is initialized to a derivable constant value based on the zero-state of the corresponding metadata. If read in from storage, the checksum is initialized to the stored and verified checksum.

4.3 Incremental Checksum Updates

Legal modification of a metadata block in memory is typically accomplished by invoking well-defined APIs in the file system code. We add a hook in each of these APIs so that the corresponding block checksum is kept up to date on modifications. Before a block is modified, the new checksum is computed incrementally by using the old checksum, the original data that is being overwritten, the position of the original data in the block, and the new data. In the next few paragraphs, we show precisely how to incrementally recompute an Adler [23] checksum when a single byte in a block is modified.

In the following equations, "div" and "mod" represent operators that return a quotient and a remainder, respectively, in an integer division. Assuming that a data block D is composed of n bytes $D_1, D_2, ..., D_n$, the *Adler32* checksum of D is computed as follows:

$$A = (1 + D_1 + D_2 + ... + D_n) \bmod 65521$$
$$B = (n \times D_1 + (n-1) \times D_2 + ... + D_n + n) \bmod 65521$$
$$Adler32(D) = A + B \times 65536$$

If the original checksum of block D is C, and we replace byte D_i in the block with a new byte D'_i, then the new checksum *Adler32Incr* can be computed as follows:

$$A = C \bmod 65536$$
$$B = C \text{ div } 65536$$
$$\Delta D_i = D'_i - D_i$$
$$A' = (A + \Delta D_i) \bmod 65521$$
$$B' = (B + (n + 1 - i) \times \Delta D_i) \bmod 65521$$
$$Adler32Incr(C, D_i, D'_i, i) = A' + B' \times 65536$$

Similarly, we can also recompute the new checksum incrementally when multiple contiguous bytes are modified in the block.

If any portion of a block is scribbled, its incremental checksum becomes inconsistent with respect to its contents, and it remains so even after any number of subsequent legal updates to the block. Thus, when the file system eventually recomputes the full checksum (before it writes the block to persistent storage) by using the current contents of the block, the full checksum does not match the incremental checksum.

The incremental checksum update is optimal because it requires additional memory accesses only to read the original contents, to read the old checksum, and to update the checksum. In most cases, the file system code reads the metadata before overwriting it anyway; therefore, the original contents are likely to be in the processor cache. The CPU cost of computing the new checksum is proportional to the amount of data that is modified. In terms of the memory overhead, this approach requires an additional 4 bytes per data block to maintain the rolling checksum.

If preferred, a stronger 64-bit checksum can be used. Because they are amenable to this form of incremental computation, Adler [23], Fletcher [26], or any other position-dependent checksum can be used. For better performance, a chunk size larger than a byte can be used in the checksum computation. For WAFL, we used a modified version and a highly optimized implementation of the simple incremental checksum computation shown previously.

4.4 Lockless Multiprocessor Updates

WAFL is designed to run simultaneously on several processors [21]. Thus, it is quite common for a metadata block to be modified concurrently by multiple processors. However, to minimize cache-line thrashing and expensive lock contention, the WAFL multiprocessor programming model [21] avoids using spinlocks as much as possible.

To avoid lock contention, an incremental checksum for a block can be split across processors and be updated in a lockless manner. Each processor computes and accumulates the delta checksums for its own updates to the block in a per-processor checksum field. The per-processor deltas are then combined to derive the final incremental checksum.

Two independent checksum fragments, C_1 and C_2, can be combined as follows:

$A = (C_1 \bmod 65536 + C_2 \bmod 65536) \bmod 65521$
$B = (C_1 \operatorname{div} 65536 + C_2 \operatorname{div} 65536) \bmod 65521$
$combine(C_1, C_2) = A + B \times 65536$

Two processors that modify bytes D_i and D_j simultaneously can maintain their respective per-processor checksum fragments, C_1 and C_2. Each fragment is updated independently using *Adler32Incr* from the previous section. Before verification, the fragments are combined with the original checksum of the block, C, to arrive at the final checksum C':

$C_1 = C_2 = 0$
$C'_1 = Adler32Incr(C_1, D_i, D'_i, i)$
$C'_2 = Adler32Incr(C_2, D_j, D'_j, j)$
$C' = combine(C, combine(C'_1, C'_2))$

However, each per-processor checksum fragment requires additional memory (4 bytes) per block. This extra memory is quite a worthwhile trade-off because it saves us the cost of acquiring and contending on spinlocks.

4.5 Checksum Verification

When a block is written out to persistent storage as part of a transaction, we recompute the checksum on the entire block and compare it with the incremental checksum. If the two checksums do not match, we have detected a scribble on the block.

This verification can be performed either before or after the write I/O for the block is issued to storage. In the former case, checksum computation can be optimized by combining it with RAID parity computation [17]. However, this approach opens a window for undetectable scribbles after the checksum computation but before the write I/O is completed. Verification of the checksum after the completion of the write I/O closes that window. In any case, corruptions that are injected by the data transfer fabric while it services the write I/O cannot be detected until the block is read again.

Upon detection of a scribble, we abort the ongoing transaction commit, preventing the corruption from being persisted. To protect the ONTAP node from any other potential corruptions from the same bug, we reboot the node instead of aborting an individual transaction. Because ONTAP is configured in high-availability pairs, the partner node takes ownership of the rebooted node's file systems, and those file systems are all still consistent because they are defined by their most recently completed transaction. The partner node then replays the user operations that are recorded in the NVRAM log, and the prior and consistent metadata is read from storage, is modified, and then is committed as part of a brand-new transaction.

4.6 Assumptions and Limitations

Incremental checksumming is robust against bugs in the checksum update code. If we miss adding the incremental checksum hook to any of the legal APIs that modify metadata, then any transaction that includes a call to the API results in a checksum verification failure, thereby forcing us to fix the bug.

This mechanism helps distinguish scribbles from logic bugs, but it does not implicate the culprit code-path that tampered with the memory of the corrupted block.

The checksum may not be strong enough to detect all kinds of corruptions. Adler and Fletcher checksums are known to be stronger against larger errors than against smaller ones [43]. They work quite well for storage media failures because the corruption size is usually bigger; however, some in-memory corruptions can be smaller, and therefore the chances that they go undetected are somewhat higher. In the unlikely event that a scribble remains undetected by the checksum, it will certainly be detected by the transaction auditing (described in the next section). However, such a corruption might, unfortunately, be attributed to a logic bug.

If a bug causes an incorrect block pointer to be supplied as an argument to a legal metadata update API, the API

will modify the wrong block and its corresponding incremental checksum. This bug will not be detected by checksum verification, but will be detected by the transaction auditing mechanism. We have not come across this sort of bug in our experience with WAFL. One potential solution is to use different signatures to categorize the callers of the APIs into different groups; for example, each group could rotate its incremental checksum by a specific number of bits. That way an updater from an incorrect group would result in an incorrect checksum.

The incremental checksum technique can also be used to protect other important data structures that participate in any periodic transactional episode.

4.7 Diagnostics Using Page Protection

A checksum verification failure indicates that the block has been scribbled, but it does not implicate the culprit code-path because the scribble might have occurred long ago. Regular address space-based protection cannot help if the culprit code-path shares the same address space; this is true of much of the kernel code in ONTAP.

If a scribble is reproducible, we provide an option to enable metadata page protection to directly implicate the code-path. To provide good performance even under heavy and frequent modification of metadata, we use a combination of page-level protection and the Write-Protect Enable (WP) bit in the x86-64 processors [18].

To protect metadata blocks from scribbles we keep the individual metadata pages read-only by default. One way to allow safe updates would be to mark the corresponding page read-write just before a legal updater modifies the block. When the requested modification has finished, the page can be marked read-only again. Thus, an illegal modification finds the page read-only, generates a page fault, and yields a stack trace that points to the scribbler.

However, this scheme does not perform well if metadata is modified frequently. The frequent switching of page permissions (read-only to read/write to read-only) not only causes a flood of TLB flushes, but also creates a storm of inter-processor TLB invalidation interrupts. Using this scheme to protect all indirect blocks and bitmap blocks degraded performance by approximately 70%, in our experiments, thereby rendering the scheme unusable even in debug mode.

To reduce the performance impact, we keep the pages read-only all the time. To enable legal writes, we disable protection globally (by flipping the WP bit) before modification and we re-enable it after modification. With this approach, the performance degradation comes down to about 20%, which is acceptable for a diagnostic mode.

When write protection is disabled on a CPU core, it can write to any address. This ability implies two risks: (1) the metadata modification code itself might scribble, or (2) interrupt handlers that are serviced by the processor during that window might scribble otherwise read-only memory. However, these risks are close to zero because these code blocks (the metadata update APIs and interrupt handlers) are typically small pieces of code and are extremely well tested. Furthermore, we perform address range checks on the target address before we disable protection.

As Section 7.3 shows, this feature has proved to be invaluable for product development and has been used in the field as well. Moreover, as explained in Section 6, it has been an invaluable tool for quickly identifying code-paths that required our incremental checksum hooks

5 Transaction Auditing

In this section, we introduce a digest-based transaction auditing technique, explain what invariants it checks by using an example, breakdown its performance, and analyze its limitations. The audit verifies that the changes to the file system state being committed in a given transaction are self-consistent. There are two categories of consistency invariants: local and distributed:

Local consistency invariant: A local consistency invariant is confined to a given metadata block. For example, all block pointers in an indirect block must be within the file system block number range. Such an invariant is inexpensive to check because it does not require loading any other blocks.

Distributed consistency invariant: A distributed invariant defines a consistency relationship across several blocks from different metadata. For example, when a block pointer is cleared from an indirect block, the corresponding bit in the bitmap block must be cleared. A distributed consistency invariant is expensive to check because it requires identifying the changes to several blocks from different metadata.

In contrast to Recon, we intercept modifications to metadata at the file system layer. During a transaction, changes are accumulated to create digests that are used later to verify consistency invariants. This design obviates the need to cache the unmodified metadata blocks. Unlike Recon, the cost of recording the changes is proportional to the actual changes rather than to the number of modified blocks.

5.1 Digest-Based Auditing

Digest-based audits help verify distributed consistency invariants inexpensively. The key idea is to create digests

of changes and to cross-verify the digests rather than individual changes. This strategy drastically reduces the amount of work that is required in verification and still provides strong enough guarantees to be useful in practice.

To illustrate the mechanism, let us use cross consistency between indirect blocks and bitmap blocks as an example. We intercept indirect and bitmap block modification operations by using the aforementioned hook in the modification APIs, and we create a digest of those modifications.

Let us say that an indirect block contains 64-bit block numbers $B_1, B_2, ..., B_n$. Suppose block numbers B_i, B_j, and B_k were replaced in a transaction by newly allocated block numbers N_i, N_j, and N_k. A hook in the API that modifies the indirect block updates checksums of the original block numbers in a *free digest* and the new block numbers in an *allocated digest*; each digest is maintained per transaction across all updates to the metadata of the file system:

$\Sigma IndFree = B_i + B_j + B_k + ...$
$\Sigma IndAlloc = N_i + N_j + N_k + ...$

Similarly, a hook in the bitmap block modification API maintains per transaction digests of all block numbers whose corresponding bits flipped from 0 to 1 (new allocations) and vice-versa (frees):

$\Sigma BitmapFree = B_i + B_j + B_k + ...$
$\Sigma BitmapAlloc = N_i + N_j + N_k + ...$

At the end of the transaction, the audit verifies that the two sets of digests agree with each other:

$\Sigma IndFree == \Sigma BitmapFree$
$\Sigma IndAlloc == \Sigma BitmapAlloc$

Similarly, several other distributed consistency invariants can be inexpensively verified by using the digest scheme; Section 5.4 describes them. Audit digest verification failure is handled in the same way as checksum verification failure, as explained in Section 4.5.

5.2 Audit Performance

A digest-based audit performs much better than a Recon-like audit because digests are inexpensive to compute and metadata blocks need not be cached and compared. Specifically: (1) the data that we are adding to the digest is readily available in the processor's cache because it has just been accessed; (2) digest update involves just one addition operation and one memory access for each metadata update; (3) there are no expensive I/O operations due to buffer cache misses; and (4) the digest is created incrementally with each metadata update operation which eliminates the need for an exclusive phase in which we identify and verify all changes individually.

Note that digest update is performed together with the block's incremental checksum update by using the same hook. Thus, we efficiently use one memory access to do three things: (1) modify the indirect block, (2) update the audit digest, and (3) update the incremental checksum. The final verification is an inexpensive comparison of a few bytes.

A digest-based audit of bitmap and indirect block changes drastically reduced the overall cost of transaction auditing from 30% throughput regression to less than 2% on our database/OLTP benchmark.

5.3 Strengths and Weaknesses

The audit is provably robust with respect to bugs in the digest update code. If we miss adding the hook to update the digest in any of the legal APIs that are used to modify metadata, the corresponding digest verification fails.

A digest is typically a simple sum without any position-related information because we compare sets and not sequences. In theory, logic bugs can result in just the "right" pattern of incorrect updates that a digest-based verification cannot detect. In the previous example, it is possible for the two summations to match even if the actual updates were incorrect. Over the past five years of this feature's existence, such a case has never been hit in internal development or in the field. File systems have been corrupted only because the audit infrastructure of the corresponding ONTAP release did not include the invariant for a particular consistency property.

5.4 List of Distributed Invariants

In addition to the invariant explained in Section 5.1, we check many other distributed invariants of the WAFL file system as part of the audit. Most of them are inexpensive and are enabled by default in production systems. A few of them are somewhat expensive and might be disabled by default in production on some specific low-end configurations with insufficient CPU horsepower.

Table 1 shows a subset of the distributed invariants that we have implemented. We do not present other invariants that are very specific to the persistent layout of the WAFL file system, and that require more background to explain. Note that all invariants were not implemented in one go; rather, they were implemented in phases across several releases based on the return on investment.

	Description of Equation
1	Each inode tracks a count of all blocks to which it points. The file system also maintains a total count of all the allocated blocks. Their deltas much match.
2	The bitmap uses a bit to track the allocated state of each block in the file system [37]. The file system also maintains a total count of all allocated blocks. The delta of the latter must equal the delta of the number of bits that flipped to 1 (i.e., allocated) minus the number of bits that flipped to 0 (i.e., free).
3	The inode metadata tracks the allocation status of each inode. The file system maintains a total count of all allocated inodes. The delta of the latter must equal the number of inodes that changed state from free to allocated minus the number of inodes that changed state from allocated to free.
4	The inode metadata tracks deleted inodes that are moved to a hidden namespace awaiting block reclamation [37]. The current file system maintains a total count of these hidden inodes. The delta of the latter must equal the number of inodes that were deleted (i.e., moved into the hidden namespace) minus the number of inodes that were removed from the hidden namespace after all their blocks had been reclaimed.
5	The *refcount* file maintains an integer count to track all extra references to each block; WAFL uses this file to support de-duplication. The file system maintains the physical space that is saved by de-duplication as a count of blocks. Their deltas much match.
6	Each inode tracks a count of physical blocks that are saved by compression. The file system maintains the physical space that is saved by compression as a count of blocks. Their deltas must match.

Table 1: Some important audit equations implemented in WAFL

6 Implementation

Intercepting modifications: All three features – incremental checksum, page granular protection, and transaction auditing – must intercept all modifications to a protected metadata block. We inserted a unified hook into every legal API that is used to update metadata. The API supplies all the requisite parameters to the hook to update the incremental checksum, to update the corresponding digests, and to toggle memory protection.

Almost all modifications to metadata blocks go through well-known WAFL APIs, so it was easy to insert our hooks inside those APIs. However, given more than two decades' worth of code growth and churn, there were a few hidden and somewhat obscure places in the code that updated metadata blocks directly; we used the page protection feature to find them. All metadata pages are read-only by default, and the hook is needed to toggle that mode. When the feature was turned on, any update of metadata from an obscure code-path immediately generated a page fault with a useful stack trace, which enabled us to modularize the code-path and insert the hook.

Special optimizations: Two common modifications to the metadata of a file system are: (1) the update of block pointers in an indirect block, and (2) flipping of bits in the bitmaps to indicate the allocated or freed status of blocks. We wrote custom, optimized checksum computation routines for those cases, i.e., the modification of a fixed-length block pointer (64 bits in the case of WAFL) and the modification of a single bitmap bit. For other updates, we created a generalized incremental checksum computation routine that is based on the offset (in the block) and on the length of the update so that it can handle variable length modification. An example use of such generalized computation is file deletion processing, which requires bulk updates of metadata blocks.

Complexity: Intercepting each modification to a metadata block was a bit intrusive, but it made the code more modular and fostered better development practices. On one hand, our implementation is more complex compared with Recon because we intercept each modification to a metadata block. But on the other hand it is simpler because we do not have to implement a cache and therefore avoid the many problems associated with caching.

7 Evaluation

In this section, we evaluate the performance and the benefits of the various mechanisms that we presented earlier.

We used an in-house workload generator that emulates random reads and writes to model the query and update operations of a database/OLTP application. It was built to be very similar to the Storage Performance Council Benchmark-1 (SPC-1) [20]. The Standard Performance Evaluation Corporation home-directory style benchmark (SPEC SFS) [19] was also used, but those results are not presented here because our protection mechanisms showed negligible overhead. The heavy random overwrites that the database/OLTP benchmark produces put a much higher stress on our mechanisms because more metadata is modified per transaction.

7.1 Incremental Checksum Performance

As explained earlier in Section 4.3 this mechanism uses the per-block persistent checksum that is already enforced and paid for by the WAFL I/O stack. The only additional work required is to compute the checksum incrementally when the metadata is modified in memory. The raw CPU cycles for computing and updating the checksum are negligible and likely are partially absorbed by idle cycles between processor pipeline stages. The memory overhead is 4 bytes per-processor (for lockless concurrent updates) for every 4KB metadata block, which is less than 0.1% of the total buffer cache memory. Furthermore, checksum computation and update does not directly affect the latency of a user operation because almost all metadata updates occur asynchronously after the user operation has been completed and acknowledged.

Figure 1(a) and Figure 1(b), respectively, show this cost on: (a) a midrange system with 12 Intel® Westmere cores, 96GB DRAM and 4GB NVRAM; and (b) a low-end system with 4 Intel Wolfdale cores, 20GB DRAM, and 2GB NVRAM. Sufficient numbers of SAS hard disks were attached to both systems to eliminate any storage bottleneck. Experiments were run by using our database/OLTP benchmark with incremental checksums turned on and turned off, and the observed latency was plotted against the achieved IOPS throughput with an increasing input IOPS load. We plotted only the load points with latencies that were less than or equal to 30ms, which is the maximum that the SPC-1 benchmark allows.

We see an increase in latency in the range of -0.9ms to 1.7ms and zero to 0.8ms on the midrange and low-end systems, respectively; these ranges translate to -3.5% to 10.5% and zero to 17%, respectively. Note that at very low latencies, a small increase in latency translates into a large percentage of change even though the absolute change is of little practical consequence. If we look at the achieved throughput at any given latency on the midrange system, we see from zero to a maximum of 1% regression. On the low-end system, the throughput regression varies from 5% (at 5ms) to zero (at 24ms).

File deletion is another workload that generates heavy metadata updates, and thereby stresses the incremental checksum mechanism. To process files that are pending deletion [37], WAFL must asynchronously walk several indirect blocks, clear block pointers in them, and update the relevant metadata in large quantities. Unfortunately, SPEC SFS generates an insufficient file deletion load. Similarly, the *SCSI_UNMAP* operation also causes heavy metadata updates, but is not generated in sufficient numbers by benchmarks such as SPC-1. Hence, we fashioned a custom benchmark that creates numerous very large files (a few terabytes' worth of space),

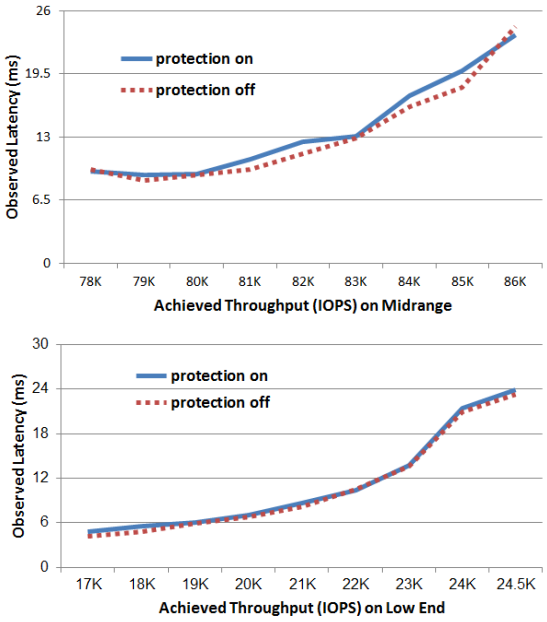


Figure 1: Latency versus throughput with and without incremental checksum protection with a database/OLTP workload on: (a) a midrange system with 12 cores, 96GB DRAM, and 4GB NVRAM; and (b) a low-end system with 4 cores, 20GB DRAM, and 2GB NVRAM.

then deletes them all and measures the delete rate that the system achieves. Although the achievable delete rate was pretty much the same (and there was no actual client latency to measure), we computed the overhead of incremental checksum protection as a function of the number of blocks freed. To compute this overhead, we added up cycles that were spent by the functions that update these pointers and the associated metadata. The benchmark was run on the previously mentioned midrange and low-end systems. On the midrange system, the WAFL CPU cost for freeing each block was computed to be around 2.16μs and 2.33μs with incremental checksums turned off and on, respectively. The corresponding numbers on the low-end system were 2.26μs and 2.43μs, respectively. This represents a 7.8% and 7.5% overhead, respectively. To put things in perspective, the CPU cycles that were spent across all WAFL code-paths during that interval were 3μs to 4μs per block freed. Therefore, this overhead is 2.5% as a fraction of the total WAFL CPU cycles.

7.2 Full Protection Performance

In this section, we measure the performance overhead of both auditing and incremental checksum working together. As explained earlier in Section 5.2 the CPU cost of maintaining audit digests is negligible, and the cost is minimized further by combining it with incremental

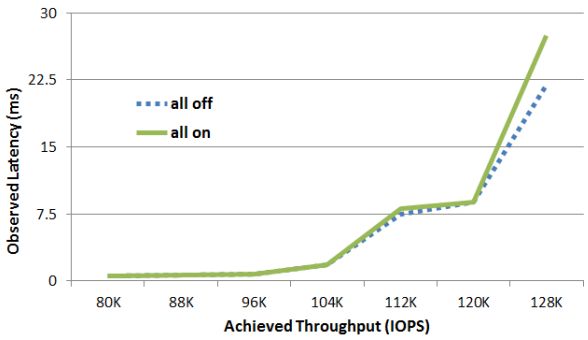


Figure 2: Latency versus throughput with and without all protection (audit and incremental checksum) with a database/OLTP workload on a high-end all-flash system with 20 cores, 128GB DRAM, and 8GB NVRAM.

checksum updates. The memory overhead is tiny as it requires just a few bytes of digest per auditing invariant.

Figure 2 studies the effect of protection mechanisms with the database/OLTP benchmark on a high-end all-flash system with 20 Intel Ivy Bridge cores, and 128GB of DRAM, and several shelves' worth of solid-state drives. Experiments were run with all protection turned off and on. The protection-on case includes incremental checksum and auditing with the 20+ audit equations that we have implemented till date. The observed latency was plotted against the achieved IOPS throughput with an increasing input IOPS load. We see that at up to 120,000 IOPS, there is absolutely no impact on latencies. At high loads of 128,000 IOPS and above, we see a rapid latency increase of about 5.5ms, or about 25%. Note that most customers choose to run their high-performance database/OLTP-style loads either at or below the 2ms latency mark, where we see absolutely zero impact on latencies.

We should point out that compared to the midrange system used in Figure 1 the system used in Figure 2 is a high-end system with CPU cycles to spare, next-generation Intel chipsets, larger processor caches, more DRAM, and solid-state drives.

7.3 Scribble Diagnostics

Page protection is turned off by default in the shipped product; it can be turned on when required for diagnostics. If a corruption bug is repeatedly hit on a system, the other two protection mechanisms prevent it from being written to persistent storage. However, the repeated file system restarts can be disruptive for the customer. Because the rogue code-path or thread has already run to completion by the time the scribble is detected, it typically takes about one person-month of senior developer time to find the root cause of an average memory scribble bug. However, if the customer is willing to incur a 20% performance penalty by turning on page protection, the root cause presents itself in the resultant core dump that the page fault generates.

In the past five years, and over millions of hours of total run-time across hundreds of thousands of deployed ONTAP systems, page protection has been needed only once. A customer system hit an incremental checksum protection panic every few hours, and even though the root cause had been narrowed down to some suspect code-paths, it had not been found. The customer turned on page protection, and the resultant core file helped find the bug, which was a buffer overflow in a rarely hit code-path.

During product development, page protection is turned on by default. It has proved to be invaluable not only in identifying corruption bugs early, but also in reducing bugs of unknown origin that might have resurfaced later in the field. Section 7.5 provides relevant statistics.

7.4 Ability to Detect Bugs

WAFL has a built-in command-line tool that injects corruptions into in-memory data structures. Scripts invoke this command with arguments that specify the file system ID, inode number, indirect block level, offset, and length, together with the corruption pattern. The tool loads the data structure into memory (if it isn't already present) and injects the corruption. Several test plans were built so that injected corruptions would be caught by the protection mechanisms before the superblock of the subsequent transaction commit was written out. In some cases, the corruption pattern flipped specific bits in the block bitmap such that the resultant checksum remained unchanged and it could slip through the incremental checksum protection. However, those cases were always caught by the audit.

7.5 Benefits

These mechanisms have proved invaluable in protecting customer systems from becoming corrupted. First, we looked at corruption bugs that hit customer systems over a four-year period before the protection mechanisms were in place. For each bug, we tracked the time from when the development team started looking at it to when the root cause was discovered, and we looked at the number and the expertise of the people involved in fixing the bugs. On average, it took about one person-month of very senior developer time to find the root cause of each bug.

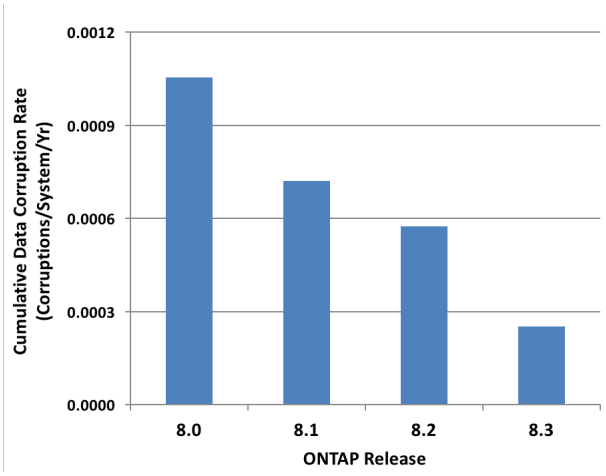


Figure 3: Number of times that inconsistencies made it to persistent storage on customer systems, normalized to total system run-time-hours across four different releases. Repair and recovery procedures were required. Because this data adds up different systems hitting the same bug, the number of actual bugs will be lower.

Since our protection mechanisms have been in place, most of the corruption bugs in WAFL, ONTAP, and hardware drivers have been caught well before customer deployment. Over a five-year period, a total of 75 memory scribble bugs have been found by page protection when testing with debug kernels, 32 memory scribble bugs were found in non-debug kernels by the incremental checksum mechanism, and 23 logic bugs were found by the auditing mechanism. We cannot overstate the value of these results in terms of organizational productivity.

Of course, as is true with any large system, some corruption bugs do manage to escape in-house testing. The first auditing equations shipped with NetApp Data ONTAP® 8.1 in September 2011; incremental checksum, page protection, and more auditing equations shipped with Data ONTAP 8.2; and even more equations shipped with Data ONTAP 8.3. In the past five years, the incremental checksum mechanism has protected customer systems from 8 unique memory scribble bugs 33 times, and the auditing mechanism has protected against 9 unique logic bugs 50 times. In total, that's 83 times that a customer system was saved from running expensive file system recovery procedures and from potential data loss.

Figure 3 shows the number of times that inconsistencies have made it to the persistent file system on customer systems, normalized to total system run-time-hours across the past four releases over a one-year period. The normalization by run-time-hours was necessary because, during that one-year period, the system hours that were recorded on Data ONTAP 8.1 and 8.2 were much higher than for the other two releases. Most systems had upgraded from the older 8.0 release, and not many systems had upgraded to 8.3 yet. Therefore, the raw data was biased toward the 8.1 and 8.2 releases. We see release-to-release improvements of 34%, 21%, and 44%, respectively. The total improvement amounts to a more than 3 times reduction in the rate of occurrence of inconsistencies. Some auxiliary WAFL metadata is not yet covered by the protection mechanisms. Therefore, we sometimes get benign inconsistencies in the persistent file system (not real data corruptions but inconsistencies nevertheless). We expect to see further reduction in inconsistencies once we fix these gaps.

8 Conclusion

We introduced two techniques, incremental checksum and digest-based auditing, that prevent in-memory scribbles and logic bugs from corrupting persistent file system metadata. We disproved the commonly held belief that strong data integrity requires a high performance penalty; we achieved integrity with a negligible performance tax. We distinguished scribbles from logic bugs, and also provided diagnostic capabilities to pinpoint the culprit for software scribbles.

These techniques have greatly improved data integrity in WAFL, resulting in an unprecedented reduction in recovery runs. By catching corruptions early in the development cycle, these techniques have enabled our engineers to innovate rapidly without risking data integrity.

We believe that end-to-end incremental checksumming can be applied to user data blocks, thereby providing round-trip application-level protection at a low cost. This technique can be especially useful in protecting applications that are hosted on third-party infrastructure, in which the reliability of hardware cannot be established or guaranteed. Moreover, continuous checksum protection can harden applications against induced corruption attacks on shared cloud infrastructure. Databases and file systems that are hosted on fabric-attached or cloud storage are good examples of potential benefiters of such end-to-end protection.

9 Acknowledgments

We thank the many WAFL developers who have contributed to the work presented in this paper: Ananthan Subramanian, Santosh Venugopal, Tijin George, Ganga Kondapalli, Mihir Gorecha, Varada Kari, Vishnu Vardhan, and Santhosh Paul. We also thank Remzi H. Arpaci-Dusseau, our reviewers, and our shepherd, Jiri Schindler, for their invaluable feedback.

References

[1] CERT/CC Advisories. http://www.cert.org/advisories/.

[2] Kernel Bug Tracker. http://bugzilla.kernel.org/.

[3] US-CERT Vulnerabilities Notes Database. http://www.kb.cert.org/vuls/.

[4] Computation of the internet checksum via incremental update. https://tools.ietf.org/html/rfc1624, 1994.

[5] Dave Anderson, Jim Dykes, and Erik Riedel. More Than an Interface: SCSI vs. ATA. In *FAST*, 2003.

[6] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS*, 2007.

[7] 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 *FAST*, 2008.

[8] Wendy Bartlett and Lisa Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Trans. on Dependable and Secure Computing*, 1(1), 2004.

[9] Robert Baumann. Soft errors in advanced computer systems. *IEEE Des. Test*, 22(3):258–266, 2005.

[10] M. P. Baze and S. P. Buchner. Attenuation of single event induced pulses in cmos combinational logic. *IEEE Transactions on Nuclear Science*, 44(6):2217–2223, Dec 1997.

[11] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[12] Steve Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.

[13] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.

[14] Wayne Burleson, Onur Mutlu, and Mohit Tiwari. Invited - who is the major threat to tomorrow's security?: You, the hardware designer. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 145:1–145:5, New York, NY, USA, 2016. ACM.

[15] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *SOSP*, 1995.

[16] C. L. Chen. Error-correcting codes for semiconductor memories. *SIGARCH Comput. Archit. News*, 12(3):245–247, 1984.

[17] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2004.

[18] Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual Volume 3A: System Programming Guide, Part 1. https://software.intel.com/sites/default/files/managed/7c/f1/253668-sdm-vol-3a.pdf, December 2016.

[19] Standard Performance Evaluation Corporation. Spec sfs 2014. https://www.spec.org/sfs2014/.

[20] Storage Performance Council. Storage performance council-1 benchmark. www.storageperformance.org/results/#spc1_overview.

[21] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceeding of Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[22] Fryer Daniel. Personal Communication. Date - 09/07/2016.

[23] P. Deutsch and J.-L. Gailly. Zlib compressed data format specification version 3.3. https://www.ietf.org/rfc/rfc1950.txt, 1996.

[24] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. FlexVol: flexible, efficient file volume virtualization in WAFL. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 129–142, Jun 2008.

[25] A. Eto, M. Hidaka, Y. Okuyama, K. Kimura, and M. Hosono. Impact of neutron flux on soft errors in mos memories. In *International Electron Devices Meeting*, 1998.

[26] J. Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, 1982.

[27] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *FAST '12*, San Jose, CA, February 2012.

[28] Gregory R. Ganger and Yale N. Patt. Metadata Up-

date Performance in File Systems. In *OSDI '94*, 1994.

[29] Jim Gray. Why do computers stop and what can be done about it? http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf, 1985.

[30] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87*, 1987.

[31] James Hamilton. Successfully Challenging the Server Tax. http://perspectives.mvdirona.com/2009/09/03/SuccessfullyChallengingTheServerTax.aspx.

[32] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter 1994 Technical Conference*, pages 235–246, Jan 1994.

[33] NetApp Inc. Netapp quick notes - wafl check. http://netapp-notes.blogspot.com/2008/03/wafl-check.html, 2008.

[34] NetApp Inc. Data ONTAP 8. http://www.netapp.com/us/products/platform-os/ontap/, 2010.

[35] NetApp Inc. Overview of wafliron. https://kb.netapp.com/support/index?page=content\&id=3011877, 2016.

[36] Dell T. J. A white paper on the benefits of chipkill-correct ecc for pc server main memory. *IBM Microelectronics Division*, 1997.

[37] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. Algorithms and data structures for efficient free space reclamation in WAFL. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2017.

[38] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372, Piscataway, NJ, USA, 2014. IEEE Press.

[39] Xin Li, Kai Shen, Michael C. Huang, and Lingkun Chu. A memory soft error measurement on production systems. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 21:1–21:6, Berkeley, CA, USA, 2007. USENIX Association.

[40] Joshua MacDonald, Hans Reiser, and Alex Zarochentcev. Reiser4 Transaction Design Document. https://reiser4.wiki.kernel.org/index.php/Txn-doc, 2002.

[41] T. Mallory and A. Kullberg. Incremental updating of the internet checksum. https://tools.ietf.org/html/rfc1141, 1990.

[42] Avantika Mathur, Mingming Cao, and Andreas Dilger. ext4: the next generation of the ext3 file system. *;login: - The Usenix Magazine*, Vol. 31, June 2006.

[43] Theresa C. Maxino and Philip J. Koopman. The effectiveness of checksums for embedded control networks. *IEEE Trans. Dependable Secur. Comput.*, 6(1):59–72, January 2009.

[44] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Trans. on Electron Dev*, 26(1), 1979.

[45] Thomas R. Nicely. Pentium fdiv bug. http://www.trnicely.net/pentbug/bugmail1.html.

[46] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM J. Res. Dev.*, 40(1):41–50, 1996.

[47] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD*, 1988.

[48] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1–18, Austin, TX, August 2016. USENIX Association.

[49] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[50] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1), 1992.

[51] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *FAST*, 2007.

[52] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A Large-Scale Field Study. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, Seattle, Washington, June 2007.

[53] Thomas J. E. Schwarz, Qin Xin, Ethan L. Miller, Darrell D. E. Long, Andy Hospodor, and Spencer Ng. Disk scrubbing in large archival storage systems. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MAS-

COTS '04, pages 409–418, Washington, DC, USA, 2004. IEEE Computer Society.

[54] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 73–88, Berkeley, CA, USA, 2003. USENIX Association.

[55] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 297–310, New York, NY, USA, 2015. ACM.

[56] Inquirer Staff. Amd opteron bug can cause incorrect results. `http://www.theinquirer.net`.

[57] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying file system protection. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 79–90, Berkeley, CA, USA, 2001. USENIX Association.

[58] The Data Clinic. Hard Disk Failure. `http://www.dataclinic.co.uk/hard-disk-failures.htm`, 2004.

[59] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, 1998.

[60] Stephen C. Tweedie. EXT3, Journaling File System. `olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`, 2000.

[61] Glenn Weinberg. The Solaris Dynamic File System. `http://members.visi.net/$\sim$thedave/sun/DynFS.pdf`, 2004.

[62] Yichen Xie, Andy Chou, and Dawson Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 327–336, New York, NY, USA, 2003. ACM.

[63] Yupu Zhang, Daniel Myers, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Zettabyte Reliability with Flexible End-to-end Data Integrity. In *MSST'13*, Long Beach, CA, May 2013.

[64] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: A zfs case study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, Berkeley, CA, USA, 2010. USENIX Association.

[65] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.

NETAPP, the NETAPP logo, and the marks listed at *http://www.netapp.com/TM* are trademarks of NetApp, Inc.

Mirador: An Active Control Plane for Datacenter Storage

Jake Wires and Andrew Warfield
Coho Data

Abstract

This paper describes *Mirador*, a dynamic placement service implemented as part of an enterprise scale-out storage product. Mirador is able to encode multi-dimensional placement goals relating to the performance, failure response, and workload adaptation of the storage system. Using approaches from dynamic constraint satisfaction, Mirador migrates both data and client network connections in order to continuously adapt and improve the configuration of the storage system.

1 Introduction

In becoming an active resource within the datacenter, storage is now similar to the compute and network resources to which it attaches. For those resources, recent years have seen a reorganization of software stacks to cleanly disentangle the notions of control and data paths. This thrust toward "software defined" systems aims for designs in which virtualized resources may be provisioned on demand and in which central control logic allows the programmatic management of resource placement in support of scale, efficiency, and performance.

This paper observes that modern storage systems both warrant and demand exactly this approach to design. The emergence of high-performance rack-scale hardware [10, 17, 40] is amplifying the importance of *connectivity* between application workloads and their data as a critical aspect of efficient datacenter design. Fortunately, the resource programmability introduced by software defined networks and the low cost of data migration on non-volatile memory means that the dynamic reconfiguration of a storage system is achievable.

How is dynamic placement useful in the context of storage? First, consider that network topology has become a very significant factor in distributed storage designs. Driven by the fact that intra-rack bandwidth continues to outpace east/west links and that storage device latencies are approaching that of Ethernet round-trip times, efficient storage placement should ensure that data is placed in the same rack as the workloads that access it, and that network load is actively balanced across physical links.

A separate goal of distributing replicas across isolated failure domains requires a similar understanding of physical and network topology, but may act in opposition to the goal of performance and efficiency mentioned above. While placement goals such as these examples can be motivated and described in relatively simple terms, the resulting placement problem is multi-dimensional and continuously changing, and so very challenging to solve.

Mirador is a dynamic storage placement service that addresses exactly this problem. Built as a component within a scale-out enterprise storage product [12], Mirador's role is to translate configuration *intention* as specified by a set of *objective functions* into appropriate placement decisions that continuously optimize for performance, efficiency, and safety. The broader storage system that Mirador controls is capable of dynamically migrating both the placement of individual chunks of data and the client network connections that are used to access them. Mirador borrows techniques from dynamic constraint satisfaction to allow multi-dimensional goals to be expressed and satisfied dynamically in response to evolutions in environment, scale, and workloads.

This paper describes our experience in designing and building Mirador, which is the second full version of a placement service we have built. Our contributions are threefold: We demonstrate that robust placement policies can be defined as simple declarative objective functions and that general-purpose solvers can be used to find solutions that apply these constraints to both network traffic and data placement in a production storage system, advancing the application of optimization techniques to the storage configuration problem [1, 6–8, 49]. We show that for performance-dense storage clusters, placement decisions informed by the relative capabilities of net-

work and storage tiers can yield improvements over more static layouts originally developed for large collections of disks. And finally, we investigate techniques for exploiting longitudinal workload profiling to craft custom placement policies that lead to additional improvements in performance and cost-efficiency.

2 A Control Plane for Datacenter Storage

Mirador implements the control plane of a scale-out enterprise storage system which presents network-attached block devices for use by virtual machines (VMs), much like Amazon's Elastic Block Store [11]. A typical deployment consists of one or more independent storage nodes populated with performance-dense NVMe devices, each capable of sustaining random-access throughputs of hundreds of thousands of IOPS. In order to capitalize on the low latency of these devices, storage nodes are commonly embedded horizontally throughout the datacenter alongside the compute nodes they serve. In this environment, Mirador's role is to provide a centralized placement service that continuously monitors the storage system and coordinates the migration of both data and network connections in response to workload and environmental changes.

A guiding design principle of Mirador is that placement decisions should be ***dynamic*** and ***flexible***.

Dynamic placement decisions allow the system to adapt to environmental change. We regularly observe deployments of hundreds to thousands of VMs where only a small number of workloads dominate resource consumption across the cluster at any given time. Moreover, the membership of this set often changes as VMs are created and deleted or they transition through different workload phases. For these reasons, the initial choices made when placing data in the cluster may not always be the best ones; significant improvements can often be had by periodically re-evaluating placement decisions over time in response to changes in workload behavior.

Flexible placement decisions allow the system to articulate complex and multidimensional policy. Rather than trying to combine diverse and often conflicting goals in a single monolithic description, Mirador approaches system configuration as a search problem. Policies are composed of one or more *objective functions*, simple rules that express how resources should be allocated by computing numerical costs for specific configurations. A planning engine employs established constraint satisfaction techniques to efficiently search the configuration space for a minimal-cost solution.

In our experience, policies expressed as simple independent rules are substantially more perspicuous and robust than their monolithic alternatives. For example, after upgrading the customized planning engine that shipped in an early version of the product to a generic constraint solver, we were able to replace a load balancing policy originally defined in 2,000 lines of imperative Python with a similar policy composed of seven simple rules each expressed in less than thirty lines of code (see § 3.2.1 for examples). Much of the complexity of the original policy came from describing *how* it should be realized rather than *what* it intended to achieve. By disentangling these two questions and answering the former with a generic search algorithm, we arrived at a policy description that is equally efficient as the first version, yet much easier to reason about and maintain.

Mirador implements the configuration changes recommended by the planning engine by coordinating a cluster-wide schedule of data and network migration tasks, taking care to minimize the performance impact on client workloads. It communicates directly with switches and storage nodes to effect these migrations, continually monitoring system performance as it does so. In this way it actively responds to environmental and workload changes and results in a more responsive, robust system.

3 Mirador

Mirador is a highly-available data placement service that is part of a commercial scale-out storage product. Figure 1 presents a typical cluster composed of multiple storage *nodes*. Each node is a regular server populated with one or more directly-attached, non-volatile storage *devices*. Nodes implement an object interface on top of these devices and manage virtual to physical address translations internally. Objects present sparse 63-bit address spaces and are the primary unit of placement. A virtual block device interface is presented to clients. Virtual devices may be composed of one or more objects distributed across multiple nodes; by default, they are striped across 16 objects, resulting in typical object sizes on the order of tens to hundreds of GiB.

The storage cluster is fronted by a set of Software Defined Network (SDN) switches that export the cluster over a single virtual IP address. Clients connect to the virtual IP and are directed to storage nodes by a custom SDN controller. Nodes are connected in a mesh topology, and any node is capable of servicing requests from any client, allowing the mapping between clients and nodes to be modified arbitrarily.

One or more nodes in the cluster participate as a Mirador service provider. Service providers work together to monitor the state of the cluster and initiate *rebalance jobs* in response to topology and load changes. Rebalance jobs are structured as a control pipeline that generates and executes plans for dynamically reconfiguring

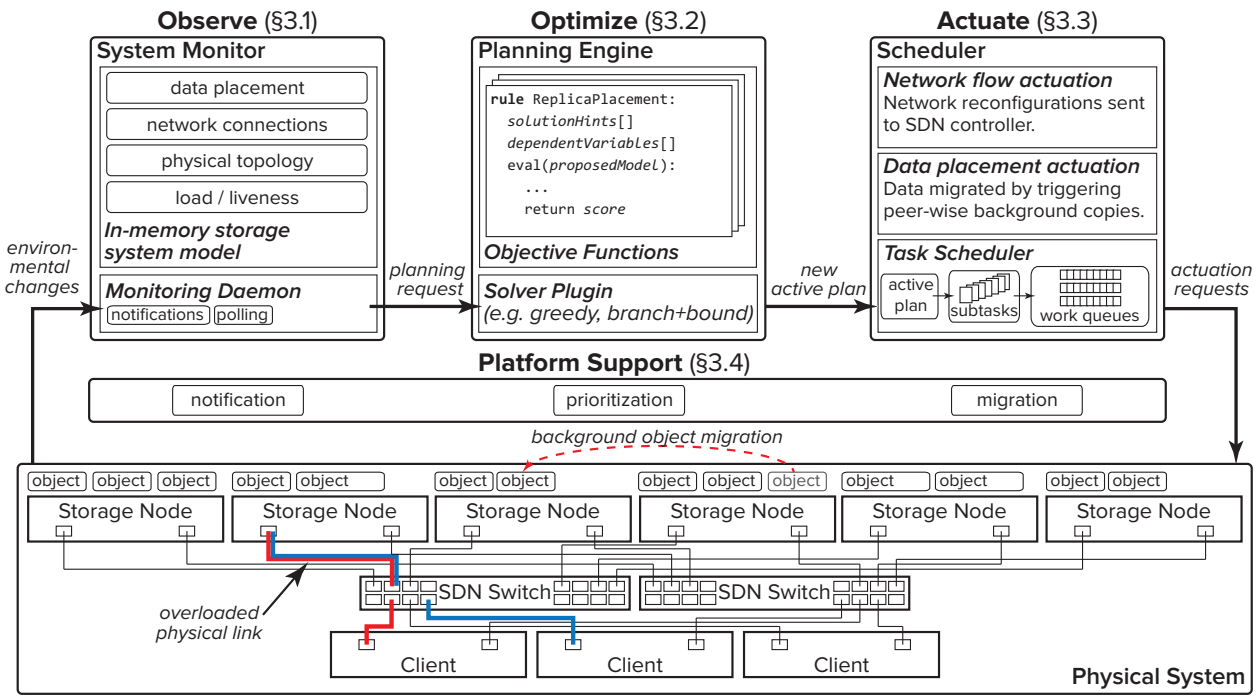


Figure 1: The storage system architecture (below) and the Mirador rebalance pipeline (above). The figure shows two examples of the system performing actuations in response to observed state. First, the fourth storage node has become disproportionately full relative to the other nodes. To balance capacity in the system, the rightmost object on that node is undergoing background migration to the third node. Second, the physical network link into the left side port of the second storage node has come under pressure from two high-volume flows from the first two clients. The system will observe this overload, and then chose one of the flows to migrate to a different physical link.

the placement of data and client connections in order to optimize for performance, efficiency, and safety. Job state is periodically checkpointed in a replicated state machine [28], providing strong resiliency against failures.

The rebalance pipeline is composed of three stages:

Observation A *system monitor* collects resource metrics like device and network load along with detailed workload profiles to construct a model of the cluster.

Optimization A *planning engine* computes a numerical cost for the current configuration and searches for alternative configurations that would reduce or eliminate this cost. If a lower-cost arrangement is identified, a plan is constructed that yields the desired results.

Actuation A *scheduler* implements the plan by coordinating the migration of data and client connections.

3.1 Observation

The system monitor maintains a *storage system model* that captures all relevant properties of the physical system, including static features like cluster topology (e.g., the number of devices and nodes, the capacity of their network links, and user-defined failure domains) and dynamic features like the current free space and IO load of devices and the utilization of network ports.

The monitor also collects highly-compressed sketches of individual workload behavior [55]. These summaries are collected by a dedicated *workload analysis* service, and they include features such as *miss ratio curves* and *windowed footprints*. Unlike hardware utilization levels, this data cannot be computed from instantaneous measurements, but instead requires detailed profiling of workloads over extended periods of time.

The monitor synchronizes the model by polling the cluster; sampling frequencies vary from every few seconds for metrics like link load to tens of minutes for workload footprint measurements, while exceptional events such as device failures are signalled via special alerts.

3.2 Optimization

The planning engine implements the logic responsible for generating rebalance plans. Placement logic is encapsulated in one or more *objective functions* that specify

rules for how data and flows should be distributed across the cluster. The engine invokes a *solver* to search for new configurations that reduce placement costs, as defined by the objective functions.

The planning engine manipulates a copy of the storage model when considering alternative configurations. For example, if a decision is made to move an object from one device to another, the modelled free space and load of each device is adjusted to reflect the change.

Modelling data migration within the cluster is a challenging problem. While an object's size serves as a rough approximation of the cost of migrating it, the actual time required to move the data depends on many things, including the type and load of the source and destination devices, network contention along the migration path, and fragmentation of the data being migrated. This is important, however, because system resources like free space and bandwidth may be consumed at both the source and destination devices during migration, and the solver may make poor decisions if this usage is modelled incorrectly. For this reason, migrations initiated during the optimization stage are modelled conservatively by *reserving* space on the destination device at the beginning of operation and only releasing it from the source device once the migration has completed.

3.2.1 Objective Functions

Data placement is expressed as an optimization problem by representing objects and flows as variables and devices and links as the values these variables can take, respectively. Within this framework, objective functions model the cost (or benefit) of assigning a value to a given variable (e.g., placing a replica on a specific device). [1]

Mirador objective functions can assign arbitrary numerical costs to a given configuration. Hard constraints, implemented by rules imposing an infinite cost, can never be violated – any configuration with an infinite cost is rejected outright. Negative costs can also be used to express *affinities* for preferred assignments. An *optimal* configuration is one that minimizes the cumulative cost of all assignments; solvers employ various search strategies to find minimal-cost solutions. In the case that no finite-cost configuration can be found (e.g., due to catastrophic hardware failure), Mirador raises an alert that manual intervention is required.

Objective functions are expressed as simple Python functions operating on the storage system model described above. Listing 1 shows a rule designed to minimize load imbalances by stipulating that the spread between the most- and least-loaded devices falls within a given range. (Note that this formulation codifies a system-level notion of balance by assigning costs to *all* objects located on overloaded devices; moving just one such object to a different device may be enough to eliminate the cost for all the remaining objects.) During the optimization stage, the planning engine converts the storage model into an abstract representation of variables, values, and objectives, and computes the cost of each assignment by invoking its associated rules (see § 3.2.2).

A special annotation specifies the *scope* of the rule, indicating which components it affects (e.g., objects, connections, devices, links). Solvers refer to these annotations when determining which rules need to be re-evaluated during configuration changes. For example, the `load_balanced` rule affects `devices`, and must be invoked whenever the contents of a device changes.

Mutual objectives can be defined over multiple related objects. For instance, Listing 2 gives the implementation of a rule stipulating that no two objects in a replica set reside on the same device; it could easily be extended to include broader knowledge of rack and warehouse topology as well. Whenever a solver assigns a new value to a variable affected by a mutual objective, it must also re-evaluate all related variables (e.g., all other replicas in the replica set), as their costs may have changed as a consequence of the reassignment.

Rules can provide hints to the solver to help prune the search space. Rule implementations accept a *domain* argument, which gives a dictionary of the values that can be assigned to the variable under consideration, and is initially empty. Rules are free to update this dictionary with the expected cost that would be incurred by assigning a particular value. For example, the rule in Listing 2 populates a given replica's domain with the precomputed cost of moving it onto *any* device already hosting one of its copies, thereby deprioritizing these devices during the search. The intuition behind this optimization is that most rules in the system only affect a small subset of the possible values a variable can take, and consequently, a handful of carefully chosen hints can efficiently prune a large portion of the solution space.

A policy consists of one or more rules, which can be restricted to specific hardware components or object groups in support of multi-tenant deployments.

3.2.2 Solvers

The planning engine is written in a modular way, making it easy to implement multiple solvers with different search strategies. Solvers accept three arguments: a dictionary of *assignments* mapping variables to their current values, a dictionary of *domains* mapping variables to all possible values they can take, and a dictionary of

[1] For clarity of exposition, we use the terms *objective function* and *rule* interchangably throughout the paper.

```
@rule(model.Device)
def load_balanced(fs, device, domain):
   cost, penalty = 0, DEVICE_BALANCED_COST
   # compute load of current device
   # for the current sample interval
   load = device.load()
   # compute load of least-loaded device
   minload = fs.mindevice().load()
   if load - minload > LOAD_SPREAD:
      # if the difference is too large,
      # the current device is overloaded
      cost = penalty
   return cost
```

Listing 1: Load Balancing Rule

```
@rule(model.ReplicaSet)
def rplset_devices_unique(fs, replica, domain):
   cost, penalty = 0, INFINITY
   for rpl in replica.rplset:
      if rpl is replica:
         # skip current replica
         continue
      if rpl.device is replica.device:
         # two replicas on the same device
         # violate redundancy constraint
         cost = penalty
      # provide a hint to the solver that the
      # devices already hosting this replica set
      # are poor candidates for this replica.
      domain[rpl.device] += penalty
   return cost
```

Listing 2: Hardware Redundancy Rule

objectives mapping variables to the rules they must satisfy. Newly-added variables may have no assignment to start with, indicating that they have not yet been placed in the system. Solvers generate a sequence of *solutions*, dictionaries mapping variables to their new values. The planning engine iterates through this sequence of solutions until it finds one with an acceptable cost, or no more solutions can be found.

Mirador provides a pluggable solver interface that abstracts all knowledge of the storage model described abover. Solvers implement generic search algorithms and are free to employ standard optimization techniques like forward checking [24] and constraint propagation [36] to improve performance and solution quality.

We initially experimented with a branch and bound solver [44] because at first glance it fits well with our typical use case of soft constraints in a dense solution space [19]. A key challenge to using backtracking algorithms for data placement, however, is that these algorithms frequently yield solutions that are very different from their initial assignments. Because reassigning variables in this context may imply migrating a large amount of data from one device to another, this property can be quite onerous in practice. One way to address this is to add a rule whose cost is proportional to the difference between the solution and its initial assignment (as measured, for example, by its Hamming distance) [25]. However, this technique precludes zero-cost reconfigurations (since every reassignment incurs a cost) and thus requires careful tuning when determining whether a solution with an acceptable cost has been found.

We eventually adopted a simpler greedy algorithm. While it is not guaranteed to identify optimal solutions in every case, we find in practice that it yields quality solutions with fewer reassignments and a much more predictable run time. In fact, the greedy algorithm has been shown to be a 2-approximate solution for the related makespan problem [22], and it is a natural fit for load rebalancing as well [3].

Listing 3 presents a simplified implementation of the greedy solver. It maintains a priority queue of variables that are currently violating rules, ordered by the cost of the violations, and a priority-ordered domain for each variable specifying the possible values it can take. A pluggable module updates domain priorities in response to variable reassignments, making it possible to model capacity and load changes as the solver permutes the system searching for a solution. The current implementation prioritizes values according to various utilization metrics, including free space and load.

As described in § 3.2.1, objective functions can provide hints to the solver about potential assignments. The greedy algorithm uses these hints to augment the priority order defined by the storage system model, so that values that would violate rules are deprioritized. The search is performed in a single pass over all variables, starting with the highest-cost variables. First the rules for the variable are invoked to determine whether any values in its domain violate the prescribed placement objectives (or alternatively, satisfy placement affinities). If the rules identify a zero or negative-cost assignment, this is chosen. Otherwise, the highest-priority unconstrained value is selected from the variable's domain. The search yields its solution once all violations have been resolved or all variables have been evaluated.

Besides its predictable run time, the greedy algorithm generally yields low migration overheads, since only variables that are violating rules are considered for reassignment. However, if the initial assignments are poor, the algorithm can get trapped in local minima and fail to find a zero-cost solution. In this case, a second pass clears the assignment of a group of the costliest variables collectively, providing more freedom for the solver, but potentially incurring higher migration costs. We find that this second pass is rarely necessary given the typically under-constrained policies we use in production and is limited almost exclusively to unit tests that intentionally stress the planning engine (see § 5 for more details).

```
def greedy(assignments, domains, objectives):
  # rank variables according to cost
  queue = PriorityQueue(domains)

  while queue.cost() > 0:
    # select the highest-cost variable
    val = None
    var = queue.pop()
    cur = assignments.get(var)
    domain = domains[var]

    # retrieve the variable's current cost
    # and any domain hints provided by the
    # rules
    cost, hints = score(var, cur, objectives)
    if cost <=0:
      # current assignment is good
      continue

    if hints:
      # find the lowest-cost hint. NB: we
      # assume that typically, most values
      # are unconstrained, so this linear
      # scan adds a small constant overhead.
      try:
        val = min(
          v for v in hints
          if v in domain and v != cur
        )
      except ValueError:
        pass

    if val is None or hints[val] > 0:
      # if we have no hints, or the best
      # hints are costly, choose the lowest-
      # cost unconstrained value in the domain
      val = next(
        (
          v for v in domain
          if v not in hints and v != cur
        ),
        val
      )

    if val is None:
      # couldn't find a value
      c = infinity
    else:
      # compute cost of new value
      c, _ = score(var, val, objectives)

    if c >= cost:
      # no benefit to re-assigning
      continue

    # found a better assignment
    assignments[var] = val

    # recompute the cost of any mutually-
    # constrained variables that haven't
    # already been evaluated
    for v in rulemap(var, objectives):
      if v in queue:
        queue.reschedule(v)

  # we've arrived at a solution
  return assignments
```

Listing 3: Greedy Solver

3.3 Actuation

Mirador can migrate both data and client connections. The scheduler models the cost of data migration conservatively, and attempts to minimize the impact of such migrations on client performance whenever possible. Connection migrations are generally cheaper to perform and as such occur much more frequently – on the order of minutes rather than hours.

Optimally scheduling data migration tasks is NP-hard [31–33]; Mirador implements a simple global scheduler that parallelizes migrations as much as possible without overloading individual devices or links.

Data migrations are performed in two steps: first, a background task copies an object to the destination device, and then, only after the object is fully replicated at the destination, it is removed from the source. This ensures that the durability of the object is never compromised during migration. Client connections are migrated using standard SDN routing APIs augmented by custom protocol handlers that facilitate session state handover.

3.4 Platform Support

Mirador executes rebalance jobs in batches by (1) selecting a group of objects and/or client connections to inspect, (2) invoking the planning engine to search for alternative configurations for these entities, and (3) coordinating the migration tasks required to achieve the new layout. Batches can overlap, allowing parallelism across the three stages. Mirador attempts to prioritize the worst offenders in early batches in order to minimize actuation costs, but it guarantees that every object is processed at least once during every job.

Mirador is able to perform its job efficiently thanks to three unique features provided by the storage platform. First, the system monitor relies on a *notification* facility provided by the cluster metadata service to quickly identify objects that have been recently created or modified. This allows nodes in the cluster to make quick, conservative placement decisions on the data path while making it easy for Mirador to inspect and modify these decisions in a timely manner, providing a strong decoupling of data and control paths. Second, the planning engine makes use of a *prioritization* interface implemented at each node that accepts a metric identifier as an argument (e.g., network or disk throughput, storage IOPS or capacity) and returns a list of the busiest workloads currently being serviced by the node. Mirador can use this to inspect problematic offenders first when attempting to minimize specific objective functions (such as load balancing and capacity constraints) rather than inspecting objects in arbitrary order. Finally, the actuation scheduler implements plans with the help of a *migration* rou-

tine that performs optimized background copies of objects across nodes and supports online reconfiguration of object metadata. This interface also provides hooks to the network controller to migrate connections and session state across nodes.

4 Evaluation

In this section we explore both the expressive power of Mirador policies and the impact such policies can have on real storage workloads. Table 1 lists the rules featured in this section; some have been used in production deployments for over a year, while others are presented to demonstrate the breadth and variety of placement strategies enabled by Mirador.

§ 4.1 measures the performance and scalability of the planning engine, independent of storage hardware. § 4.2 shows how Mirador performs in representative enterprise configurations; storage nodes in this section are equipped with 12 1 TB SSDs, two 10 gigabit Ethernet ports, 64 GiB of RAM, and 2 Xeon E5-2620 processors at 2 GHz with 6 cores each and hyperthreading enabled. § 4.3 and § 4.4 highlight the flexibility of rule-based policies, as measured on a smaller development cluster where 2 800 GB Intel 910 PCIe flash cards replace the 12 SSDs on each node.

Client workloads run in virtual machines hosted on four Dell PowerEdge r420 boxes running VMware ESXi 6.0, each with two 10 gigabit Ethernet ports, 64 GiB of RAM, and 2 Xeon ES-2470 processors at 2.3 GHz with 8 cores and hyperthreading enabled. Clients connect to storage nodes using NFSv3 via a dedicated 48-port SDN-controlled Arista 7050Tx switch, and VM disk images are striped across sixteen objects.

4.1 Optimization

We begin by benchmarking the greedy solver, which is used in all subsequent experiments. Given rules that run in constant time, this solver has a computational complexity of $O(N \log N \log M)$ for a system with N objects and M devices.

We measure solver runtime when enforcing a simple load-balancing policy (based on the `device_has_space` and `load_balanced` rules, with the latter enforcing a `LOAD_SPREAD` of 20%) in deployments of various sizes. In each experiment, a simulated cluster is modelled with fixed-capacity devices (no more than ten per node) randomly populated with objects whose sizes and loads are drawn from a Pareto distribution, scaled such that no single object exceeds the capacity of a device and the cluster is roughly 65% full. For each configuration we present the time required to find a zero-cost solution as well as the number of reconfigurations required to achieve the solution, averaged over ten runs. Some experiments require no reconfigurations because their high object-to-device ratios result in very small objects that yield well-balanced load distributions under the initial, uniformly random placement; the runtimes for these experiments measure only the time required to validate the initial configuration.

As Table 2 shows, the flexibility provided by Python-based rules comes with a downside of relatively high execution times (more than a minute for a system with 100K objects and 1K devices). While we believe there is ample opportunity to improve our unoptimized implementation, we have not yet done so, primarily because rebalance jobs run in overlapping batches, allowing optimization and actuation tasks to execute in parallel, and actuation times typically dominate.

4.2 Actuation

In the following experiment we measure actuation performance by demonstrating how Mirador restores redundancy in the face of hardware failures. We provision four nodes, each with 12 1 TB SSDs, for a total of 48 devices. We deploy 1,500 client VMs, each running `fio` [18] with a configuration modelled after virtual desktop workloads. VMs issue 4 KiB requests against 1 GiB disks. Requests are drawn from an 80/20 Pareto distribution with an 80:20 read:write ratio; read and write throughputs are rate-limited to 192 KiB/sec and 48 KiB/sec, respectively, with a maximum queue depth of 4, generating an aggregate throughput of roughly 100K IOPS.

Five minutes into the experiment, we take a device offline and schedule a rebalance job. The `rplset_durable` rule assigns infinite cost to objects placed on failed devices, forcing reconfigurations, while load-balancing and failure-domain rules prioritize the choice of replacement devices. The job defers actuation until a 15 minute stabilization interval expires so that transient errors do not trigger unnecessary migrations. During this time it inspects more than 118,000 objects, and it eventually rebuilds 3053 in just under 20 minutes, with negligible effect on client workloads, as seen in Figure 2.

4.3 Resource Objectives

We now shift our attention to the efficacy of specific placement rules, measuring the degree to which they can affect client performance in live systems. We first focus on resource-centric placement rules that leverage knowledge of cluster topology and client configurations to improve performance and simplify lifecycle operations.

Name	Objective	Cost	Lines of Code
device_has_space	devices are not filled beyond capacity	∞	4
rplset_durable	replica sets are adequately replicated on healthy devices	∞	4
load_balanced	load is balanced across devices	70	13
links_balanced	load is balanced across links	20	13
node_local	client files are co-located on common nodes	60	30
direct_connect	client connections are routed directly to their most-frequently accessed nodes	10	14
wss_best_fit	active working set sizes do not exceed flash capacities	40	4
isolated	cache-unfriendly workloads are co-located	20	30
co_scheduled	competing periodic workloads are isolated	20	35

Table 1: Objective functions used in evaluation section; *cost* gives the penalty incurred for violating the rule.

Objects	Devices	Reconfigurations	Time (seconds)
1K	10	6.40 ± 2.72	0.40 ± 0.06
1K	100	145.50 ± 33.23	0.83 ± 0.08
1K	1000	220.00 ± 12.53	10.11 ± 0.49
10K	10	0.00 ± 0.00	1.61 ± 0.01
10K	100	55.70 ± 5.46	5.54 ± 0.37
10K	1000	1475.00 ± 69.70	16.71 ± 0.88
100K	10	0.00 ± 0.00	17.10 ± 0.37
100K	100	9.30 ± 4.62	22.37 ± 5.38
100K	1000	573.80 ± 22.44	77.21 ± 2.87

Table 2: Greedy solver runtime for various deployment sizes with a basic load-balancing policy; *reconfigurations* gives the number of changes made to yield a zero-cost solution.

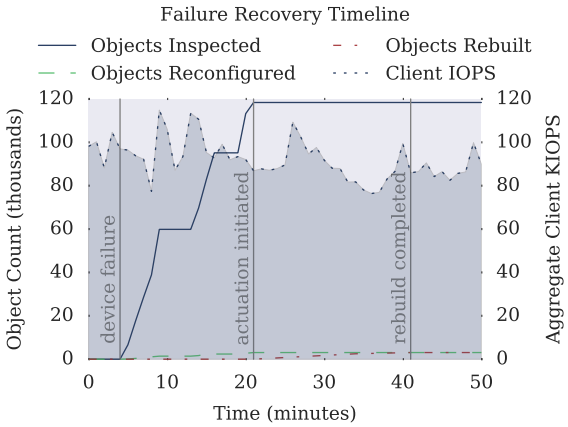


Figure 2: Rebuilding replicas after a device failure.

4.3.1 Topology-Aware Placement

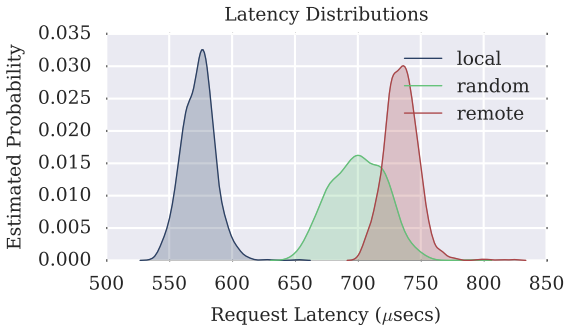
In this experiment we measure the value of topology-aware placement policies in distributed systems. We deploy four storage nodes and four clients, with each client hosting 8 VMs running a `fio` workload issuing random 4 KiB reads against dedicated 2 GiB virtual disks at queue depths ranging between 1 and 32.

Figure 3a presents the application-perceived latency achieved under three different placement policies when VMs issue requests at a queue depth of one. The *random* policy distributes stripes across backend devices using a simple consistent hashing scheme and applies a random one-to-one mapping from clients to storage nodes. This results in a configuration where each node serves requests from exactly one client, and with four nodes, roughly 75% of reads access remotely-hosted stripes. This topology-agnostic strategy is simple to implement, and, assuming workload uniformity, can be expected to achieve even utilization across the cluster, although it does require significant backend network communication. Indeed, as the number of storage nodes in a cluster increases, the likelihood that any node is able to serve requests locally decreases; in the limit, all requests require a backend RTT. This behavior is captured by the *remote* policy, which places stripes such that no node has a local copy of any of the data belonging to the clients it serves. The *local* policy follows the opposite strategy, placing all stripes for a given VM on a single node and ensuring that clients connect directly to the nodes hosting their data. Notably, all three policies are implemented in less than twenty lines of code, demonstrating the expressiveness of Mirador's optimization framework.

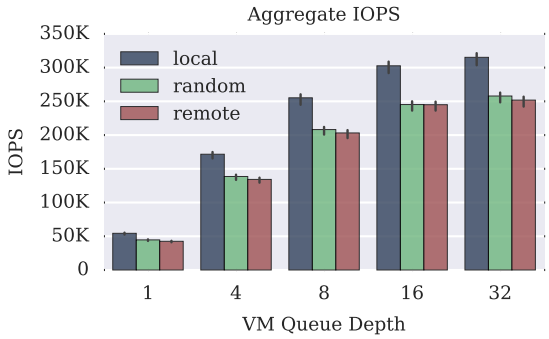
By co-locating VM stripes and intelligently routing client connections, the local policy eliminates additional backend RTTs and yields appreciable performance improvements, with median latencies 18% and 22% lower than those of the random and remote policies, respectively. Similar reductions are obtained across all measured queue depths, leading to comparable increases in throughput, as shown in Figure 3b.

4.3.2 Elastic Scale Out

In addition to improving application-perceived performance, minimizing cross-node communication enables linear scale out across nodes. While a random placement policy would incur proportionally more network RTTs as a cluster grows in size (potentially consuming oversubscribed cross-rack bandwidth), local placement strate-



(a) Latency distributions at queue depth of 1



(b) Mean throughput at various queue depths

Figure 3: Performance under three different placement strategies. The *local* policy yields a median latency 18% and 22% lower than the *random* and *remote* policies, respectively, resulting in an average throughput increase of 26%. (Error bars in Figure 3b give 95% confidence intervals.)

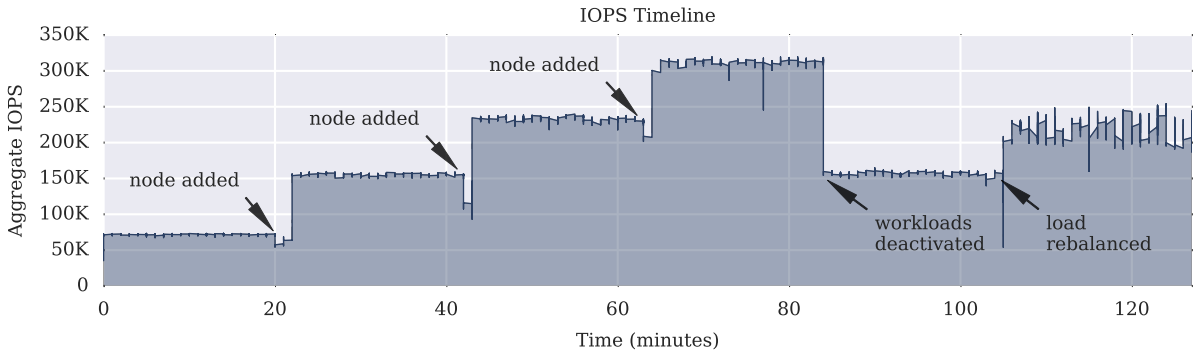


Figure 4: Mirador responds to changes in cluster topology and workload behavior. Data is immediately migrated to new storage nodes as they are introduced in 20 minute increments, starting at time t_{20}; the brief throughput drops are due to competition with background data copies. At time t_{85}, two of the four client machines are deactivated; the remaining client load is subsequently redistributed, at which point performance is limited by client resources.

gies can make full use of new hardware with minimal communication overhead. This is illustrated in Figure 4, which presents a timeline of aggregate client IOPS as storage nodes are added to a cluster. At time t_0 the cluster is configured with a single storage node serving four clients, each hosting 16 VMs issuing random 4 KiB reads at a queue depth of 32; performance is initially bottlenecked by the limited storage. At time t_{20}, an additional node is introduced, and the placement service automatically rebalances the data and client connections to make use of it. It takes just over two minutes to move roughly half the data in the cluster onto the new node. This migration is performed as a low-priority background task to limit interference with client IO. Two additional nodes are added at twenty minute intervals, and in each case, after a brief dip in client performance caused by competing migration traffic, throughput increases linearly.

The performance and scalability benefits of the local policy are appealing, but to be practical, this approach requires a truly dynamic placement service. While both local and random policies are susceptible to utilization imbalances caused by non-uniform workload patterns (e.g., workload 'hot spots'), the problem is exacerbated in the local case. For example, if all workloads placed on a specific node happen to become idle at the same time, that node will be underutilized. Figure 4 shows exactly this scenario at time t_{85}, where two clients are deactivated and the nodes serving them sit idle, halving overall throughput. After waiting for workload behavior to stabilize, the placement service responds to this imbalance by migrating some of the remaining VMs onto the idle storage, at which point the clients become the bottleneck.

4.4 Workload Objectives

Placement policies informed by resource monitoring can provide significant improvements in performance and efficiency, but they are somewhat *reactive* in the sense that they must constantly try to 'catch up' to changes in work-

load behavior. In this section we introduce and evaluate several techniques for improving data placement based on longitudinal observations of workload behavior.

The following examples are motivated by an analysis of hundreds of thousands of workload profiles collected from production deployments over the course of more than a year. The synthetic workloads evaluated here, while relatively simple, reflect some of the broad patterns we observe in these real-world profiles.

For these experiments, we extend the storage configuration described in § 4.3 with a disk-based capacity tier. The placement service controls how objects are assigned to flash devices as before; nodes manage the flash cards as LRU caches and page objects to disk in 512 KiB blocks. We artificially reduce the capacity of each flash device to 4 GiB to stress the tiering subsystem. While our evaluation focuses on conventional tiered storage, we note that the techniques presented here are applicable to a wide variety of hierarchical and NUMA architectures in which expensive, high-performance memories are combined with cheaper, more capacious alternatives, possibly connected by throughput-limited networks.

4.4.1 Footprint-Aware Placement

Many real-world workloads feature working sets (roughly defined as the set of data that is frequently accessed over a given period of time) that are much smaller than their total data sets [13, 56]. Policies that make decisions based only on knowledge of the latter may lead to suboptimal configurations. We show how augmenting traditional capacity rules with knowledge of working set sizes can lead to improved placement decisions.

We begin by deploying eight VMs across two clients connected to a cluster of two nodes. Each VM disk image holds 32 GiB, but the VMs are configured to run random 4 KiB read workloads over a fixed subset of the disks, such that working set sizes range from 500 MiB to 4 GiB. Given two nodes with 8 GiB of flash each, it is impossible to store all 256 GiB of VM data in flash; however, the total workload footprint as measured by the analysis service is roughly 17 GiB, and if carefully arranged, it can fit almost entirely in flash without exceeding the capacity of any single device by more than 1 GiB.

We measure the application-perceived latency for these VMs in two configurations. In the first, VMs are partitioned evenly among the two nodes using the *local* policy described in § 4.3.1 to avoid network RTTs. In the second, the same placement policy is used, but it is extended with one additional rule that discourages configurations where combined working set sizes exceed the capacity of a given flash card. The cost of violating this rule is higher than the cost of violating the node-local rule, codifying

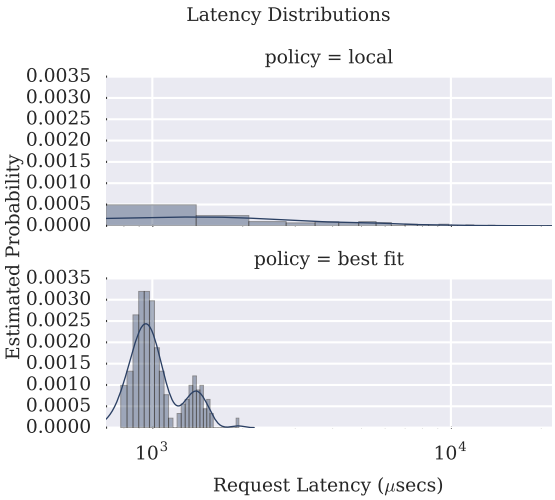


Figure 5: Fitting working sets to flash capacities ('best fit') yields a median latency of 997 μsecs, compared to 2088 μsecs for the 'local' policy that eliminates backend network RTTs but serves more requests from disk.

a preference for remote flash accesses over local disk accesses. The greedy solver is a good fit for this problem and arrives at a configuration in which only one flash device serves a combined working set size larger than its capacity.

As Figure 5 shows, the *best-fit* policy results in significantly lower latencies, because the cost of additional network hops is dwarfed by the penalty incurred by cache misses. The purely local policy exhibits less predictable performance and a long latency tail because of cumulative queuing effects at the disk tier. This is a clear example of how combining knowledge of the relative capabilities of network links and storage tiers with detailed workload profiling can improve placement decisions.

4.4.2 Noisy Neighbor Isolation

We next introduce four cache-unfriendly workloads each with 4 GiB disks. The workloads perform linear scans that, given 4 GiB LRU caches, are always served from disk and result in substantial cache pollution. These workloads make it impossible to completely satisfy the working set size rule of the previous experiment.

We measure the request latency of the original workloads as they compete with these new cache-unfriendly workloads under two policies: a *fair share* policy that distributes the cache-unfriendly workloads evenly across the flash devices, and an *isolation* policy that attempts to limit overall cache pollution by introducing a new rule that encourages co-locating cache-unfriendly workloads on common nodes, regardless of whether or not they fit within flash together. As Figure 6 shows, this lat-

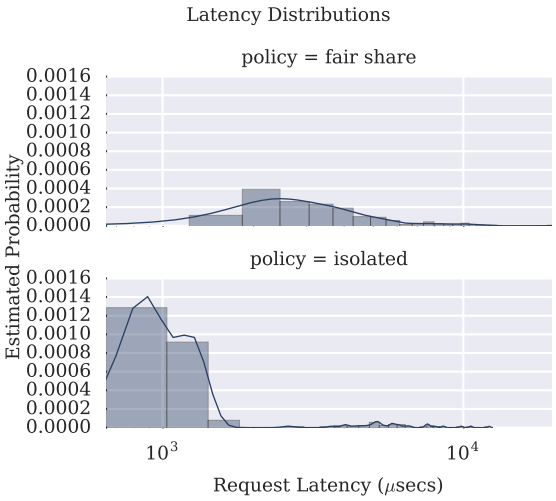


Figure 6: Isolating cache-unfriendly workloads on a single device yields a median latency of 1036 μsecs, compared to 3220 μsecs for the 'fair' policy that distributes these workloads uniformly across all devices.

ter policy exhibits a bimodal latency distribution, with nearly 48% of requests enjoying latencies less than one millisecond while a handful of 'victim' workloads experience higher latencies due to contention with cache-unfriendly competitors. The fair share policy, on the other hand, features a more uniform distribution, with all workloads suffering equally, and a median latency more than three times higher than that of the isolated policy.

4.4.3 Workload Co-scheduling

Finally, we introduce a technique for leveraging long-term temporal patterns in workload behavior to improve data placement. We frequently see storage workloads with pronounced diurnal patterns of high activity at key hours of the day followed by longer periods of idleness. This behavior typically correlates with workday habits and regularly scheduled maintenance tasks [16, 37, 46]. Similar effects can be seen at much smaller scales in CPU caches, where the strategy of co-locating applications to avoid contention is called 'co-scheduling' [50].

We present a simple algorithm for reducing cache contention of periodic workloads. The workload analysis service maintains an extended time series of the footprint of each workload, where footprint is defined as the number of unique blocks accessed over some time window; in this experiment we use a window of ten minutes. Given a set of workloads, we compute the degree to which they contend by measuring how much their bursts overlap. Specifically, we model the cost of co-locating two workloads W_1 and W_2 with corresponding footprint functions $f_1(t)$ and $f_2(t)$ as $\int \min(f_1(t), f_2(t))$. We use this metric

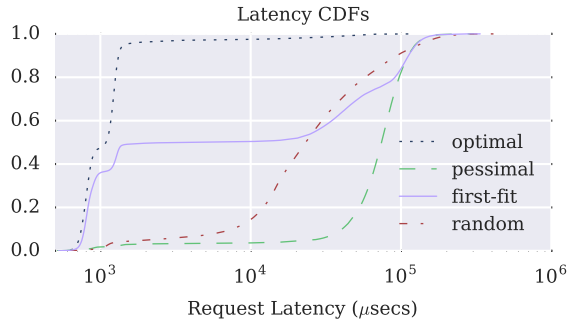


Figure 7: Co-scheduling periodic workloads.

to estimate the cost of placing workloads together on a given device, and employ a linear *first-fit* algorithm [14] to search for an arrangement of workloads across available devices that minimizes the aggregate cost. Finally, we introduce the `co_scheduled` rule which encodes an affinity for assignments that match this arrangement.

We evaluate this heuristic by deploying 8 VMs with 4 GiB disks across two storage nodes each with two 4 GiB flash devices. The VMs perform IO workloads featuring periodic hour-long bursts of random reads followed by idle intervals of roughly 3 hours, with the periodic phases shifted in some VMs such that not all workloads are active at the same time. The combined footprint of any two concurrent bursts exceeds the size of any single flash device, and if co-located, will incur significant paging. We measure request latency under a number of different configurations: *random*, in which stripes are randomly distributed across devices, *optimal* and *pessimal*, in which VMs are distributed two to a device so as to minimize and maximize contention, respectively, and *first-fit*, as described above.

Figure 7 plots latency CDFs for each of these configurations. The penalty of concurrent bursts is evident from the pronounced disparity between the optimal and pessimal cases; in the latter configuration, contention among co-located workloads is high, drastically exceeding the available flash capacity. The first-fit approximation closely tracks optimal in the first two quartiles but performs more like random in the last two, suggesting room for improvement either by developing a more sophisticated search algorithm or responding more aggressively to workload changes.

5 Experience

To see how Mirador performs in real-world environments, we sample logs detailing more than 8,000 rebalance jobs in clusters installed across nearly 50 customer sites and ranging in size from 8 to 96 devices. Figure 8 illustrates how time spent in the optimization stage scales

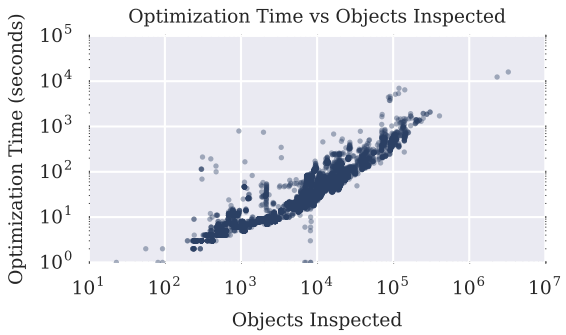


Figure 8: Optimization time vs. objects inspected.

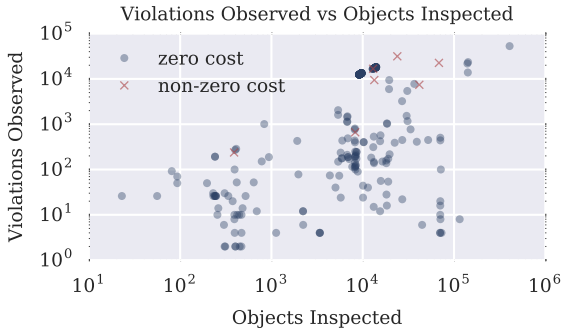


Figure 9: Violations observed vs. objects inspected (jobs where no zero-cost solution was found after a single optimization round are marked with a red x).

in proportion to the number of objects inspected; these measurements include rate-limiting delays imposed to prevent Mirador from impacting client workloads when reading metadata. Figure 9 plots the number of observed violations against the number of objects inspected per job, and highlights jobs that fail to find a zero-cost solution after a single optimization pass. This occurs in only 2.5% of sampled jobs in which objective functions are violated, and in 71% of these cases, no zero-cost solutions are possible due to environmental circumstances (some log samples cover periods in which devices were intentionally taken offline for testing or maintenance).

We have found Mirador's flexibility and extensibility to be two of its best attributes. Over the nearly 18 months in which it has been in production, we have adapted it to new replication policies and storage architectures simply by modifying existing rules and adding new ones. It has also been straightforward to extend Mirador to support new functionality: in addition to providing capacity balancing across storage devices and network links, it now plays a central role in cluster expansion, hardware retirement, failure recovery, health monitoring, and disk scrubbing features. For example, upon discovering an invalid data checksum, our disk scrubbing service simply marks the affected object as corrupt and notifies the placement service, where a custom rule forces the migration of marked objects to new locations, effectively rebuilding them from valid replicas in the process.

Our deployment strategy to date has been conservative: we ship a fixed set of rules (currently seven) and control how and when they are used. Assigning appropriate costs to rules requires domain knowledge, since rules often articulate conflicting objectives and poorly chosen costs can lead to unintended behavior. As an example, if solvers fail to identify a zero-cost solution, they yield the one with the lowest *aggregate* cost – if multiple rules conflict for a given assignment, the assignment which minimizes the overall cost is chosen. It is thus important to know which objective functions a replica set may violate so that high priority rules are assigned costs sufficiently large enough to avoid priority inversion in the face of violations of multiple lower-priority rules.

While objective functions neatly encapsulate individual placement goals and are relatively easy to reason about, comprehensive policies are more complex and must be carefully vetted. We validate rules, both in isolation and combination, with hundreds of policy tests. Declarative test cases specify a cluster configuration and initial data layout along with an expected optimization plan; the test harness generates a storage system model from the specification, invokes the planning engine, and validates the output. We have also built a fuzz tester that can stress policies in unanticipated ways. The test induces a sequence of random events (such as the addition and removal of nodes, changes in load, etc.) and invokes the policy validation tool after each step. Any cluster configuration that generates a policy violation is automatically converted into a test case to be added to the regression suite after the desired behavior is determined by manual inspection. Validating *any* non-trivial placement policy can require a fair amount of experimentation, but in our experience, the cost-based framework provided by Mirador provides knobs that greatly simplify this task.

In production, rebalance jobs run in two passes: the first enforces critical rules related to redundancy and fault tolerance, while the second additionally enforces rules related to load-balancing and performance. This is done because the planning engine must inspect objects in batches (batches are limited to roughly 10,000 objects to keep memory overheads constant), and we want to avoid filling a device in an early batch in order to satisfy low-priority rules when that same device may be necessary to satisfy higher-priority rules in a later batch.

Early testing revealed the importance of carefully tuning data migration rates. Our migration service originally provided two priorities, with the higher of these intended for failure scenarios in which replicas need to be rebuilt. In practice, however, we found that such failures place

additional stress on the system, often driving latencies up. Introducing high-priority migration traffic in these situations can lead to timeouts that only make things worse, especially under load. We have since adopted a single migration priority based on an adaptive queuing algorithm that aims to isolate migration traffic as much as possible while ensuring forward progress is made.

6 Related Work

Researchers have proposed a wide variety of strategies for addressing the data placement problem, also known as the file assignment problem [15]. Deterministic approaches are common in large-scale systems [38, 41, 48, 51, 53] because they are decentralized and impose minimal metadata overheads, and they achieve probabilistically uniform load distribution for large numbers of objects [43,45]. Consistent hashing [30] provides relatively stable placement even as storage targets are added and removed [21, 57]. Related schemes offer refinements like the ability to prioritize storage targets and modify replication factors [26, 27, 52], but these approaches are intrinsically less flexible than dynamic policies.

Non-deterministic strategies maintain explicit metadata in order to locate data. Some of these systems employ random or semi-random placement policies for the sake of simplicity and scalability [34, 39, 42], but others manage placement with hard-coded policies [20, 47]. Customized policies provide better control over properties such as locality and fault tolerance, which can be particularly important as clusters expand across racks [29].

Explicit metadata also make it easier to perform fine-grain migrations in response to topology and workload changes, allowing systems to redistribute load and ameliorate hot spots [35, 37]. Hierarchical Storage Management and multi-tier systems dynamically migrate data between heterogeneous devices, typically employing policies based on simple heuristics intended to move infrequently accessed data to cheaper, more capacious storage or slower, more compact encodings [4, 54].

Mirador has much in common with recent systems designed to optimize specific performance and efficiency objectives. Guerra et al. [23] describe a tiering system that makes fine-grain placement decisions to reduce energy consumption in SANs by distributing workloads among the most power-efficient devices capable of satisfying measured performance requirements. Janus [5] is a cloud-scale system that uses an empirical cacheability metric to arrange data across heterogeneous media in a manner that maximizes reads from flash, using linear programming to compute optimal layouts. Volley [2] models latency and locality using a weighted spring analogy and makes placement suggestions for geographically distributed cloud services. Tuba [9] is a replicated key-value store designed for wide area networks that allows applications to specify latency and consistency requirements via service level agreements (SLAs). It collects hit ratios and latency measurements and periodically reconfigures replication and placement settings to maximize system utility (as defined by SLAs) while honoring client-provided constraints on properties like durability and cost. Mirador supports arbitrary cost-function optimizations using a generic framework and supports policies that control network flows as well as data placement.

Mirador also resembles resource planning systems [6, 8] like Hippodrome [7], which employ a similar observe/optimize/actuate pipeline to design cost-efficient storage systems. Given a set of workload descriptions and an inventory of available hardware, these tools search for low-cost array configurations and data layouts that satisfy performance and capacity requirements. Like Mirador, they simplify a computationally challenging multidimensional bin-packing problem by combining established optimization techniques with domain-specific heuristics. However, while these systems employ customized search algorithms with built-in heuristics, Mirador codifies heuristics as rules with varying costs and relies on generic solvers to search for low-cost solutions, making it easier to add new heuristics over time.

Ursa Minor [1] is a clustered storage system that supports dynamically configurable *m-of-n* erasure codes, extending the data placement problem along multiple new dimensions. Strunk et al. [49] describe a provisioning tool for this system that searches for code parameters and data layouts that maximize user-defined *utility* for a given set of workloads, where utility quantifies metrics such as availability, reliability, and performance. Utility functions and objective functions both provide flexibility when evaluating potential configurations; however, Mirador's greedy algorithm and support for domain-specific hints may be more appropriate for online rebalancing than the randomized genetic algorithm proposed by Strunk et al.

7 Conclusion

Mirador is a placement service designed for heterogeneous distributed storage systems. It leverages the high throughput of non-volatile memories to actively migrate data in response to workload and environmental changes. It supports flexible, robust policies composed of simple objective functions that specify strategies for both data and network placement. Combining ideas from constraint satisfaction with domain-specific language bindings and APIs, it searches a high-dimension solution space for configurations that yield performance and efficiency gains over more static alternatives.

Acknowledgements

The authors would like to thank our shepherd, Kim Keeton, for multiple rounds of thorough and constructive feedback. The paper benefited enormously from Kim's help. We would also like to thank John Wilkes for some very frank and direct comments on an early version of the paper, and Mihir Nanavati for all of his help and feedback along the way.

References

[1] ABD-EL-MALEK, M., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M. P., PRASAD, M., SALMON, B., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Ursa minor: Versatile cluster-based storage. In *FAST* (2005), G. Gibson, Ed., USENIX.

[2] AGARWAL, S., DUNAGAN, J., JAIN, N., SAROIU, S., AND WOLMAN, A. Volley: Automated data placement for geo-distributed cloud services. In *NSDI* (2010), USENIX Association, pp. 17–32.

[3] AGGARWAL, G., MOTWANI, R., AND ZHU, A. The load rebalancing problem. *J. Algorithms 60*, 1 (2006), 42–59.

[4] AGUILERA, M. K., KEETON, K., MERCHANT, A., MUNISWAMY-REDDY, K.-K., AND UYSAL, M. Improving recoverability in multi-tier storage systems. In *DSN* (2007), IEEE Computer Society, pp. 677–686.

[5] ALBRECHT, C., MERCHANT, A., STOKELY, M., WALIJI, M., LABELLE, F., COEHLO, N., SHI, X., AND SCHROCK, E. Janus: Optimal flash provisioning for cloud storage workloads. In *USENIX Annual Technical Conference* (2013), USENIX Association, pp. 91–102.

[6] ALVAREZ, G. A., BOROWSKY, E., GO, S., ROMER, T. H., BECKER-SZENDY, R. A., GOLDING, R. A., MERCHANT, A., SPASOJEVIC, M., VEITCH, A. C., AND WILKES, J. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst. 19*, 4 (2001), 483–518.

[7] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. C. Hippodrome: Running circles around storage administration. In *FAST* (2002), D. D. E. Long, Ed., USENIX, pp. 175–188.

[8] ANDERSON, E., SPENCE, S., SWAMINATHAN, R., KALLAHALLA, M., AND WANG, Q. Quickly finding near-optimal storage designs. *ACM Trans. Comput. Syst. 23*, 4 (2005), 337–374.

[9] ARDEKANI, M. S., AND TERRY, D. B. A self-configurable geo-replicated cloud storage system. In *OSDI* (2014), J. Flinn and H. Levy, Eds., USENIX Association, pp. 367–381.

[10] ASANOVIC, K. Firebox: A hardware building block for 2020 warehouse-scale computers. Keynote presentation, 2014 USENIX Conference on File and Storage Technologies (FAST).

[11] BARR, J. Amazon EBS (Elastic Block Store) - Bring Us Your Data. `https://aws.amazon.com/blogs/aws/amazon-elastic/`, August 2008.

[12] CULLY, B., WIRES, J., MEYER, D. T., JAMIESON, K., FRASER, K., DEEGAN, T., STODDEN, D., LEFEBVRE, G., FERSTAY, D., AND WARFIELD, A. Strata: scalable high-performance storage on virtualized non-volatile memory. In *FAST* (2014), B. Schroeder and E. Thereska, Eds., USENIX, pp. 17–31.

[13] DENNING, P. working set model of program behavior. *Communications of the ACM* (1968).

[14] DÓSA, G. The tight bound of first fit decreasing bin-packing algorithm is FFD(i) <= 11/9OPT(i) + 6/9. In *ESCAPE* (2007), vol. 4614 of *Lecture Notes in Computer Science*, Springer, pp. 1–11.

[15] DOWDY, L. W., AND FOSTER, D. V. Comparative models of the file assignment problem. *ACM Comput. Surv. 14*, 2 (1982), 287–313. comments: ACM Computing Surveys 15(1): 81-82 (1983).

[16] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. I. Passive nfs tracing of email and research workloads. In *FAST* (2003), J. Chase, Ed., USENIX.

[17] EMC. DSSD D5. `https://www.emc.com/en-us/storage/flash/dssd/dssd-d5/index.htm`, 2016.

[18] Flexible io tester. `http://git.kernel.dk/?p=fio.git;a=summary`.

[19] FREUDER, E. A sufficient condition for backtrack-free search. *Communications of the ACM 29*, 1 (1982), 24–32.

[20] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *ACM SIGOPS operating systems review* (2003), vol. 37, pp. 29–43.

[21] GOEL, A., SHAHABI, C., YAO, S.-Y. D., AND ZIMMERMANN, R. SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In *ICDE* (2002), IEEE Computer Society, pp. 473–482.

[22] GRAHAM, R. L. Bounds on multiprocessing anomalies and related packing algorithms. In *Proceedings of the May 16-18, 1972, spring joint computer conference* (New York, NY, USA, 1972), AFIPS '72 (Spring), ACM, pp. 205–217.

[23] GUERRA, J., PUCHA, H., GLIDER, J. S., BELLUOMINI, W., AND RANGASWAMI, R. Cost effective storage using extent based dynamic tiering. In *FAST* (2011), USENIX, pp. 273–286.

[24] HARALICK, R., AND ELLIOT, G. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14*, 3 (1980), 263–313.

[25] HEBRARD, E., HNICH, B., O'SULLIVAN, B., AND WALSH, T. Finding diverse and similar solutions in constraint programming. In *AAAI* (2005), M. M. Veloso and S. Kambhampati, Eds., AAAI Press / The MIT Press, pp. 372–377.

[26] HONICKY, R. J., AND MILLER, E. L. A fast algorithm for online placement and reorganization of replicated data. In *IPDPS* (2003), IEEE Computer Society, p. 57.

[27] HONICKY, R. J., AND MILLER, E. L. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *IPDPS* (2004), IEEE Computer Society.

[28] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference* (2010), P. Barham and T. Roscoe, Eds., USENIX Association.

[29] KANDULA, S., SENGUPTA, S., GREENBERG, A., PATEL, P., AND CHAIKEN, R. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference* (New York, NY, USA, 2009), IMC '09, ACM, pp. 202–208.

[30] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (New York, NY, USA, 1997), STOC '97, ACM, pp. 654–663.

[31] KASHYAP, S. R., KHULLER, S., WAN, Y.-C. J., AND GOLUBCHIK, L. Fast reconfiguration of data placement in parallel disks. In *ALENEX* (2006), R. Raman and M. F. Stallmann, Eds., SIAM, pp. 95–107.

[32] KHULLER, S., KIM, Y.-A., AND MALEKIAN, A. Improved approximation algorithms for data migration. *Algorithmica 63*, 1-2 (2012), 347–362.

[33] KHULLER, S., KIM, Y. A., AND WAN, Y.-C. J. Algorithms for data migration with cloning. *SIAM J. Comput. 33*, 2 (2004), 448–461.

[34] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: an architecture for global-scale persistent storage. *SIGPLAN Not. 35*, 11 (2000), 190–201.

[35] LIN, L., ZHU, Y., YUE, J., CAI, Z., AND SEGEE, B. Hot random off-loading: A hybrid storage system with dynamic data migration. In *MASCOTS* (2011), IEEE Computer Society, pp. 318–325.

[36] MACKWORTH, A. K. Consistency in networks of relations. *Artificial Intelligence 8*, 1 (1977), 99–118.

[37] NARAYANAN, D., DONNELLY, A., THERESKA, E., ELNIKETY, S., AND ROWSTRON, A. I. T. Everest: Scaling down peak loads through i/o off-loading. In *OSDI* (2008), USENIX Association, pp. 15–28.

[38] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O. S., HOWELL, J., AND SUZUE, Y. Flat datacenter storage. In *OSDI* (2012), USENIX Association, pp. 1–15.

[39] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J. K., AND ROSENBLUM, M. Fast crash recovery in RAMCloud. In *SOSP* (2011), ACM, pp. 29–41.

[40] PETERSEN, C. Introducing Lightning: A flexible NVMe JBOF. `https://code.facebook.com/posts/989638804458007/introducing-lightning-a-flexible-nvme-jbof/`, March 2016.

[41] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001* (2001), Springer, pp. 329–350.

[42] SAITO, Y., FRÃŸLUND, S., VEITCH, A. C., MERCHANT, A., AND SPENCE, S. FAB: building distributed enterprise disk arrays from commodity components. In *ASPLOS* (2004), ACM, pp. 48–58.

[43] SANTOS, J. R., MUNTZ, R. R., AND RIBEIRO-NETO, B. A. Comparing random data allocation and data striping in multimedia servers. In *SIGMETRICS* (2000), ACM, pp. 44–55.

[44] SCHIEX, T., FARGIER, H., AND VERFAILLIE, G. Valued constraint satisfaction problems: Hard and easy problems. In *IJCAI (1)* (1995), Morgan Kaufmann, pp. 631–639.

[45] SEO, B., AND ZIMMERMANN, R. Efficient disk replacement and data migration algorithms for large disk subsystems. *TOS 1*, 3 (2005), 316–345.

[46] SHAMMA, M., MEYER, D. T., WIRES, J., IVANOVA, M., HUTCHINSON, N. C., AND WARFIELD, A. Capo: Recapitulating storage for virtual desktops. In *FAST* (2011), G. R. Ganger and J. Wilkes, Eds., USENIX, pp. 31–45.

[47] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–10.

[48] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001* (Aug. 2001).

[49] STRUNK, J. D., THERESKA, E., FALOUTSOS, C., AND GANGER, G. R. Using utility to provision storage systems. In *FAST* (2008), M. Baker and E. Riedel, Eds., USENIX, pp. 313–328.

[50] WANG, X., LI, Y., LUO, Y., HU, X., BROCK, J., DING, C., AND WANG, Z. Optimal footprint symbiosis in shared cache. In *CCGRID* (2015), IEEE Computer Society, pp. 412–422.

[51] 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 *OSDI* (2006), USENIX Association, pp. 307–320.

[52] WEIL, S. A., BRANDT, S. A., MILLER, E. L., AND MALTZAHN, C. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.

[53] WEIL, S. A., LEUNG, A. W., BRANDT, S. A., AND MALTZAHN, C. RADOS: a scalable, reliable storage service for petabyte-scale storage clusters. In *PDSW* (2007), ACM Press, pp. 35–44.

[54] WILKES, J., GOLDING, R. A., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst. 14*, 1 (1996), 108–136.

[55] WIRES, J., INGRAM, S., DRUDI, Z., HARVEY, N. J. A., AND WARFIELD, A. Characterizing storage workloads with counter stacks. In *OSDI* (2014), J. Flinn and H. Levy, Eds., USENIX Association, pp. 335–349.

[56] WONG, T. M., AND WILKES, J. My cache or yours? making storage more exclusive. In *USENIX Annual Technical Conference, General Track* (2002), C. S. Ellis, Ed., USENIX, pp. 161–175.

[57] ZHENG, W., AND ZHANG, G. Fastscale: Accelerate RAID scaling by minimizing data migration. In *FAST* (2011), USENIX, pp. 149–161.

Chronix: Long Term Storage and Retrieval Technology for Anomaly Detection in Operational Data

Florian Lautenschlager,[1] Michael Philippsen,[2] Andreas Kumlehn,[2] and Josef Adersberger[1]

[1]*QAware GmbH, Munich, Germany*
florian.lautenschlager@qaware.de
josef.adersberger@qaware.de

[2]*University Erlangen-Nürnberg (FAU)*
Programming Systems Group, Germany
michael.philippsen@fau.de

Abstract

Anomalies in the runtime behavior of software systems, especially in distributed systems, are inevitable, expensive, and hard to locate. To detect and correct such anomalies (like instability due to a growing memory consumption, failure due to load spikes, etc.) one has to automatically collect, store, and analyze the operational data of the runtime behavior, often represented as time series. There are efficient means both to collect and analyze the runtime behavior. But traditional time series databases do not yet focus on the specific needs of anomaly detection (generic data model, specific built-in functions, storage efficiency, and fast query execution).

The paper presents Chronix, a domain specific time series database targeted at anomaly detection in operational data. Chronix uses an ideal compression and chunking of the time series data, a methodology for commissioning Chronix' parameters to a sweet spot, a way of enhancing the data with attributes, an expandable set of analysis functions, and other techniques to achieve both faster query times and a significantly smaller memory footprint. On benchmarks Chronix saves 20%–68% of the space that other time series databases need to store the data and saves 80%–92% of the data retrieval time and 73%–97% of the runtime of analyzing functions.

1 Introduction

Runtime anomalies are hard to locate and their occurrence is inevitable, especially in distributed software systems due to their multiple components, different technologies, various transport protocols, etc. These anomalies influence a system's behavior in a bad way. Examples are an anomalous resource consumption (e.g., high memory consumption, growing numbers of open files, low CPU usage), sporadic failures due to synchronization problems (e.g., deadlock), or security issues (e.g., port scanning activity). Whatever their root causes are, the resulting behavior is critical and may in general lead to economic or reputation loss (e.g., loss of sales, productivity, or data). Almost every software system has hidden anomalies that occur sooner or later. Hence one needs to detect them in an automated manner soon after their occurrence in order to initiate measures.

There are efficient means both to collect and analyze the runtime behavior. Tools [10, 18, 44] can collect all kinds of operational data like metrics (e.g., CPU usage), traces (e.g., method calls), and logs. They represent such operational data as time series. Analysis tools and research papers [27, 33, 43, 44, 47] focus on the detection of anomalies in that data. But there is a gap between collection and analysis of operational time series, because typical time series databases are general-purpose and not optimized for the domain of this paper. They typically have a data model that focuses on series of primitive type values, e.g., numbers, booleans, etc. Their built-in aggregations only support the analysis of these types. Furthermore, they do not support an explorative and correlating analysis of all the raw operational data in spontaneous and unanticipated ways. With a domain specific data model and with domain specific built-in analysis functions we achieve better query analysis times. General-purpose time series databases already have a good storage efficiency. But we show that by exploiting domain specific characteristics there is room for improvement.

Chronix, a novel domain specific time series database, addresses the collection and analysis needs of anomaly detection in operational data. Its contributions are a multi-dimensional generic time series data model, built-in domain specific high-level functions, and a reduced storage demand with better query times. Section 2 covers the requirements of such a time series database. Section 3 presents Chronix. Section 4 discusses the commissioning of Chronix and describes a methodology that finds a sweet performance spot. The quantitative evaluation in Section 5 demonstrates how much better Chronix works than general-purpose time series databases.

2 Requirements

Three main requirements span the design space of a time series database that better suits the needs of an anomaly detection in operational data: a generic data model for an explorative analysis of all types of operational data, analysis support for detecting runtime anomalies, and a time- and space-efficient lossless storage.

As shown in Table 1, the established general-purpose time series databases Graphite [12], InfluxDB [24], KairosDB [25], OpenTSDB [32], and Prometheus [36] do not or only partially fulfill these requirements.

Generic data model. Software systems are typically distributed and a multi-dimensional data model allows to link operational data to its origin (host, process, or sensor). We argue that a generic multi-dimensional data model is necessary for an explorative and correlating analysis to target non-trivial anomalies. *Explorative* means that a user can query and analyze the data without any restrictions (as by a set of pre-defined queries) to verify hypotheses. *Correlating* means that the user can combine queries on all different types of operational data (e.g., traces, metrics, etc.) without any restrictions (as by pre-defined joins and indexes). Imagine an unanticipated need to correlate the CPU usage in a distributed system with the executed methods. Such an analysis first queries the CPU usage on the hosts (metrics) for a time range. Second, it queries which methods were executed (traces). Finally, it correlates the results in a histogram.

The traditional time series databases have a specific multi-dimensional data model for storing mainly scalar/numeric values. But in the operational data of a software system there are also traces, logs, etc. that often come in (structured) strings. As explicit string encodings require implementation work, often only such raw data is encoded and collected that appears useful at the time of collection. Other raw data is lost, even though an explorative analyses may later need it. Moreover, any string encoding loses the semantics that come with the data type. Therefore, while the traditional time series databases support explorative and correlating analyses on scalar values, they often fail to do so efficiently for generic time series data (e.g., logs, traces, etc.). InfluxDB also supports strings and booleans. KairosDB is extensible with custom types. Both lack operators and functions and hence only partly fulfill the requirements.

Table 1: Design space requirements.

Time Series Database	Generic data model	Analysis support	Lossless long term storage
Graphite	○	◐	○
InfluxDB	◐	◐	●
OpenTSDB	○	○	●
KairosDB	◐	◐	●
Prometheus	○	◐	◐
Chronix	●	●	●

○ = No, ◐ = Partly, ● = Yes

Analysis support. Table 2 is an incomplete list of basic and high-level analysis functions that a storage and retrieval technology for anomaly detection in operational data must support in its plain query language. Graphite, InfluxDB, KairosDB and Prometheus only have a rich set of basic functions for transforming and aggregating operational data with scalar values. OpenTSDB even supports only a few of them. But domain specific high-level functions that other authors [27, 33, 43, 44, 47] successfully use for anomaly detection in operational data (lower part of Table 2) should be built in natively to execute them as fast as possible, without network transfer costs, etc. The evaluation in Section 5 shows the runtime benefits of having them built-in instead of emulating them.

As such a set of functions can never be complete its extensibility is also a requirement.

Efficient lossless long term storage. Complex analy-

Table 2: Common query functions.

Basic	*Graphite*	*InfluxDB*	*OpenTSDB*	*KairosDB*	*Prometheus*	*Chronix*
distinct	×	✓	×	×	×	✓
integral	✓	×	×	×	×	✓
min/max/sum	✓	✓	✓	✓	✓	✓
count	×	✓	✓	✓	✓	✓
avg/median	✓	✓	✓	✓	✓	✓
bottom/top	×	✓	×	×	✓	✓
first	✓	✓	×	✓	×	✓
last	✓	✓	×	✓	×	✓
percentile (p)	✓	✓	✓	✓	✓	✓
stddev	✓	✓	✓	✓	✓	✓
derivative	✓	✓	×	✓	✓	✓
nnderivative	✓	✓	×	×	×	✓
diff	×	✓	×	✓	✓	✓
movavg	✓	✓	×	×	×	✓
divide/scale	✓	✓	×	✓	✓	✓
High-level						
sax [33]	×	×	×	×	×	✓
fastdtw [38]	×	×	×	×	×	✓
outlier	×	×	×	×	×	✓
trend	×	×	×	×	×	✓
frequency	×	×	×	×	×	✓
grpsize	×	×	×	×	×	✓
split	×	×	×	×	×	✓

ses to identify and understand runtime anomalies, trends, and patterns use machine learning, data mining, etc. They need quick access to the full set of all the raw operational data, including its history. This allows interactive response times that lead to qualitatively better explorative analysis results [30]. Thus a lossless long term storage is required that achieves both a small storage demand and good access times. A domain specific storage should also allow to store additional pre-aggregated or in other ways pre-processed versions of the data that enhance domain specific queries.

The operational characteristics of anomaly detection tasks are also specific as there are comparatively few batch writes and frequent reads/analyses, hence the storage should optimized for this.

As Table 1 shows, the traditional time series databases (except for Graphite due to its round-robin storage, see Section 6) can be used as long term storage for raw data. Their efficiency in terms of domain specific storage demands and query runtimes is insufficient as the evaluation in Section 5 will point out. Prometheus is designed as short-term storage with a default retention of two weeks. Furthermore it does not scale, there is no API for storing data, and it uses hashes for time series identification (the resulting collisions may lead to a seldom loss of a time series value). It is therefore not included in the quantitative evaluation.

3 Design and Implementation

Generic data model. Chronix uses a generic data model that can store all kinds of operational data. The key element of the data model is called a *record*. It stores an ordered chunk of time series data of an arbitrary type (n pairs of timestamp and value) in a binary large object. There is also type information (e.g., metric, trace, or log) that defines the available type-specific functions and the ways to store or access the data (e.g., serialization, compression, etc.). A record stores technical fields (version and id of the record, as needed by the underlying Apache Solr [5]), two timestamps for start and end of the time range in the chunk, and a set of user-defined attributes, for example to describe the origin of the operational data, e.g., host and process. Thus the data model is multi-dimensional. It is explorative and correlating as queries can use any combination of the attributes stored in a chunk as well as the available fields.

Listing 1 shows a record for a metric (type) time series, with dimensions for host name, process, group (a logical group of metrics), and metric. Description and version are optional attributes.

The syntax of Chronix queries is:

```
q=<solr-query> [ & cf=<chronix-functions> ]
```

```
1  record{
2    //payload
3    data:compressed{<chunk of time series data>}
4
5    //technical fields (storage dependent)
6    id: 3dce1de0−...−93fb2e806d19 //16 bytes
7    _version_: 1501692859622883300 //8 bytes
8
9    //logical fields
10   start: 1427457011238 //27.3.2015 11:51:00 8 bytes
11   end: 1427471159292 //27.3.2015 15:45:59 8 bytes
12   type: metric //Data types: metric, log, trace etc.
13
14   //optional dimensions
15   host: prodI5
16   process: scheduler
17   group: jmx
18   metric: heapMemory.Usage.Used
19
20   //optional attributes
21   description: Benchmark
22   version: v1.0−RC
23 }
```

c bytes · comp. rate

m bytes

Listing 1: Record with fields, attributes, and dimensions.

The underlying Solr finds the data according to `q`. Then we apply the optional Chronix functions `cf` before we ship the result back to the client. `Cf` is a ;-separated list of filters, each of the form

<type>'{'<functions>'}'

The ;-separated list of <type>-specific <functions> is processed as a pipeline. A <function> has the form

<analysis-name>[:<params>]

for analysis names from Table 2. Here is an example from the evaluation project 4 in Section 5:

```
q=host:prod* AND type:[lsof OR strace]
& cf=lsof{grpsize:name,pipe};strace{split:2030}
```

First, `q` takes all time series from all hosts whose name starts with `prod` and that also hold data of the UNIX commands `lsof` or `strace`. Then `cf` applies two functions. To the `lsof` data `cf` applies `grpsize` to select the group named `name` and counts the occurrences of the value/word `pipe` in that group. On the `strace` data `cf` performs a `split`-operation on the command-column. For each of the arguments (here just for the file handle ID 2030) it produces a split of the data that contains `"2030"`.

Analysis support. Chronix offers all the basic and high-level functions listed in Table 2, e.g., there are detectors for outliers and functions that check for trends, similarities (fastdtw), and patterns (sax). The plug-in mechanism of Chronix allows to add functions that run server-side. They are fast as they operate close to the data

```
1  class GroupSize implements ChronixAnalysis<LsofTS>{
2   public GroupSize(String[] args) {
3    field = args[0];filters = args[1];
4   }
5   public void execute(LsofTS ts, Result result) {
6    result.put(this,new Map())
7    for (Group group : ts.groupBy(field)) {
8      if (filters.contains(group.key()))
9        result.get(this).put(group.key(), group.value().size());
10   }
11  }
12  public String getQueryName() {return "grpsize";}
13  public String getTimeSeriesType() {return "lsof";}
14 }
```

Listing 2: Plug-in for a function *GroupSize*.

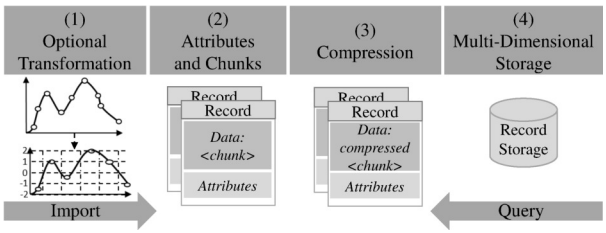


Figure 1: Building blocks of Chronix: (1) optional transformation, (2) grouping of time series data into chunks plus attributes, (3) compression of chunks, (4) storage of chunks with their attributes. The data flows from left to right for imports. For queries it is the other way round.

without shipping (e.g., through HTTP). For security reasons only an administrator is capable (and thus responsible) for installing plug-ins while regular users can only use them afterwards. The plug-in developer has to implement an interface called `ChronixAnalysis`. It defines three methods: `execute` holds the code that analyzes a time series, `getQueryName` is the function's name in Chronix' Query Language, and `getTimeSeriesType` binds the function to a time series type. An additional Google Guice [19] module is needed before (after a reboot of Chronix with the operator code in the classpath) the newly added function is available. Chronix parses a given query and delegates the execution to the plugged-in function. Listing 2 illustrates how to add the `grpsize` analysis for the time series type `lsof`. The function runs on the server, groups the time series data with respect to a field (given as its first argument), and returns the sizes of the groups. The raw time series data is never shipped to the client. With this function added, regular users just need to call one function instead of several queries to emulate the semantics themselves.

Functionally-lossless long term storage. Chronix optimizes storage demands and query times and is designed for few batch writes and frequent reads. The storage is functionally-lossless and preserves all the data that can potentially be useful for anomaly detection. We clarify the two situations below when *functionally-lossless* is different from *lossless*.

Chronix' pipeline architecture has four building blocks that run multi-threaded: *Optional Transformation, Attributes and Chunks, Compression*, and *Multi-Dimensional Storage*, see Fig. 1. Not shown is the input buffering (called Chronix Ingester in the distribution) that batches incoming raw data points.

The *Optional Transformation* can enhance or extend what is actually stored in addition to the raw data, or instead of it. The goal of this optional phase is an optimized format that better supports use-case specific analyses. For example, it is often easier to identify patterns in time series when a symbolic representation is used to store the numerical values of a time series [33]. Other examples are averaging, a Fourier transformation [9], Wavelets [35], or a Piecewise Aggregate Approximation [26]. The more is known about future queries and about the shape or structure of a time series or its pattern of redundancy, the more can a transformation speed up analyses. In general, this phase can add an additional representation of the raw data to the record. Storing the raw data can even be omitted if it can be reconstructed. In rare situations, an expert user knows for sure that certain data will never be of interest for any of her/his anomaly detection tasks. Think of a regular log file that in addition to the application's log messages also has entries that come from the framework hosting the application. The latter often do not hold anomaly-related information. It is such data that the expert user can decide to drop, keeping everything else and leaving the Chronix store *functionally-lossless*. The decision is irreversible – but this is the same with the other time series databases.

Attributes and Chunks breaks the raw time series (or the result of the optional transformation) into chunks of `n` data points that are serialized into `c` bytes. Instead of storing single points, chunking speeds up the access times. It is known to be faster to read one record that contains `n` points instead of reading `n` single points. A small `c` leads to many small records with potentially redundant user-defined attributes between records. The value of `c` is therefore a configuration parameter of the architecture.

This stage of the pipeline also calculates both the required fields and the user-defined attributes of the records. The required fields (of the data model) are the binary `data` field that holds a chunk of the time series, and the fields `start` and `end` with timestamps of the first and the last point of the chunk. In addition, a record can have user-defined attributes to store domain specific information more efficiently. For instance, information that

```
1 public interface RecordConverter<T> {
2   //Convert a record to a specific type.
3   //Use queryStart and queryEnd to filter the record.
4   T from (Record r, long queryStart, long queryEnd);
5   //Convert a specific type to a record.
6   Record to (T tsChunk);
7 }
```

Listing 3: The time series converter interface.

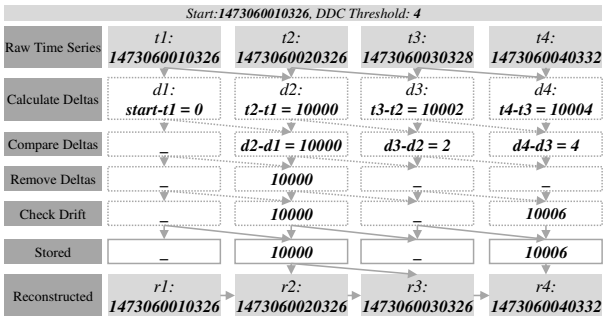


Figure 2: DDC calculates timestamp deltas (0, 10000, 10002, 10004) and compares them (0, 10000, 2, 4). It removes deltas below a threshold of 4 (_, 10000, _, _). Without additional deltas, the reconstruction would yield timestamps with increments of 10000. Since the fourth of them would be too far off from `t4` (drift of 6) DDC stores the correcting delta 10006. With the resulting two stored deltas (_, 10000, _, 10006) the reconstructed timestamps are error-free, even for `r4`.

is known to be repetitive for each point, e.g., the host name, can be stored in an attribute of the record instead of encoding it into the chunk of data multiple times.

As it is specific for a time series type which fields are repetitive Chronix leaves it to an expert user to design the records and the fields. Redundancy-free data chunks are a domain specific optimization that general-purpose time series databases do not offer. To design the records and fields the expert has two tasks: (1) define the conversion of a time series chunk to a record (see the example `RecordConverter` interface in Listing 3) and (2) define the static schema in Apache Solr that lists the field names, the types of the values, which fields are indexed, etc. Chronix stores every record that matches the schema.

Compression processes the chunked data. Chronix exploits domain specific characteristics in three ways.

First, Chronix compresses the operational data significantly as there are only small changes between subsequent data points. The cost of compression is acceptable as there are only few batch writes. When querying the data, compression even improves query times as compressed data is transferred to the analysis client faster due to the reduced amount.

Second, time series for operational data often have periodic time intervals, as measurements are taken on a regular basis. The problem in practice is that there is often a jitter in the timestamps, i.e., the time series only have *almost*-periodic time intervals because of network latency, I/O latency, etc. Traditional time series databases use optimized ways to store periodic time series (run-length encoding, delta-encoding, etc.). But in case of jitter they fall back to storing the full timestamps/deltas. In contrast, Chronix' Date-Delta-Compaction (DDC) exploits the fact that in its domain the exact timestamps do not matter that much, at least if the difference between the expected timestamp and the actual timestamp is not too large. Here Chronix' storage is *functionally-lossless* because by default it drops timestamps if it can almost exactly reproduce them. If in certain situations an expert user knows that the exact timestamps do matter, they can be kept. In contrast to timestamp jitter, Chronix never drops the exact values of the data points. They always matter in the domain of anomaly detection. Nevertheless, some parts of the related work are lossy, see Table 1.

The central idea of the DDC is: When storing an almost-periodic time series, the DDC keeps track of the expected next timestamp and the actual timestamp. If the difference is below a threshold, the actual timestamp is dropped as its reconstructed value is close enough. The DDC also keeps track of the accumulated drift as the difference between the expected timestamps and actual timestamps adds up with the number of data points stored. As soon as the drift is above the threshold, DDC stores a correcting delta that brings the reconstruction back to the actual timestamp. DDC is an important domain specific optimization. See Section 4 for the quantitative effects. The DDC threshold is another commissioning parameter. Fig. 2 holds an example.

There are related approaches [34] that apply a similar idea to both the numeric values and the timestamps. Chronix' DDC avoids the lossy compression of values as the details matter for anomaly detection. Chronix also exploits the fact that deltas are much smaller than full timestamps and that they can be stored in fewer bits. Chronix' serialization uses Protocol Buffers [20].

Third, to further lower the storage demand, Chronix compresses the records' binary data fields. The attributes remain uncompressed for faster access. Chronix uses one of the established lossless compression techniques `t` = bzip2 [39], gzip [14], LZ4 [11], Snappy [21], and XZ [42] (others can be plugged in). Since they have a varying effectiveness depending on the size of the data blocks that are to be compressed, the best choice `t` is another commissioning parameter.

The *Multi-Dimensional Storage* handles large records

of pure data in a compressed binary format. Only the logical fields, attributes, and dimensions that are necessary to locate the records are explicitly visible to the data storage which uses a configurable set of them for constructing indexes. (Dimensions can later be changed to match future analysis needs.) Queries can use any combination of the dimensions to efficiently locate records, i.e., to find a matching (compressed) chunk of time series data. Information about the data itself is not visible to the storage and hence it is open to future changes as needed for explorative and correlating analyses. Chronix is based on Apache Solr as both the document-oriented storage format of the underlying reverse-index Apache Lucene [4] and the query opportunities match Chronix' requirements. Furthermore Lucene applies a lossless compression (LZ4) to all stored fields in order to reduce the index size and Chronix inherits the scalability of Solr as it runs in a distributed mode called SolrCloud that implements load balancing, data distribution, fault tolerance, etc.

4 Commissioning

Many of the available general-purpose time series databases come with a set of parameters and default values. It is often unclear how to adjust these values to tune the database so that it performs well on the use-case at hand.

We now discuss the commissioning that selects values for Chronix' three adjustable parameters: `d` (DDC threshold), `c` (chunk size), and `t` (compression technique). Commissioning has two purposes: (1) to find default parameters for the given domain (and to show the effects of an unfortunate choice) and (2) to describe a tailoring of Chronix for use-case specific characteristics.

Let us start with a typical domain specific dataset and query mix (composed from real-world projects). We then sketch the measurement infrastructure that the evaluation in Section 5 also uses and discuss the commissioning.

Commissioning Data = Dataset plus Query Mix. To determine its default parameter configuration, Chronix relies on three real-world projects that we consider typical for the domain of anomaly detection in operational data. From these projects, Chronix uses the time series data and the queries that analyze the data.

Project 1 is a web application for searching car maintenance and repair instructions. In production, 8 servers run the web application to visualize the results and 20 servers perform the users' searches. The operational time series data is analyzed to understand the resource consumption for growing numbers of users and new functions, i.e., to answer questions like: 'How do multiple users and new functions affect the CPU load, memory consumption, or method runtimes?', or 'Is the time needed for a user search still within predefined limits?'

Table 3: Project Statistics.

Project	1	2	3	total
pairs (mio)	2.4	331.4	162.6	496.4
time series	1,080	8,567	4,538	14,185

(a) Pairs and time series per project.

Project	1	2	3	average
Attributes (bytes)	43	47	48	46

(b) Average size of attributes per record.

`r`	0.5	1	7	14	21	28	56	91
`q`	15	30	30	10	5	3	1	2

(c) Time ranges (in days) and their occurrence.

Project 2 is a retail application for orders, billing, and customer relations. The production system has a central database, plus two servers. From their local machines, users run the application on the servers via a remote desktop service. The analysis goals are to investigate the impact of a new JavaFX-based UI Framework that replaces a former Swing-based version, and to locate the causes of various reported problems, e.g., memory leaks, high CPU load, and long runtimes of use-cases.

Project 3 is a sales application of a car manufacturer. There are two application servers and a central database server in the production environment. The analysis goals are to understand and optimize the batch-import, to identify the causes of long-running use-cases reported by users, to improve the database layer, and understand the impact that several code changes have.

In total, the projects' operational data have about 500 million pairs of timestamp and (scalar) value in 14,185 time series of diverse time ranges and metrics, see Table 3(a). Two projects have an almost-periodic time interval of 30 or 60 seconds. All projects also have event-driven data, e.g., the duration of method calls. There are recurring patterns (e.g., heap usage), sequences of constant values (e.g., size of a connection pool), or even erratic values (e.g., CPU load).

All time series of the three projects have the same user-defined attributes (host, process, group, and metric) that takes 46 bytes on average, see Table 3(b). The required fields of the records take 40 bytes, leading to a total of $\mathtt{m} = 40 + 46 = 86$ uncompressed bytes per record.

The three projects have 96 queries in total. Table 3(c) shows what time ranges (`r`) they are asking for and how often such a time range is needed (`q`). For example, there are two queries that request the log data that was accumulated over 3 months (91 days).

Measurement Infrastructure. Measurements were conducted on a 12-core Intel Xeon CPU E5-2650L v3@1.80GHz, equipped with 32 GB of RAM and a 380 GB SSD and operating under Ubuntu 16.04.1 x64.

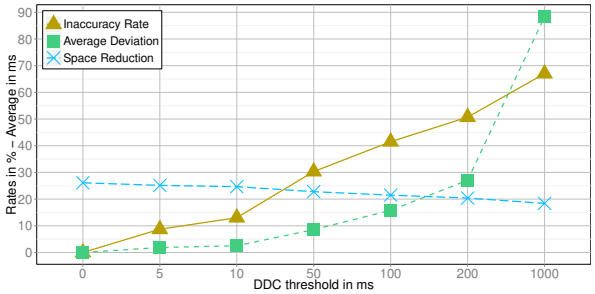


Figure 3: Impact of different DDC thresholds.

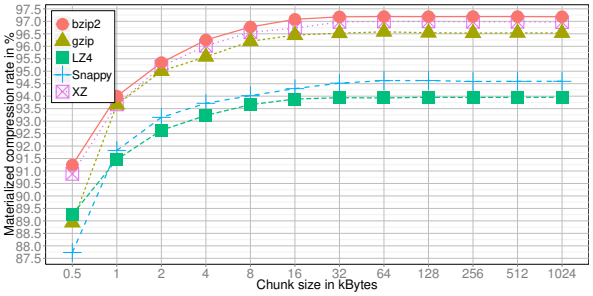


Figure 4: Materialized compression rate in the data store.

Commissioning of the DDC threshold. The higher the DDC threshold is, the higher is the deviation between the actual and the reconstructed timestamp. On the other hand, higher DDC thresholds result in fewer deltas that hence need less storage. Since for anomaly detection, accuracy is more important than storage consumption, and since the acceptable degree of inaccuracy is use-case specific, the commissioning of the DDC threshold focuses on accuracy and only takes the threshold's impact on the storage efficiency into account if there is a choice.

The commissioning works as follows: For a broad range of potential thresholds, apply the DDC to the time series in the dataset. For each timestamp, record whether the reconstructed value is different from the actual timestamp, and if so, how far the reconstructed value is off. The fraction of the number of inaccurately reconstructed timestamps to error-free ones is the *inaccuracy rate*. For all inaccurately reconstructed timestamps compute their *average deviation*. With those two values plotted, the commissioner selects a threshold that yields the desired level of accuracy.

Default Values. The commissioning of the DDC threshold can be done for individual time series or for all time series in the dataset. For the default value we use all time series that are not event-driven. For thresholds from 0 milliseconds up 1 second. Fig. 3. shows that the *inaccuracy rate* grows up to 67%; the *average deviation* of the reconstructed timestamps grows up to 90 ms.

From the experience gained from anomaly detection projects (the three above projects are among them) an average deviation of 30 ms seems to be reasonably small – recall that the almost-periodic time series in the dataset have intervals of 30 or 60 seconds. The acceptable jitter is thus below 0.1% or 0.05%, resp. Therefore we choose a default DDC threshold of $\mathtt{d} = 200$ ms which implies an *inaccuracy rate* of about 50% that we deem acceptable because of the absolute size of the deviation.

Note that the DDC is effective as the resulting data only take 27% to 19% of the original space. But for the dataset the curve of the *space reduction* is too flat to affect the selection of the DDC threshold.

Commissioning of the compression parameters. Many of the standard compression techniques achieve better compression rates on larger buffers of bytes [7]. On the other hand, it takes longer to decompress larger buffers to get to the individual data point. Because of the conflicting goals of storage efficiency and query runtimes, Chronix uses a domain specific way to find a sweet spot. There are three steps.

1. Minimal Chunk Size. First, Chronix finds a minimal chunk size for its records. Chronix is not interested in the compression rate in a memory buffer. For anomaly detection, what matters instead is the total storage demand, including the size of the index files (smaller for larger and fewer records). On the other hand, smaller chunks have more redundancy in the records' (potentially) repetitive uncompressed attributes which makes the compression techniques work better. We call the quotient of this total storage demand and the size of the raw time series data the *materialized compression rate*.

Finding a minimal chunks size works as follows: For a range of potential chunk sizes, construct the records (with their chunks and attributes) from the DDC-transformed time series data and compress them with the standard compression techniques. With the materialized compression rate plotted, the commissioner finds the minimal chunk size where saturation sets in.

Default Values. Fig. 4 shows the materialized compression rates that $\mathtt{t}=$ bzip2, gzip, LZ4, Snappy, and XZ achieve on records with various chunk sizes that are constructed from the (DDC-transformed) dataset. Saturation sets in around a minimal chunk size of $\mathtt{c}_{min}$=32 KB. Larger chunks do not improve the compression rate significantly, regardless of the compression technique.

2. Candidate Compression Techniques. Then Chronix drops some of the standard compression techniques from the set of candidates. Papers on compression techniques usually include benchmarks on decompression times. But for the domain of anomaly detection, the decompression of a whole time series that is in a memory buffer is irrelevant. What matters instead is the total time needed to find and access an individual record, to then ship the

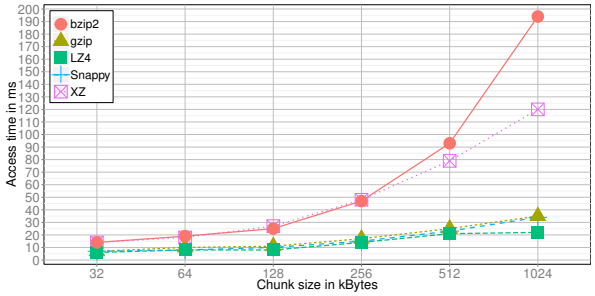


Figure 5: Access time for a single chunk (in ms).

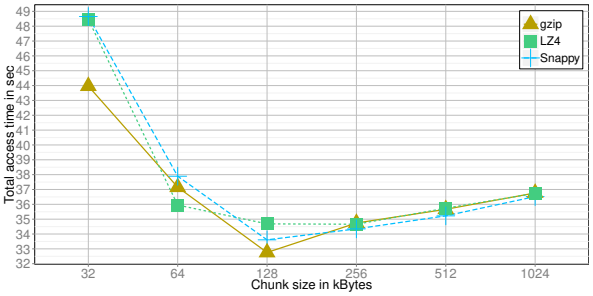


Figure 6: Total access times for $20 \cdot 96$ queries (in sec.).

compressed record to the client, and to decompress it there. Note that the per-byte cost of shipping goes down with growing chunks due to latency effects.

Finding the set of candidate compression techniques works as follows: For potential chunk sizes above c_{min}, find a record, ship it, and reconstruct the raw data. For meaningful results, process all records, and compute the average runtime. The commissioner drops those compression techniques that have a slow average access to single records. The reason is that in anomaly detection data is read much more often than written.

Default Values. On the compressed dataset, all of the standard compression techniques take longer to find, access, and decompress larger chunks, see Fig. 5. It is obvious that bzip2 and XZ can be dropped from the set of compression techniques because the remaining ones clearly outperform them.

3. Best Combination. Now that the range of potential values for `c` is reduced and the set of candidates for the compression technique `t` is limited, the commissioning considers all combinations. Since query performance is more important than storage efficiency for the domain of anomaly detection, commissioning works with a typical (or a use-case specific) query mix. The access time to a single record as considered above can only be an indicator, because real queries request time ranges that are either part of a single record (a waste of time in shipping and decompression if the record is large) or that span multiple records (a waste of time if records are small).

This commissioning step works as follows: Randomly retrieve `q` time ranges of size `r` from the data. The values of `r` and `q` reflect the characteristics of the query mix, see Table 3(c). Repeat this 20 times to stabilize the results. For a query of size `r` the commissioning does not pick a time series that is shorter than `r`. From the resulting plot, the commissioner then picks the combination of chunk size `c` and compression technique `t` that achieves the shortest total runtime for the query mix.

Default Values. For chunk sizes `c` from 32 to 1024 KB and for the remaining three compression techniques `t`, Fig. 6 shows the *total access time* of all the $20 \cdot 96 = 1920$ randomly selected data retrievals that represent the query mix. There is a bath tub curve for each of the three compression techniques `t`, i.e., there is a chunk size `c` that results in a minimal total access time. As `t`=gzip achieves the absolute minimum with a chunk size of `c`=128 KB, Chronix selects this sweet spot, especially as the other options do not show better materialized compression rates for that chunk size (see Fig. 4). The default parameters are good-natured, i.e., small variations do not affect the results much. Fig. 6 also shows the effect of suboptimal choices for `c` and `t`.

Commissioning and re-commissioning for a use-case specific set of parameters are possible but costly as the latter affects the whole data. At the end of the next section, we discuss the effect of a use-case specific commissioning compared to the default parameter set.

5 Evaluation

We quantitatively compare the memory footprint, the storage efficiency, and the query performance of Chronix (without any optional transformation and without any pre-computed analysis results) to InfluxDB, OpenTSDB, and KairosDB. After a sketch of the setup, we describe two case-studies whose operational data serve as benchmark. Then we discuss the results and demonstrate that Chronix' default parameters are sound.

Setup. For the minimal realistic installation all databases run on one computer and the Java process issuing the queries via HTTP runs on a second computer (for hardware details see Section 4). InfluxDB, OpenTSDB, and KairosDB store time series with different underlying storage technologies: InfluxDB (v.1.0.0) uses a custom storage called time structured merge tree, OpenTSDB (v.2.2.0) uses the wide-column store HBase [23] that stores data in the Hadoop Distributed File System (HDFS) [41], and KairosDB (v.1.1.2) stores the data in the wide-column store Cassandra [3]. They also have different strategies for storing the attributes: InfluxDB stores them once with references to the data points they belong to. OpenTSDB and KairosDB store

Table 4: Project Statistics.

Project		4	5	total
time series		500	24,055	24,555
pairs (mio)	metric	3.9	3,762.3	3,766.2
	lsof	0.4	0.0	0.4
	strace	12.1	0.0	12.1

(a) Pairs and time series per project.

r	0.5	1	7	14	21	28	56	91	180	
q	2	11	15	8	12	5	1	2	2	58
b	1	6	5	7	2	4	4	1	2	32
h	2	6	10	8	6	6	3	2	0	43

(b) Time ranges r (days); # of raw data queries (q), of queries with basic (b) and high-level (h) functions.

them in key-value pairs that are part of the row-key. We use default configurations for each database. To measure the query runtimes and the resource consumptions each database is called separately and runs it in its own Docker container [16] to ensure non-interference. We batch-import the data once and run the queries afterwards. We do not discuss the import times as they are less important for the analysis.

Case-studies/benchmark. We collected 108 GByte of operational data from two real-world applications. In contrast to the dataset used in the commissioning there are also data of `lsof` and `strace`, see Table 4(a).

Project 4 detects anomalies in a service application for modern cars (such as music streaming). The goal is to locate the root cause of a file handle leak that forces the application to nightly rebootings of its two application servers. The collected operational data covers 3 hours with an almost-periodic interval of 1 second.

Since we have mentioned an example query of this project in Section 3 and since we have shown how the `grpsize` function (Listing 2) can be plugged-in, let us give some more details on this project.

The initial situation was that the application kept opening new file handles without closing old ones. After reproducing the anomaly in a test environment, for the explorative analysis we employed `lsof` to show the open files, and stored this operational data in Chronix. The results of queries like

```
q=type:lsof & cf=lsof{grpsize:name,*}
```

were the key to explain the rise in the number of open file handles as about 2,000 new file handles were `pipes` or `anon_inodes` that are part of the `java.nio` package. Hence it was necessary to dig deeper and to link file handles to system calls. To do so, we used `strace` and also stored the data in Chronix. By narrowing down the time series data to individual file handle IDs, with correlating queries like the one shown in Section 3 we found

```
1 //End of strace for file handle ID = 2030
2 epoll_ctl(2129, EPOLL_CTL_ADD, 2030,
3                 {EPOLLIN, {u32=2030, u64=2030}}) = 0
4 //End of strace for file handle ID = 2032
5 epoll_ctl(2032, EPOLL_CTL_ADD, 1889,
6                 {EPOLLIN, {u32=1889, u64=1889}}) = 0
```

Listing 4: Last `strace` calls for two file handle IDs.

Table 5: Memory footprint in MBytes.

	InfluxDB	OpenTSDB	KairosDB	Chronix
Initially	33	2,726	8,763	446
Import (max)	10,336	10,111	18,905	7,002
Query (max)	8,269	9,712	11,230	4,792

that `epoll_ctl` was often the last function call before the anomaly (see Listing 4). By then analyzing which third party libraries the application uses and by gathering information on `epoll_ctl` we deduced that the application used an old version of Grizzly that leaks selectors [1] when it tries to write to an already closed channel. The solution was to upgrade the affected library.

Project 5 detects anomalies in an application that manages the compatibility of software components in a vehicle. The production system has a central database and six application servers. The operational data is analyzed to find and understand production anomalies, such as long running method calls, positive trends of resource consumption, etc. The dataset has an almost-periodic interval of 1 min. and holds seven months of operational data.

Table 4(b) shows the mix of the 133 queries that the projects needed to achieve their analysis goals. There are different ranges (`r`) for the 58 raw data retrievals (`q`) that do not have a `cf`-part and also for the 32 queries that use basic analysis functions (`b`-queries) and for the 43 `h`-queries that use high-level analyses. Table 8 lists which of the built-in analysis functions from Table 2 the projects actually use (and how often).

Memory Footprint. Table 5 shows the memory footprint of all database *processes* at different times. This is relevant as analyses on large amounts of time series data are often memory intensive. The first line shows the memory consumption just after a container's start, when all components of the time series database are up and running. The next two lines show the maximal memory footprints that we encountered while the data of the two benchmark projects was imported and while the query mix was executed. All databases stay below the maximal available memory of 31.42 GB. The import (buffering, index construction, etc.) needs more memory than the query mix (reading, decompression, serialization, ship-

Table 6: Storage demands of the data in GBytes.

Project	Raw data	InfluxDB	OpenTSDB	KairosDB	Chronix
4	1.2	0.2	0.2	0.3	0.1
5	107.0	10.7	16.9	26.5	8.6
total	108.2	10.9	17.1	26.8	8.7

ping, etc.). OpenTSDB and KairosDB clearly take the most memory due to their various underlying components. InfluxDB is better but still takes 1.5 times more memory than Chronix. The reasons for Chronix' lower memory usage are: (a) it does not hold large amounts of data in memory at once, (b) it runs as a single process with lightweight thread parallelism, and (c) its implementation avoids expensive object allocations.

Storage Efficiency. Table 6 shows the storage demands of the data, including the index files, commit logs, etc. There are three aspects to note. First, out-of-the-box none of the general-purpose databases can handle the `lsof` and `strace` data. Extra programming was needed to make these databases utilizable for the case-studies. (For both OpenTSDB and InfluxDB we had to encode the non-numeric data in tags or in timestamps plus strings, including some escape mechanisms for special characters. For KairosDB we had to explicitly implement and add custom types.) Second, both OpenTSDB and KairosDB cannot handle the nanosecond precision of the `strace` data. We chose to let them store imprecise data instead, because (explicitly) converted timestamps would have taken even more space. Third, all measurements are done after the optimizations and compactions. During the import the databases temporarily take more disk space (e.g., for commit logs etc.).

Chronix only needs 8.7 GBytes to store the 108.2 GBytes of raw time series data. Compared to the general-purpose time series databases Chronix saves 20%–68% of the storage demand. This is caused by Chronix' domain specific optimizations and by differences in the underlying storage technologies. By default, OpenTSDB does not compress data, but for a fair comparison we used it with gzip. InfluxDB stores rows of single data points using various serialization techniques, such as Pelkonen et al. [34] for numeric data. KairosDB uses LZ4 that has a lower compression rate.

Data Retrieval Performance. The case-studies have 58 raw data queries (`q`) in their mix, with various time ranges (`r`). Table 7 gives the retrieval times. They include the time to find, load, and ship the data and the time to deserialize it on the side of client. For the measurements, the data retrieval mix is again repeated 20 times to stabilize the results, with `q` randomly picked time ranges `r`.

Table 7: Data retrieval times for $20 \cdot 58$ queries (in s).

r	q	InfluxDB	OpenTSDB	KairosDB	Chronix
0.5	2	4.3	2.8	4.4	0.9
1	11	5.2	5.6	6.6	5.3
7	15	34.1	17.4	26.8	7.0
14	8	36.2	14.2	25.5	4.0
21	12	76.5	29.8	55.0	6.0
28	5	7.9	3.9	5.6	0.5
56	1	35.4	12.4	24.1	1.2
91	2	47.5	15.5	33.8	1.1
180	2	96.7	36.7	66.6	1.1
total		343.8	138.3	248.4	27.1

Table 8: Times for $20 \cdot 75$ `b`- and `h`-queries (in s).

	Basic (`b`)	InfluxDB	OpenTSDB	KairosDB	Chronix
4	avg	0.9	6.1	9.8	4.4
5	max	1.3	8.4	9.1	6.0
3	min	0.7	2.7	5.3	2.8
3	stddev.	6.7	16.7	21.1	2.3
5	sum	0.7	6.0	12.0	2.0
4	count	0.8	5.5	10.5	1.0
8	perc.	10.2	25.8	34.5	8.6
	High-level (`h`)				
12	outlier	30.7	29,1	117.6	18.9
14	trend	162.7	50.4	100.6	30.2
11	frequency	47.3	23.9	45.7	16.3
3	grpsize	218.9	2927.8	206.3	29.6
3	split	123.1	2893.9	47.9	37.2
75	total	604.0	5996.3	620.4	159.3

InfluxDB is the slowest, followed by KairosDB and OpenTSDB. Chronix is the fastest and saves 80%–92% of the time needed for the raw data retrieval. For all databases the retrieval times grow with larger ranges. But for Chronix, they grow more slowly. There are several reasons for this: (a) Chronix uses an index to access the chunks and hence avoids full scans, (b) its pipeline architecture ships the raw chunks to the client that can process (decompress, deserialize) them in parallel, and (c) Chronix selects its chunk size and compression to suit these queries.

Built-in Function Advantages. In addition to raw data retrieval, anomaly detection in operational data also needs analyzing functions, several of which the general-purpose time series databases do not natively support (see Table 2) and whose functionality has to be implemented by hand and typically with more than one query.

Table 8 shows the runtimes (20 repetitions for stabilization) that use basic (`b`) and high-level (`h`) functions and how often the projects use them (first column). In total, Chronix saves 73%–97% of the time needed by the general-purpose time series databases. We discuss the results for queries with basic functions (`b`-queries) and with high-level functions (`h`-queries) in turn.

In total, the 32 `b`-queries that other time series databases also natively support account for not more than about 17% of the total runtime. Thus, speed variations for `b`-queries do not matter that much for anomaly detection tasks. Nevertheless, let us delve into the upper part of Table 8. OpenTSDB and KairosDB are often slower than InfluxDB or Chronix. Whenever InfluxDB can use its pre-computed values (for instance for average, maximum, etc.) it outperforms Chronix. When on-the-fly computations are needed (deviation and percentile), Chronix is faster.

The lower part of Table 8 illustrates the runtimes of the 43 `h`-queries. They are important for the anomaly detection projects as they are used much more often than the other functions. Here Chronix has a much more pronounced lead over the general-purpose databases. The reason is that Chronix offers built-in means to evaluate these functions server-side, whereas they have to be manually implemented on the side of the client in the other systems, with additional raw data retrievals.

Let us look closer at the penalties for the lack of such built-in functions. To implement an *outlier* detector in the other systems, one has to calculate the threshold value as (Q3 - Q1) · 1.5 + Q3 where Q1 is the first and Q3 is the third quartile of the time series values. With InfluxDB this needs one extra query. KairosDB needs two extra queries, one for getting Q1 and one for Q3, plus a client-side filter. OpenTSDB does neither provide a function for getting Q1 nor for filtering values. In the other systems a *trend* detector (that checks if the slope of a linear regression through the data is above or below a threshold) has to be built completely on the side of the client. A *frequency* detector (that splits a time series into windows, counts the data points, and checks if the delta between two windows is above a predefined threshold) is more costly to express and to run in the other systems as well. InfluxDB needs one extra query and a client-side validation. OpenTSDB and KairosDB need a query plus code for an extra function on the side of the client. The *grpsize* and the *split* functions that run through this paper are crucial for project 4 both have to be implemented on the side of the client with an extra query for raw values.

Although it was possible to emulate the high-level functions, we ran into problems that are either caused by the missing support of nanosecond timestamps (KairosDB and OpenTSDB) or the string encoding (`lsof/strace`) in tags (OpenTSDB). Missing precision causes the *split* function to construct wrong results – we ignored this and measured the times nevertheless. String decoding and serialization simply took too long, so we measured the time of the raw data retrieval only.

The online distribution of Chronix holds the code and also the re-implementation of the queries with other time series databases.

Extra queries and client-side evaluations cause a significant slowdown. This can be seen in the lower part of Table 8 where Chronix is faster. But this effect is also visible in the `b`-queries. For instance, InfluxDB needs 343.8 s / (58 · 20) = 0.3 s on average for a raw data `q`-query without evaluating any function at all. Its average for the 43 · 20 `b`-queries instead is only 0.03 s because the function is evaluated server-side and only the result is shipped. This is similar for the other databases. Built-in functions are therefore a clear advantage.

Default values of Chronix. All the results show that even with its default parameters Chronix outperforms the general-purpose time series databases on anomaly detection projects. A use-case specific commissioning with both projects' datasets as input did not change the values for `c` and `t` and did not buy any extra performance.

For the default DDC threshold of `d`=200 ms we see an inaccuracy rate of 20% for both projects. The average deviations are around 42 ms and 80 ms, resp. From our experience, this is acceptable. With the DDC threshold set to the period of the almost-periodic time series, inaccuracy reaches the worst case as only the first timestamp is stored. But even then the resulting materialized compression rate would only be 1.1% lower but for the costs of a high inaccuracy rate.

6 Related Work

We discuss related work along the main requirements of Sec. 2 and the domain specific design decisions of Sec. 3.

Generic data model. We are not aware of any time series database that has such a generic data model as Chronix. Often only scalar values are supported [2, 12, 13, 28, 31, 32, 34, 36]. InfluxDB [24] has also strings and boolean types. KairosDB [25] is extensible but the types lack support for custom operators and functions. As discussed in Section 2, this is too restrictive for the operational data of software systems.

Analysis support. There are indexing approaches for an efficient analysis and retrieval of time series data, e.g., approximation techniques and tree-structured indexes [6, 8, 15, 26]. They optimize an approximate representation of the time series with pointers to the files that contain the raw data for example for a similarity search. In contrast, Chronix is not tailored to a specific analysis but it is optimized for explorative and correlating analyses of operational time series data. Note that the Optional

Transformation stage *can* add indexing values.

Several researchers presented methods that detect anomalies in time series [43, 45, 46]. Chronix implements in its API most of them and also offers plug-ins.

Efficient long term storage. While many time series databases focus on cloud computing and PBytes of general time series data, Chronix is domain specific for the anomaly detection in operational data. Chronix builds upon and extends the architectural principles proposed by Shafer et al. [40] and Dunning et al. [17]. Its strengths and the reasons for the better performance are its pipeline architecture with domain specific optimizations and the commissioning methodology.

Many time series databases are distributed systems that run in separate processes [24, 25, 28, 32] on multiple nodes. Some are affected by synchronization costs and inter-process communication even when configured to run on a single node [25, 28, 32]. In contrast, on a single node Chronix uses lightweight thread parallelism within a process. Moreover, since OpenTSDB and KairosDB use an external storage technology (HBase [23] and Cassandra [3]) with a built-in compression, they cannot save time by shipping compressed data to/from the analysis client [22], whereas Chronix uses the chunk size and the compression technique that not only achieve the best results on the data but also cut down on query latency. OpenTSDB and KairosDB have a large memory footprint due to their building blocks.

In-memory databases or databases that keep large parts of the data in memory, like Gorilla [34] or BTrDB [2], can quickly answer queries on recent data, but they need a long term storage like Chronix for the older data that some anomaly detectors need.

Lossless storage. A few of the related time series databases are not lossless. RRDtool [31], Ganglia [29] (that uses RRDtool), and Graphite [12] store data points in a fixed-size cyclic buffer, called Round Robin Archive (RRA). If there is more data than fits into the buffer, they purge old data. This may cause wrong analysis results. For identification purposes Prometheus [36] uses a 64-bit fingerprint of the attribute values to find data. Potential hash collisions of fingerprints may cause missed data.

Focus on Queries. While most of the databases are optimized for write throughput [2, 12, 13, 25, 31, 32, 34] and suffer from scan and filter operations when data is requested, Chronix optimizes the query performance (mainly by means of a reverse index). This is an advantage for the needs of anomaly detection. Chronix delays costly index reconstructions when a batch import of many small chunks of data is needed.

Chronix processes raw data for aggregations. To optimize such aggregates Chronix can be enriched with techniques of related time series databases to store pre-aggregated values [2, 24].

Domain specific compression. Most time series databases use some form of compression. There are lossy approaches that do not fit the requirements of anomaly detection in operational data. (For instance, one idea is to down-sample and override old data [12, 31].) Many time series databases [2, 25, 28, 32, 34, 36] use a lossless compression. (Tsdb [13] applies QuickLZ [37] and only achieves a mediocre rate of about 20% [13].) Most of them also use a value encoding that is similar in spirit to the DDC. The difference is that Chronix only removes jitter from the timestamps in almost-periodic intervals as exact values matter for anomaly detection.

Commissioning. For none of the other time series databases there is a commissioning methodology to tune it to the domain specific or even use-case specific needs of anomaly detection in operational data.

7 Conclusion

The paper illustrates that general-purpose time series databases impede anomaly detection projects that analyze operational data. Chronix is a domain specific alternative that exploits the specific characteristics of the domain in many ways, e.g., with a generic data model, extensible data types and operators, built-in high-level functions, a novel encoding of almost-periodic time series, records with attributes and binary-encoded chunks of data, domain specific chunk sizes, etc. Since the configuration parameters need to be chosen carefully to achieve an ideal performance, there is also a commissioning methodology to find values for them.

On real-world operational data from industry and on queries that analyze these data, Chronix outperforms general-purpose time series databases by far. With a smaller memory footprint, it saves 20%–68% of the storage space, and it saves 80%–92% on data retrieval time and 73%–97% of the runtime of analyzing functions.

Chronix is open source, see `www.chronix.io`.

Acknowledgements

This research was in part supported by the *Bavarian Ministry of Economic Affairs and Media, Energy and Technology* as an IuK-grant for the project *DfD – Design for Diagnosability*.

We are grateful to the reviewers for their time, effort, and constructive input.

References

All online resources accessed on Sep. 22, 2016.

[1] https://java.net/jira/browse/GRIZZLY-1690.

[2] ANDERSEN, M. P., AND CULLER, D. E. BTrDB: Optimizing Storage System Design for Timeseries Processing. In *USENIX Conf. File and Storage Techn. (FAST)* (Santa Clara, CA, 2016), pp. 39–52.

[3] APACHE CASSANDRA. Manage massive amounts of data. http://cassandra.apache.org.

[4] APACHE LUCENCE. A full-featured text search engine. http://lucene.apache.org.

[5] APACHE SOLR. Open source enterprise search platform. http://lucene.apache.org/solr.

[6] ASSENT, I., KRIEGER, R., AFSCHARI, F., AND SEIDL, T. The TS-tree: Efficient Time Series Search and Retrieval. In *Intl. Conf. Extending database technology: Advances in database technology* (Nantes, France, 2008), pp. 252–263.

[7] BURROWS, M., AND WHEELER, D. A Block-sorting Lossless Data Compression Algorithm. Tech. Rep. 124, Systems Research Center, Palo Alto, CA, 1994.

[8] CAMERRA, A., PALPANAS, T., SHIEH, J., AND KEOGH, E. iSAX 2.0: Indexing and Mining One Billion Time Series. In *Intl. Conf. Data Mining (ICDM)* (Sydney, Australia, 2010), pp. 58–67.

[9] CHAN, K.-P., AND FU, A. W.-C. Efficient Time Series Matching by Wavelets. In *Intl. Conf. Data Eng.* (Sydney, Australia, 1999), pp. 126–133.

[10] COLLECTD. The system statistics collection daemon. https://collectd.org.

[11] COLLET, Y. LZ4 is lossless compression algorithm. http://www.lz4.org.

[12] DAVIS, C. Graphite project – a highly scalable real-time graphing system. https://github.com/graphite-project.

[13] DERI, L., MAINARDI, S., AND FUSCO, F. tsdb: A Compressed Database for Time Series. In *Intl. Conf. Traffic Monitoring and Analysis* (Vienna, Austria, 2012), pp. 143–156.

[14] DEUTSCH, P. GZIP file format specification version 4.3. https://www.ietf.org/rfc/rfc1952.txt, 1996.

[15] DING, H., TRAJCEVSKI, G., SCHEUERMANN, P., WANG, X., AND KEOGH, E. Querying and Mining of Time Series Data: Experimental Comparison of Representations and Distance Measures. In *Very Large Databases Endowment* (Auckland, New Zealand, 2008), pp. 1542–1552.

[16] DOCKER. Docker is the leading software containerization platform. https://www.docker.com.

[17] DUNNING, T., AND FRIEDMAN, E. *Time Series Databases: New Ways to Store and Access Data*. O'Reilly Media, 2014.

[18] ELASTIC. Logstash: Collect, Parse, Transform Logs. https://www.elastic.co/products/logstash.

[19] GOOGLE. Guice. https://github.com/google/guice.

[20] GOOGLE. Protocol buffers – google's data interchange format. https://github.com/google/protobuf.

[21] GOOGLE. Snappy. http://google.github.io/snappy.

[22] GRAEFE, G., AND SHAPIRO, L. Data Compression and Database Performance. In *Symp. Applied Computing* (Kansas City, MO, 1991), pp. 22–27.

[23] HBASE, A. The hadoop database, a distributed, scalable, big data store. http://hbase.apache.org.

[24] INFLUXDATA. InfluxDB: Time-Series Storage. https://influxdata.com/time-series-platform/influxdb.

[25] KAIROSDB. Fast Time Series Database. https://kairosdb.github.io/.

[26] KEOGH, E., CHAKRABARTI, K., PAZZANI, M., AND MEHROTRA, S. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *Knowledge and information Systems 3*, 3 (2001), 263–286.

[27] LAPTEV, N., AMIZADEH, S., AND FLINT, I. Generic and scalable framework for automated time-series anomaly detection. In *Intl. Conf. Knowledge Discovery and Data Mining (KDD)* (Sydney, Australia, 2015), pp. 1939–1947.

[28] LOBOZ, C., SMYL, S., AND NATH, S. DataGarage: Warehousing Massive Amounts of Performance Data on Commodity Servers. In *Very Large Databases Endowment* (Singapore, 2010), pp. 1447–1458.

[29] MASSIE, M. L., CHUN, B. N., AND CULLER, D. E. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing 30*, 7 (2004), 817–840.

[30] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive Analysis of Web-Scale Datasets. In *Very Large Databases Endowment* (Singapore, 2010), pp. 330–339.

[31] OETIKER, T. RRDtool: Data logging and graphing system for time series data. http://oss.oetiker.ch/rrdtool.

[32] OPENTSDB. The Scalable Time Series Database. http://opentsdb.net.

[33] PATEL, P., KEOGH, E., LIN, J., AND LONARDI, S. Mining Motifs in Massive Time Series Databases. In *Intl. Conf. Data Mining* (Maebashi City, Japan, 2002), pp. 370–377.

[34] PELKONEN, T., FRANKLIN, S., TELLER, J., CAVALLARO, P., HUANG, Q., MEZA, J., AND VEERARAGHAVAN, K. Gorilla: A Fast, Scalable, In-Memory Time Series Database. In *Conf. Very Large Databases* (Kohala Coast, HI, 2015), pp. 1816–1827.

[35] POPIVANOV, I., AND MILLER, R. Similarity Search Over Time-Series Data Using Wavelets. In *Intl. Conf. Data Eng.* (San Jose, CA, 2002), pp. 212–221.

[36] PROMETHEUS. Monitoring system and time series database. http://prometheus.io.

[37] QUICKLZ. Fast compression library. http://www.quicklz.com.

[38] SALVADOR, S., AND CHAN, P. FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space. *Intelligent Data Analysis 11*, 5 (2007), 561–580.

[39] SEWARD, J. bzip2. http://www.bzip.org.

[40] SHAFER, I., SAMBASIVAN, R., ROWE, A., AND GANGER, G. Specialized Storage for Big Numeric Time Series. In *USENIX Conf. Hot Topics Storage and File Systems* (San Jose, CA, 2013).

[41] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Symp. Mass Storage Systems and Technologies* (Lake Tahoe, NV, 2010), pp. 1–10.

[42] TUKAANI. Xz. http://tukaani.org/xz.

[43] VALLIS, O., HOCHENBAUM, J., AND KEJARIWAL, A. A Novel Technique for Long-Term Anomaly Detection in the Cloud. In *USENIX Conf. Hot Topics Cloud Computing* (Philadelphia, PA, 2014).

[44] VAN HOORN, A., WALLER, J., AND HASSELBRING, W. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Intl. Conf. Perf. Eng. (ICPE)* (Boston, MA, 2012), pp. 247–248.

[45] WANG, C., VISWANATHAN, K., CHOUDUR, L., TALWAR, V., SATTERFIELD, W., AND SCHWAN, K. Statistical Techniques for Online Anomaly Detection in Data Centers. In *Intl. Symp. Integrated Network Management (IM)* (Dublin, Ireland, 2011), pp. 385–392.

[46] WERT, A., HAPPE, J., AND HAPPE, L. Supporting Swift Reaction: Automatically Uncovering Performance Problems by Systematic Experiments. In *Intl. Conf. Soft. Eng. (ICSE)* (San Francisco, CA, 2013), pp. 552–561.

[47] XU, W., HUANG, L., FOX, A., PATTERSON, D. A., AND JORDAN, M. I. Mining console logs for large-scale system problem detection. In *Conf. Tackling Computer Systems Problems with Machine Learning Techniques* (San Diego, CA, 2008).

Crystal: Software-Defined Storage for Multi-tenant Object Stores

Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López
Universitat Rovira i Virgili (Tarragona, Spain)
{*raul.gracia,josep.sampe,edgar.zamora,marc.sanchez,pedro.garcia*}*@urv.cat*

Yosef Moatti, Eran Rom
IBM Research (Haifa, Israel)
{*moatti,eranr*}*@il.ibm.com*

Abstract

Object stores are becoming pervasive due to their scalability and simplicity. Their broad adoption, however, contrasts with their rigidity for handling heterogeneous workloads and applications with evolving requirements, which prevents the adaptation of the system to such varied needs. In this work, we present *Crystal*, the first Software-Defined Storage (SDS) architecture whose core objective is to efficiently support multi-tenancy in object stores. Crystal adds a filtering abstraction at the data plane and exposes it to the control plane to enable high-level policies at the tenant, container and object granularities. Crystal translates these policies into a set of distributed controllers that can orchestrate filters at the data plane based on real-time workload information. We demonstrate Crystal through two use cases on top of OpenStack Swift: One that proves its storage automation capabilities, and another that differentiates IO bandwidth in a multi-tenant scenario. We show that Crystal is an extensible platform to deploy new SDS services for object stores with small overhead.

1 Introduction

Object stores are becoming an increasingly pervasive storage building block due to their scalability, availability and usage simplicity via HTTP RESTful APIs [1, 8]. These desirable features have spurred the adoption of object stores by many heterogeneous applications and systems, ranging from Personal Clouds [4, 25], Big Data companies such as DataBricks [2] and Mirantis [6] and analytics frameworks [22, 17], and Web applications [19], to name a few.

Despite their growing popularity, object stores are not well prepared for heterogeneity. Typically, a deployment of an object store in the cloud uses a monolithic configuration setup, even when the same object store acts as a substrate for different types of applications with time-varying requirements [17, 16]. This results in all applications experiencing the same service level, though the workloads from different applications can vary dramatically. For example, while a social network application such as Facebook would have to store a large number of small-medium sized photos (KB- to MB-sized objects), a Big Data analytics framework would probably generate read and write requests for large files. It is clear that using a static configuration *inhibits optimization of the system* to such varying needs.

But not only this; beyond the particular needs of a type of workload, the requirements of applications can also vary greatly. For example, an archival application may require of transparent compression, annotation, and encryption of the archived data. In contrast, a Big Data analytics application may benefit from the computational resources of the object store to eliminate data movement and enable in-place analytics capabilities [22, 17]. Supporting such a variety of requirements in an object store is challenging, because in current systems, custom functionality is hard-coded into the system implementation due to the *absence of a true programmable layer*, making it difficult to maintain as the system evolves.

1.1 Scope and Challenges

In this paper, we argue that Software-Defined Storage (SDS) is a compelling solution to these problems. As in SDN, the separation of the "data plane" from the "control plane" is the best-known principle in SDS [41, 34, 39, 23, 38]. Such separation of concerns is the cornerstone of supporting heterogeneous applications in data centers. However, the application of SDS fundamentals on cloud object stores is not trivial. Among other things, it needs to address two main challenges:

A flexible control plane. The control plane should be the key enabler that makes it possible to support multiple applications separately using dynamically configurable functionalities. Since the *de facto* way of expressing management requirements and objectives in SDS is via

policies, they should also dictate the management rules for the different tenants in a shared object store. This is not easy since policies can be very distinct. They can be as simple as a calculation on an object such as compression, and as complex as the distributed enforcement of per-tenant IO bandwidth limits. Further, as a singular attribution of object storage, such policies have to express objectives and management rules at the tenant, container and object granularities, which requires of a largely distinct form of policy translation into the data plane compared with prior work [41, 39, 38]. Identifying the necessary abstractions to concisely define the management policies is not enough. If the system evolves over time, the control plane should be flexible enough to properly describe the new application needs in the policies.

An extensible data plane. Although the controller in all SDS systems is assumed to be easy to extend [41, 39, 38], data plane extensibility must be significantly richer for object storage; for instance, it must enable to perform "on the fly" computations as the objects arrive and depart from the system to support application-specific functions like sanitization, Extract-Transform-Load (ETL) operations, caching, etc. This entails the implementation of a lightweight, yet versatile computing layer, which do not exist today in SDS systems. Building up an extensible data plane is challenging. On the one hand, it requires of new abstractions that enable policies to be succinctly expressed. On the other hand, these abstractions need to be flexible enough to handle heterogeneous requirements, that is, from resource management to simple automation, which is not trivial to realize.

1.2 Contributions

To overcome the rigidity of object stores we present *Crystal*: The first SDS architecture for object storage to efficiently support multi-tenancy and heterogeneous applications with evolving requirements. Crystal achieves this by separating policies from implementation and unifying an extensible data plane with a logically centralized controller. As a result, Crystal allows to dynamically adapt the system to the needs of specific applications, tenants and workloads.

Of Crystal, we highlight two aspects, though it has other assets. First, Crystal presents an extensible architecture that unifies individual models for each type of resource and transformation on data. For instance, global control on a resource such as IO bandwidth can be easily incorporated as a small piece of code. A dynamic management policy like this is materialized in form of a distributed, supervised *controller*, which is the Crystal abstraction that enables the addition of new control algorithms (Section 5.2). In particular, these controllers, which are deployable at runtime, can be fed with pluggable per-workflow or resource *metrics*. Examples of metrics are the number of IO operations per second and the bandwidth usage. An interesting property of Crystal is that it can even use object metadata to better drive the system towards the specified objectives.

Second, Crystal's data plane abstracts the complexity of individual models for resources and computations through the *filter* abstraction. A filter is a piece of programming logic that can be injected into the data plane to perform custom calculations on object requests. Crystal offers a filter framework that enables the deployment and execution of general computations on objects and groups of objects. For instance, it permits the pipelining of several actions on the same object(s) similar to stream processing frameworks [30]. Consequently, practitioners and systems developers only need to focus on the development of storage filters, as their deployment and execution is done transparently by the system (Section 5.1). To our knowledge, no previous SDS system offers such a computational layer to act on resources and data.

We evaluate the design principles of Crystal by implementing two use cases on top of OpenStack Swift: One that demonstrates the automation capabilities of Crystal, and another that enforces IO bandwidth limits in a multi-tenant scenario. These uses cases demonstrate the feasibility and extensibility of Crystal's design. The experiments with real workloads and benchmarks are run on a 13-machine cluster. Our experiments reveal that policies help to overcome the rigidity of object stores incurring small overhead. Also, defining the right policies may report performance and cost benefits to the system.

In summary, our key contributions are:

- Design of Crystal, the first SDS architecture for object storage that efficiently supports multi-tenancy and applications with evolving requirements;
- A control plane for multi-tenant object storage, with flexible policies and their transparent translation into the enforcement mechanisms at the data plane;
- An extensible data plane that offers a *filter* abstraction, which can encapsulate from arbitrary computations to resource management functionality, enabling concise policies for complex tasks;
- The implementation and deployment of policies for storage automation and IO bandwidth control that demonstrate the design principles of Crystal.

2 Crystal Design

Crystal seeks to efficiently handle workload heterogeneity and applications with evolving requirements in shared object storage. To achieve this, Crystal separates high-level policies from the mechanisms that implement them at the data plane, to avoid hard-coding the policies in the

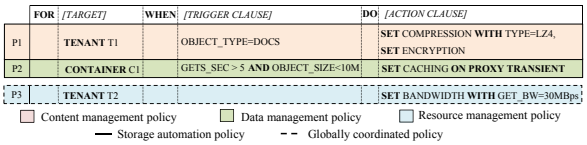


Figure 1: Structure of the Crystal DSL.

system itself. To do so, it uses three abstractions: *filter*, *metric*, and *controller*, in addition to *policies*.

2.1 Abstractions in Crystal

Filter. A filter is a piece of code that a system administrator can inject into the data plane to perform custom computations on incoming object requests[1]. In Crystal, this concept is broad enough to include *computations on object contents* (e.g., compression, encryption), *data management* like caching or pre-fetching, and even *resource management* such as bandwidth differentiation (Fig. 1). A key feature of filters is that the instrumented system is oblivious to their execution and needs no modification to its implementation code to support them.

Inspection trigger. This abstraction represents information accrued from the system to automate the execution of filters. There are two types of information sources. A first type that corresponds to the *real-time metrics* got from the running workloads, like the number of `GET` operations per second of a data container or the IO bandwidth allocated to a tenant. As with filters, a fundamental feature of workload metrics is that they can be deployed at runtime. A second type of source is the *metadata from the objects* themselves. Such metadata is typically associated with read and write requests and includes properties like the size or type of objects.

Controller. In Crystal, a controller represents an algorithm that manages the behavior of the data plane based on monitoring metrics. A controller may contain a *simple rule to automate* the execution of a filter, or a complex algorithm requiring *global visibility* of the cluster to control a filter's execution under multi-tenancy. Crystal builds a logically centralized control plane formed by supervised and distributed controllers. This allows an administrator to easily deploy new controllers on-the-fly that cope with the requirements of new applications.

Policy. Our policies should be extensible for really allowing the system to satisfy evolving requirements. This means that the structure of policies must facilitate the incorporation of new filters, triggers and controllers.

To succinctly express policies, Crystal abides by a structure similar to that of the popular IFTTT (If-This-Then-That) service [5]. This service allows users to express small rule-based programs, called "recipes", using *triggers* and *actions*. For example:

[1]Filters work in an *online* fashion. To explore the feasibility of batch filters on already stored objects is matter of future work.

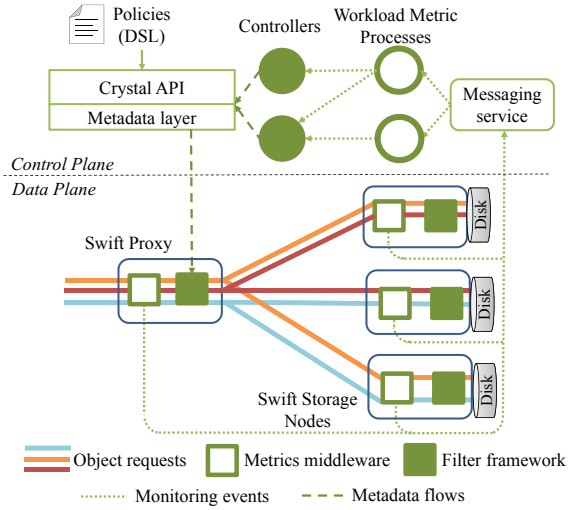


Figure 2: High-level overview of Crystal's architecture materialized on top of OpenStack Swift.

`TRIGGER`: compressibility of an object is $> 50\%$
`ACTION`: compress
`RECIPE: IF` compressibility is $> 50\%$ `THEN` compress

An IFTTT-like language can reflect the extensibility capabilities of the SDS system; at the data plane, we can infer that triggers and actions are translated into our inspection triggers and filters, respectively. At the control plane, a policy is a "recipe" that guides the behavior of control algorithms. Such apparently simple policy structure can express different policy types. On the one hand, Fig. 1 shows *storage automation policies* that enforce a filter either statically or dynamically based on simple rules; for instance, `P1` enforces compression and encryption on document objects of tenant `T1`, whereas `P2` applies data caching on small objects of container `C1` when the number of `GET`s/second is > 5. On the other hand, such policies can also express objectives to be achieved by controllers requiring *global visibility and coordination* capabilities of the data plane. That is, `P3` tells a controller to provide at least 30MBps of aggregated `GET` bandwidth to tenant `T2` under a multi-tenant workload.

2.2 System Architecture

Fig. 2 presents Crystal's architecture, which consists of:

Control Plane. Crystal provides administrators with a system-agnostic DSL (Domain-Specific Language) to define SDS services via high-level policies. The DSL "vocabulary" can be extended at runtime with new filters and inspection triggers. The control plane includes an API to compile policies and to manage the life-cycle and metadata of controllers, filters and metrics (see Table 1).

Moreover, the control plane is built upon a distributed model. Although logically centralized, the controller is, in practice, split into a set of autonomous micro-services,

each running a separate control algorithm. Other micro-services, called workload metric processes, close the control loop by exposing monitoring information from the data plane to controllers. The control loop is also extensible, given that both controllers and workload metric processes can be deployed at runtime.

Data Plane. Crystal's data plane has two core extension points: Inspection triggers and filters. First, a developer can deploy new workload metrics at the data plane to feed distributed controllers with new runtime information on the system. The metrics framework runs the code of metrics and publishes monitoring events to the messaging service. Second, data plane programmability and extensibility is delivered through the filter framework, which intercepts object flows in a transparent manner and runs computations on them. A developer integrating a new filter only needs to contribute the logic; the deployment and execution of the filter is managed by Crystal.

3 Control Plane

The control plane allows writing policies that adapt the data plane to manage multi-tenant workloads. It is formed by the DSL, the API and distributed controllers.

3.1 Crystal DSL

Crystal's DSL hides the complexity of low-level policy enforcement, thus achieving simplified storage administration (Fig. 1). The structure of our DSL is as follows:

Target: The target of a policy represents the recipient of a policy's action (e.g., filter enforcement) and it is mandatory to specify it on every policy definition. To meet the specific needs of object storage, targets can be *tenants*, *containers* or even individual *data objects*. This enables high management and administration flexibility.

Trigger clause (optional): Dynamic storage automation policies are characterized by the trigger clause. A policy may have one or more trigger clauses —separated by AND/OR operands— that specify the workload-based situation that will trigger the enforcement of a filter on the target. Trigger clauses consist of inspection triggers, operands (e.g, $>$, $<$, $=$) and values. The DSL exposes both types of inspection triggers: workload metrics (e.g., GETS_SEC) and request metadata (e.g., OBJECT_SIZE<512).

Action clause: The action clause of a policy defines how a filter should be executed on an object request once the policy takes place. The action clause may accept parameters after the WITH keyword in form of key/value pairs that will be passed as input to customize the filter execution. Retaking the example of a compression filter, we may decide to enforce compression using a gzip or an lz4 engine, and even their compression level.

Crystal Controller Calls	*Description*
add_policy delete_policy list_policies	Policy management API calls. For storage automation policies, the add_policy call can either to directly enforce the filter or to deploy a controller to do so. For globally coordinated policies, the call sets an objective at the metadata layer.
register_keyword delete_keyword	Calls that interact with Crystal registry to associate DSL keywords with filters, inspection triggers or coin new terms to be used as trigger conditions (e.g., DOCS).
deploy_controller kill_controller	These calls are used to manage the life-cycle of distributed controllers and workload metric processes in the system.
Filter Framework Calls	*Description*
deploy_filter undeploy_filter list_filters	Calls for deploying, undeploying and listing filters associated to a target. deploy/undeploy_filter calls interact with the filter framework at the data plane for enabling/disabling filter binaries to be executed on a specific target.
update_slo list_slo delete_slo	Calls to manage "tenant objectives" for coordinated resource management filters. For instance, bandwidth differentiation controllers take as input this information in order to provide an aggregated IO bandwidth share at the data plane.
Workload Metric Calls	*Description*
deploy_metric delete_metric	Calls for managing workload metrics at the data plane. These calls also manage workload metric processes to expose data plane metrics to the control plane.

*For the sake of simplicity, we do not include call parameters in this table.

Table 1: Main calls of Crystal controller, filter framework and workload metrics management APIs.

To cope with object stores formed by proxies/storage nodes (e.g., Swift), our DSL enables to explicitly control the execution stage of a filter with the ON keyword. Also, dynamic storage automation policies can be *persistent or transient*; a persistent action means that once the policy is triggered the filter enforcement remains indefinitely (by default), whereas actions to be executed only during the period where the condition is satisfied are transient (keyword TRANSIENT, P2 in Fig. 1).

The vocabulary of our DSL can be extended on-the-fly to accommodate new filters and inspection triggers. That is, in Fig. 1 we can use keywords COMPRESSION and DOCS in P1 once we associate "COMPRESSION" with a given filter implementation and "DOCS" with some file extensions, respectively (see Table 1).

The Crystal DSL has other features: i) *specialization* of policies based on the target scope, so that if several policies apply to the same request, only the most specific one is executed (e.g., container-level policy is more specific than a tenant-level one), ii) *pipelining* several filters on a single request (e.g., compression + encryption) ordered as they are defined in the policy, similar to stream processing frameworks [30], and iii) *grouping*, which enables to enforce a single policy to a group of targets; that is, we can create a group like WEB_CONTAINERS to represent all the containers that serve Web pages.

As visible in Table 1, Crystal offers a DSL compilation service via API calls. Crystal compiles simple automation policies as *target→filter* relationships at the metadata layer. Next, we show how dynamic policies (i.e., with WHEN clause) use controllers to enforce filters.

3.2 Distributed Controllers

Crystal resorts to distributed controllers, in form of supervised micro-services, which can be deployed in the system at runtime to extend the control plane [15, 18, 40].

We offer two types of controllers: *automation* and *global* controllers. On the one hand, the Crystal DSL compiles dynamic storage automation policies into au-

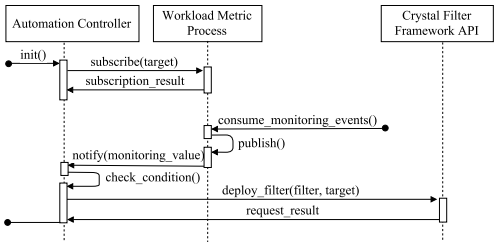


Figure 3: Interactions among automation controllers, workload metric processes and the filter framework.

tomation controllers (e.g., `P2` in Fig. 1). Their life-cycle consists of consuming the appropriate monitoring metrics and interact with the filter framework API to enforce a filter when the trigger clause is satisfied.

On the other hand, global controllers are not generated by the DSL; instead, by simply extending a base class and overriding its `computeAssignments` method, developers can deploy controllers that contain complex algorithms with global visibility and continuous control of a filter at the data plane (e.g., `P3` in Fig. 1). To this end, the base global controller class encapsulates the logic i) to ingest monitoring events, ii) to disseminate the computed assignments across nodes[2], and iii) to get Service-Level Objectives (SLO) to be enforced from the metadata layer (see Table 1). This allowed us to deploy distributed IO bandwidth control algorithms (Section 5).

Extensible control loop: To close the control loop, *workload metric processes* are micro-services that provide controllers with monitoring information from the data plane. While running, a workload metric process consumes and aggregates events from one workload metric at the data plane. For the sake of simplicity [40], we advocate to separate workload metrics not only per metric type, but also by target granularity.

Controllers and workload metrics processes interact in a publish/subscribe fashion [21]. For instance, Fig. 3 shows that, once initialized, an automation controller subscribes to the appropriate workload metric process, taking into account the target granularity. The subscription request of a controller specifies the target to which it is interested in, such as tenant `T1` or container `C1`; this ensures that controllers do not receive unnecessary monitoring information from other targets. Once the workload metric process receives the subscription request, it adds the controller to its observer list. Periodically, it notifies the activity of the different targets to the interested controllers that may trigger the execution of filters.

4 Data Plane

At the data plane, we offer two main extension hooks: Inspection triggers and a filter framework.

[2]For efficiency reasons, global controllers disseminate assignments to data plane filters also via the messaging service.

4.1 Inspection Triggers

Inspection triggers enable controllers to dynamically respond to workload changes in real time. Specifically, we consider two types of introspective information sources: *object metadata* and *monitoring metrics*.

First, some object requests embed semantic information related to the object at hand in form of metadata. Crystal enables administrators to enforce storage filters based on such metadata. Concretely, our filter framework middleware (see Section 4.2) is capable of analyzing at runtime HTTP metadata of object requests to execute filters based on the object size or file type, among others.

Second, Crystal builds a metrics middleware to add new workload metrics on the fly. At the data plane, a workload metric is a piece of code that accounts for a particular aspect of the system operation and publishes that information. In our design, a new workload metric can inject events to the monitoring service without interfering with existing ones (Table 1). Our metrics framework allows developers to plug-in metrics that inspect both the type of requests and their contents (e.g., compressibility [29]). We provide the logic (i.e., `AbstractMetric` class) to abstract developers from the complexity of request interception and event publishing.

4.2 Filter Framework

The Crystal filter framework enables developers to deploy and run general-purpose code on object requests. Crystal borrows ideas from active storage literature [36, 35] as a mean of building filters to enforce policies.

Our framework achieves flexible execution of filters. First, it enables to easily *pipeline several filters* on a single storage request. Currently, the execution order of filters is set explicitly by the administrator, although filter metadata can be exploited to avoid conflicting filter ordering errors [20]. Second, to deal with object stores composed by proxies/storage nodes, Crystal allows administrators to define the *execution point of a filter*.

To this end, the Crystal filter framework consists of i) a filter middleware, and ii) filter execution environments.

Filter middleware: Our filter middleware intercepts data streams and classifies incoming requests. Upon a new object request, the middleware at the proxy performs a single metadata request to infer the filters to be executed on that request depending on the target. If the target has associated filters, the filter middleware sets the appropriate metadata headers in the request for triggering the execution of filters through the read/write path.

Filters that change the content of data objects may receive a special treatment (e.g., compression, encryption). To wit, if we create a filter with the *reverse* flag enabled, it means that the execution of the filter when the object was stored should be always undone upon a `GET` request.

That is, this yields that we may activate data compression on certain periods, but tenants will always download decompressed objects. To this end, prior to storing an object, we tag it with *extended metadata* that keeps track of the enforced filters with reverse flag set. Upon a `GET` request, the filter middleware fetches such metadata from the object itself to trigger the reverse transformations on it prior to the execution of regular filters.

Filter execution environments: Currently, our middleware can support two filter execution environments:

Isolated filter execution: Crystal provides an isolated filter execution environment to perform general-purpose computations on object streams with high security guarantees. To this end, we extended the Storlets framework [7] with pipelining and stage execution control functionalities. Storlets provide Swift with the capability to run computations close to the data in a secure and isolated manner making use of `Docker` containers [3]. Invoking a Storlet on a data object is done in an isolated manner so that the data accessible by the computation is only the object's data and its user metadata. Also, a `Docker` container only runs filters of a single tenant.

Native filter execution: The isolated filter execution environment trades-off higher security for lower communication capabilities and interception flexibility. For this reason, we also contribute an alternative way to intercept and execute code natively. As with Storlets, a developer can deploy code modules as native filters at runtime by following simple implementation guidelines. However, native filters can i) execute code at all the possible points of a request's life-cycle, and ii) communicate with external components (e.g, metadata layer), as well as to access storage devices (e.g., SSD). As Crystal is devised to execute trusted code from administrators, this environment represents a more flexible alternative.

5 Hands On: Extending Crystal

Next, we show the benefits of Crystal's design by extending the system with data management filters and distributed control of IO bandwidth for OpenStack Swift.

5.1 New Storage Automation Policies

Goal: To define policies that enforce filters, like *compression*, *encryption* or *caching*, even dynamically:

```
P1:FOR TENANT T1 WHEN OBJECT_TYPE=DOCS DO SET
COMPRESSION ON PROXY, SET ENCRYPTION ON STORAGE_NODE

P2:FOR CONTAINER C1 WHEN GETS_SEC > 5 DO SET CACHING
```

Data plane (Filters): To enable such storage automation policies, we first need to *develop the filters* at the data plane. In Crystal this can be done using either native or isolated execution environments.

The next code snippet shows how to develop a filter for our isolated execution environment. A system developer only needs to create a class that implements an interface (`IStorlet`), providing the actual data transformations on the object request streams (`iStream`, `oStream`) inside the `invoke` method. To wit, we implemented the compression (`gzip` engine) and encryption (AES-256) filters using storlets, whereas the caching filter exploits SSD drives at proxies via our native execution environment. Then, once these filters were developed, we installed them via the Crystal filter framework API.

```
public class StorletName implements IStorlet {

 @Override
 public void invoke(ArrayList<StorletInputStream> iStream,
   ArrayList<StorletOutputStream> oStream,
   Map<String, String> parameters, StorletLogger logger)
   throws StorletException {
      //Develop filter logic here
 }
}
```

Data plane (Monitoring): Via the Crystal API (see Table 1), we deployed metrics that capture various workload aspects (e.g., `PUTs`/`GETs` per second of a tenant) to satisfy policies like `P2`. Similarly, we deployed the corresponding workload metrics processes (one per metric and target granularity) that aggregate such monitoring information to be published to controllers. Also, our filter framework middleware is already capable of enforcing filters based on object metadata, such as object size (`OBJECT_SIZE`) and type (`OBJECT_TYPE`).

Control Plane: Finally, we registered intuitive keywords for both filters and workload metrics at the metadata layer (e.g., `CACHING`, `GET_SEC_TENANT`) using the Crystal registry API. To achieve `P1`, we also registered the keyword `DOCS`, which contains the file extensions of common documents (e.g, `.pdf`, `.doc`). At this point, we can use such keywords in our DSL to design new storage policies.

5.2 Global Management of IO Bandwidth

Goal: To provide Crystal with means of defining policies that enforce a global IO bandwidth SLO on `GETs`/`PUTs`:

```
P3:FOR TENANT T1 DO SET BANDWIDTH WITH GET_BW=30MBps
```

Data plane (Filter). To achieve global bandwidth SLOs on targets, we first need to locally control the bandwidth of object requests. Intuitively, bandwidth control in Swift may be performed at the proxy or storage node stages. At the proxy level this task may be simpler, as fewer nodes should be coordinated. However, this approach is agnostic to the background tasks (e.g., replication) executed by storage nodes, which impact on performance [33]. We implemented a native bandwidth control filter that enables the enforcement at both stages.

Our filter dynamically creates threads that serve and control the bandwidth allocation for individual tenants,

Algorithm 1 `computeAssignments` pseudo-code embedded into a bandwidth differentiation controller

```
1:  function COMPUTEASSIGNMENTS(info):
2:              /* Retrieve the defined tenant SLOs from the metadata layer */
3:      SLOs ← getMetadataStoreSLOs();
4:        /* Compute assignments on current tenant transfers to meet SLOs */
5:      SLOAssignments ← minSLO(info, SLOs);
6:    /* Estimate spare bw at proxies/storage nodes based on current usage */
7:      spareBw ← min(spareBwProxies(SLOAssignments), spareBwStor-
    ageNodes(SLOAssignments));
8:      spareBwSLOs ← {};
9:                /* Distribute spare bandwidth equally across all tenants */
10:     for tenant in info do
11:         spareBwSLOs[tenant] ← spareBW / numTenants(info);
12:     end for
13:       /* Calculate assignments to achieve spare bw shares for tenants */
14:     spareAssignments ← spareSLO(SLOAssignments, spareBwSLOs);
15:               /* Combine SLO and spare bw assignments on tenants */
16:     return SLOAssignments ∪ spareAssignments;
17: end function
```

either at proxies or storage nodes. Our filter garbage-collects control threads that are inactive for a certain timeout. Moreover, it has a consumer process that receives bandwidth assignments from a controller to be enforced on a tenant's object streams. Once the consumer receives a new event, it propagates the assignments to the filter that immediately take effect on current transfers.

Data plane (Monitoring): For building the control loop, our bandwidth control service integrates individual monitoring metrics per type of traffic (i.e., `GET`, `PUT`, `REPLICATION`); this makes it possible to define policies for each type of traffic if needed. In essence, monitoring events contain a data structure that represents the bandwidth share that tenants exhibit at proxies or per storage node disk. We also deployed workload metric processes to expose these events to controllers.

Control plane. We deployed Algorithm 1 as a global controller to orchestrate our bandwidth differentiation filter. Concretely, we aim at satisfying three main requirements: i) A *minimum bandwidth share per tenant*, ii) *Work-conservation* (do not leave idle resources), and iii) *Equal shares of spare bandwidth* across tenants. The challenge is to meet these requirements considering that we do not control neither the data access of tenants nor the data layout of Swift [28, 44].

To this end, Algorithm 1 works in three stages. First, the algorithm tries to ensure the SLO for tenants specified in the metadata layer by resorting to function `minSLO` (requirement 1, line 6). Essentially, `minSLO` first assigns a proportional bandwidth share to tenants with guaranteed bandwidth. Note that such assignment is done in descending order based on the number of parallel transfers per tenant, provided that tenants with fewer transfers have fewer opportunities of meeting their SLOs. Moreover, `minSLO` checks whether there exist overloaded nodes in the system. In the affirmative case, the algorithm tries to reallocate bandwidth of tenants with multiple transfers from overloaded nodes to idle ones. In case that no reallocation is possible, the algorithm reduces the bandwidth share of tenants with SLOs on overloaded nodes.

In second place, once Algorithm 1 has calculated the assignments for tenants with SLOs, it estimates the spare bandwidth available to achieve full utilization of the cluster (requirement 2, line 8). Note that the notion of spare bandwidth depends on the cluster at hand, as the bottleneck may be either at the proxies or storage nodes.

Algorithm 1 builds a new assignment data structure in which the spare bandwidth is equally assigned to all tenants. The algorithm proceeds by calling function `spareSLO` to calculate the spare bandwidth assignments (requirement 3, line 15). Note that `spareSLO` receives the `SLOAssignments` data structure that keeps the already reserved node bandwidth according to the SLO tenant assignments. The algorithm outputs the combination of SLO and spare bandwidth assignments per tenant. While more complex algorithms can be deployed in Crystal [27], our goal in Algorithm 1 is to offer an attractive simplicity/effectiveness trade-off, validating our bandwidth differentiation framework.

6 Prototype Implementation

We tested our prototype in OpenStack Kilo version. The Crystal API is implemented with the Django framework. The API manages the system's metadata from Redis 3.0 in-memory store [10]. We found that co-locating both Redis and the Swift proxies in the same servers is a suitable deployment strategy. As we show next, this is specially true as only the filter middleware in proxies accesses the metadata layer (once per request).

We resort to PyActive [46] for building distributed controllers and workload metric processes that can communicate among them (e.g., TCP, message brokers). For fault tolerance, the PyActive supervisor is aware of all the instantiated remote micro-services (either at one or many servers) and can spawn a new process if one dies.

We built our metrics and filter frameworks as standard WSGI middlewares in Swift. The code of workload metrics is dynamically deployed on Swift nodes, intercepts the requests and periodically publishes monitoring information (e.g., 1 second) via RabbitMQ 3.6 message broker. Similarly, the filter framework middleware intercepts a storage request and redirects it via a pipe either to the Storlets engine or to a native filter, depending on the filter pipeline definition. As both filters and metrics can run on all Swift nodes, in the case of server failures they can be executed in other servers holding object replicas.

The code of Crystal is publicly available[3] and our contributions to the Storlets project are submitted for acceptance to the official OpenStack repository.

[3] `https://github.com/Crystal-SDS`

7 Evaluation

Next, we evaluate a prototype of Crystal for OpenStack Swift in terms of flexibility, performance and overhead.

Objectives: Our evaluation addresses the challenges of Section 1.1 by showing: i) Crystal can define policies at multiple granularities, achieving administration flexibility; ii) The enforcement of storage automation filters can be dynamically triggered based on workload conditions; iii) Crystal achieves accurate distributed enforcement of IO bandwidth SLOs on different tenants; iv) Finally, Crystal has low execution/monitoring overhead.

Workloads: We resort to well-known benchmarks and replays of real workload traces. First, we use `ssbench` [11] to execute stress-like workloads on Swift. `ssbench` provides flexibility regarding the type (CRUD) and number of operations to be executed, as well as the size of files generated. All these parameters can be specified in form of configuration "scenarios".

To evaluate Crystal under real-world object storage workloads, we collected the following traces[4]: ii) The first trace captures 1.28TB of a write-dominated (79.99% write bytes) document database workload storing 817K car testing/standardization files (mean object size is 0.91MB) for 2.6 years at Idiada; an automotive company. i) The second trace captures 2.97TB of a read-dominated (99.97% read bytes) Web workload consisting of requests related to 228K small data objects (mean object size is 0.28MB) from several Web pages hosted at Arctur datacenter for 1 month. We developed our own workload generator to replay a part of these traces (12 hours), as well as to perform experiments with controllable rates of requests. Our workload generator resorts to SDGen [24] to create realistic contents for data objects based on the file types described in the workload traces.

Platform: We ran our experiments in a 13-machine cluster formed by 9 Dell PowerEdge 320 nodes (Intel Xeon E5-2403 processors); 2 of them act as Swift proxy nodes (28GB RAM, 1TB HDD, 500GB SSD) and the rest are Swift storage nodes (16GB RAM, 2x1TB HDD). There are 3 Dell PowerEdge 420 (32GB RAM, 1TB HDD) nodes that were used as compute nodes to execute workloads. Also, there is 1 large node that runs the OpenStack services and the Crystal control plane (i.e., API, controllers, messaging, metadata store). Nodes in the cluster are connected via 1 GbE switched links.

7.1 Evaluating Storage Automation

Next, we present a battery of experiments that demonstrate the feasibility and capabilities of storage automation with Crystal. To this end, we make use of synthetic workloads and real trace replays (Idiada, Arctur). These

[4]Available at `http://iostack.eu/datasets-menu`.

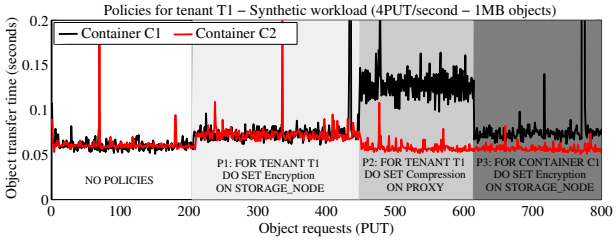


Figure 4: Enforcement of compression/encryption filters.

experiments have been executed at the compute nodes against 1 swift proxy and 6 storage nodes.

Storage management capabilities of Crystal. Fig. 4 shows the execution of several storage automation policies on a workload related to containers `C1` and `C2` belonging to tenant `T1`. Specifically, we executed a write-only synthetic workload (4PUT/second of 1MB objects) in which data objects stored at `C1` consist of random data, whereas `C2` stores highly redundant objects.

Due to the security requirements of `T1`, the first policy defined by the administrator is to encrypt his data objects (`P1`). Fig. 4 shows that the `PUT` operations of *both containers* exhibit a slight extra overhead due to encryption, given that the policy has been defined at the tenant scope. There are two important aspects to note from `P1`: First, the execution of encryption on `T1`'s requests is isolated from filter executions of other tenants, providing higher security guarantees [7] (Storlet filter). Second, the administrator has the ability to enforce the filter at the storage node in order to do not overload the proxy with the overhead of encrypting data objects (`ON` keyword).

After policy `P1` was enforced, the administrator decided to optimize the storage space of `T1`'s objects by enforcing compression (`P2`). `P2` also enforces compression at the proxy node to minimize communication between the proxy and storage node (`ON PROXY`). Note that the enforcement of `P1` and `P2` demonstrates the filter pipelining capabilities of our filter framework; once `P2` is defined, Crystal enforces compression at the proxy node and encryption at storage nodes for each object request. Also, as shown in Section 4, the filter framework tags objects with extended metadata to trigger the reverse execution of these filters on `GET` requests (i.e., decryption and decompression, in that order).

However, the administrator realized that the compression filter on `C1`'s requests exhibited higher latency and provided no storage space savings (incompressible data). To overcome this issue, the administrator defined a new policy `P3` that essentially enforces only encryption on `C1`'s requests. After defining `P3`, the performance of `C1`'s requests exhibits the same behavior as before the enforcement of `P2`. Thus, the administrator is able to manage storage at different granularities, such as tenant or container. Furthermore, the last policy also proves the usefulness of policy specialization; policies `P1` and `P2`

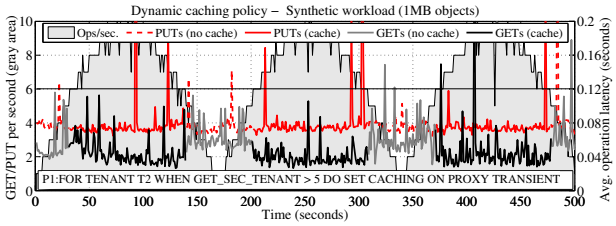


Figure 5: Dynamic enforcement of caching filter.

apply to C2 at the tenant scope, whereas the system only executes P3 on C1's requests, as it is the most specialized policy.

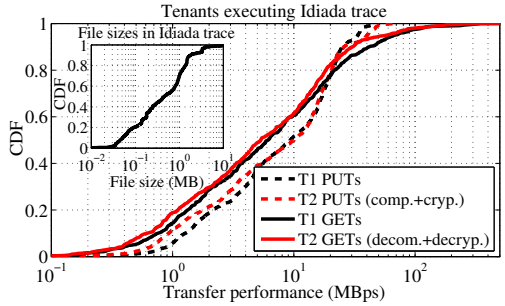
Dynamic storage automation. Fig. 5 shows a dynamic caching policy (P1) on one tenant. The filter implements LRU eviction and exploits SSD drives at the proxy to improve object retrievals. We executed a synthetic oscillatory workload of 1MB objects (gray area) to verify the correct operation of automation controllers.

In Fig. 5, we show the average latency of PUT/GET requests and the intensity of the workload. As can be observed, the caching filter takes place when the workload exceeds 5 GETs per second. At this point, the filter starts caching objects at the proxy SSD on PUTs, as well as to lookup the SSD to retrieve potentially cached objects on GETs. First, the filter provides performance benefits for object retrievals; when the caching filter is activated, object retrievals are in median 29.7% faster compared to non-caching periods. Second, we noted that the costs of executing asynchronous writes on the SSD upon PUT requests may be amortized by offloading storage nodes; that is, the average PUT latency is in median 2% lower when caching is activated. A reason for this may be that storage nodes are mostly free to execute writes, as a large fraction of GETs are being served at the proxy's cache.

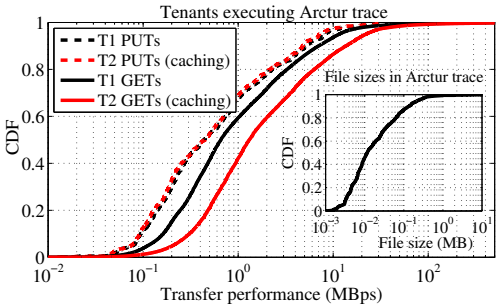
In conclusion, Crystal's control loop enables dynamic enforcement of storage filters under variable workloads. Moreover, native filters in Crystal allow developers to build complex data management filters.

Managing real workloads. Next, we show how Crystal policies can handle real workloads (12 hours). That is, we compress and encrypt documents (P1 in Fig. 1) on a replay of the Idiada trace (write-dominated), whereas we enforce caching of small files (P2 in Fig. 1) on a replay of Arctur workload (read-dominated).

Fig. 6(a) shows the request bandwidth exhibited during the execution of the Idiada trace. Concretely, we executed two concurrent workloads, each associated to a different tenant. We enforced compression and encryption only on tenant T2. Observably, tenant T2's transfers are over 13% and 7% slower compared to T1 for GETs and PUTs, respectively. This is due to the computation overhead of enforcing filters on T2's document objects. As a result, T2's documents consumed 65% less space compared to T1 with compression and they benefited from



(a) Idiada workload (file sizes in inner plot).



(b) Arctur workload (file sizes in inner plot).

Figure 6: Policy enforcement on real trace replays.

higher data confidentially thanks to encryption.

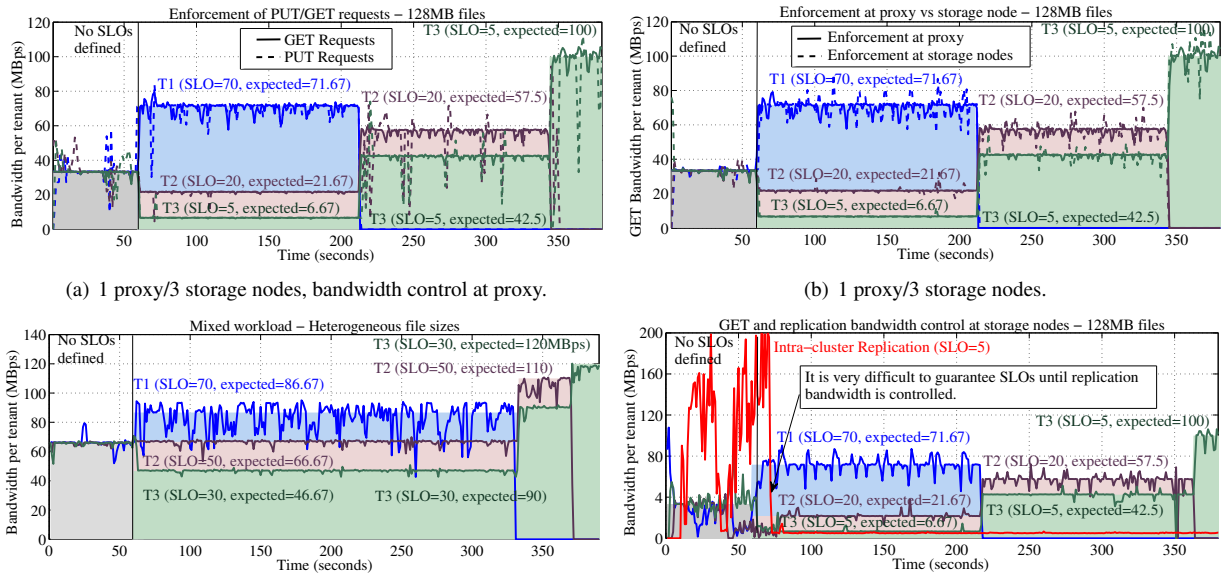
Fig. 6(b) shows tenants T1 and T2, both concurrently running a trace replay of Arctur. By executing a dynamic caching policy, T2's GET requests are in median 1.9x faster compared to T1. That is, as the workload of Arctur is intense and almost read-only, caching was enabled for tenant T2 for most of the experiment. Moreover, because the requested files fitted in the cache, the SSD-based caching filter was very beneficial to tenant T2. The median write overhead of T2 compared to T1 was 4.2%, which suggests that our filter efficiently intercepts object streams for doing parallel writes at the SSD.

Our results with real workloads suggest that Crystal is practical for managing multi-tenant object stores.

7.2 Achieving Bandwidth SLOs

Next, we evaluate the effectiveness of our bandwidth differentiation filter. To this end, we executed a ssbench workload (10 concurrent threads) in each of the 3 compute nodes in our cluster, one of each representing an individual tenant. As we study the effects of replication separately (in Fig. 7(d) we use 3 replicas), the rest of experiments were performed using one replica rings.

Request types. Fig. 7(a) plots two different SLO enforcement experiments on three different tenants for PUT and GET requests, respectively (enforcement at proxy node). Appreciably, the execution of Algorithm 1 exhibits a near exact behavior for both PUT and GET requests. Moreover, we observe that tenants obtain their



(a) 1 proxy/3 storage nodes, bandwidth control at proxy.

(b) 1 proxy/3 storage nodes.

(c) 2 proxy/6 storage nodes, bandwidth control at proxies.

(d) 1 proxy/3 storage nodes, bandwidth control at storage nodes.

Figure 7: Performance of the Crystal bandwidth differentiation service (SLOs per tenant are in MBps).

SLO plus an equal share of spare bandwidth, according to the expected policy behavior defined by colored areas. This demonstrates the effectiveness of our bandwidth control middleware for intercepting and limiting both requests types. We also observe in Fig. 7(a) that `PUT` bandwidth exhibits higher variability than `GET` bandwidth. Concretely, after writing 512MB of data, Swift stopped the transfers of tenants for a short interval; we will look for the causes of this in our next development steps.

Impact of enforcement stage. An interesting aspect to study in our framework are the implications of enforcing bandwidth control at either the proxies or storage nodes. In this sense, Fig. 7(b) shows the enforcement SLOs on `GET` requests at both stages. At first glance, we observe in Fig. 7(b) that our framework makes it possible to enforce bandwidth limits at both stages. However, Fig. 7(b) also illustrates that the enforcement on storage nodes presents higher variability compared to proxy enforcement. This behavior arises from the relationship between the number of nodes to coordinate and the intensity of the workload at hand. That is, given the same workload intensity, a fewer number of nodes (e.g., proxies) offers higher bandwidth stability, as a tenant's requests are virtually a continuous data stream, being easier to control. Conversely, each storage node receives a smaller fraction of a tenant's requests, as normally storage nodes are more numerous than proxies. This yields that storage nodes have to deal with shorter and discontinuous streams that are harder to control.

But enforcing bandwidth SLOs at storage nodes enables to control background tasks like replication. Thus, we face a trade-off between accuracy and control that may be solved with hybrid enforcement schemes.

Mixed tenant activity, variable file sizes. Next, we execute a mixed read/write workload using files of different sizes; small (8MB to 16MB), medium (32MB to 64MB) and large (128MB to 256MB) files. Besides, to explore the scalability, in this set of experiments we resort to a cluster configuration that doubles the size of the previous one (2 proxies and 6 storage nodes).

Appreciably, Fig. 7(c) shows that our enforcement controller achieves bandwidth SLOs under mixed workloads. Moreover, the bandwidth differentiation framework works properly when doubling the storage cluster size, as the policy provides tenants with the desired SLO plus a fair share of spare bandwidth, specially for `T1` and `T2`. However, Fig. 7(c) also illustrates that the `PUT` bandwidth provided to `T1` is significantly more variable than for other tenants; this is due to various reasons. First, we already mentioned the increased variability of `PUT` requests, apparently due to write buffering. Second, the bandwidth filter seems to be less precise when limiting streams that require an SLO close to the node/link capacity. Moreover, small files make the workload harder to handle by the controller as more node assignments updates are potentially needed, specially as the cluster grows. In the future, we plan to continue the exploration and mitigation of these sources of variability.

Controlling background tasks. An advantage of enforcing bandwidth SLOs at storage nodes is that we can also control the bandwidth of background processes via policies. To wit, Fig. 7(d) illustrates the impact of replication tasks on multi-tenant workloads. In Fig. 7(d), we observe that during the first 60 seconds of this experiment (i.e., no SLOs defined) tenants are far from having a sustained `GET` bandwidth of $\approx$ 33MBps, meaning

that they are importantly affected by the replication process. The reason is that, internally, storage nodes trigger hundreds of point-to-point transfers to write copies of already stored objects to other nodes belonging to the ring. Note that the aggregated replication bandwidth within the cluster reached 221MBps. Furthermore, even though we enforce SLOs from second 60 onwards, the objectives are not achieved —specially for tenants `T2` and `T3`— until replication bandwidth is under control. As soon as we deploy a controller that enforces a hard limit of 5MBps to the aggregated replication bandwidth, the SLOs of tenants are rapidly achieved. We conclude that Crystal has potential as a framework to define fine-grained policies for managing bandwidth allocation in object stores.

7.3 Crystal Overhead

Filter framework latency overheads. A relevant question to answer is the performance costs that our filter framework introduces to the regular operation of the system. Essentially, the filter framework may introduce overhead at i) *contacting the metadata layer*, ii) *intercepting the data stream through a filter*[5] and iii) *managing extended object metadata*. We show this in Fig. 8.

Compared to vanilla Swift (`SW`), Fig. 8 shows that the metadata access of Crystal incurs a median latency penalty between 1.5ms and 3ms (`MA` boxplots). For 1MB objects, this represents a relative median latency overhead of 3.9% for both `GET`s and `PUT`s. Naturally, this overhead becomes slightly higher as the object size decreases, but is still practical (8% to 13% for 10KB objects). This confirms that our filter framework minimizes communication with the metadata layer (i.e., 1 query per request). Moreover, Fig. 8 shows that an in-memory store like Redis fits the metadata workload of Crystal, specially if it is co-located with proxy nodes.

Next, we focus on the isolated interception of object requests via Storlets, which trades off performance for higher security guarantees (see Section 4). Fig. 8 illustrates that the median isolated interception overhead of a void filter (`NOOP`) oscillates between 3ms and 11ms (e.g., 5.7% and 15.7% median latency penalty for 10MB and 1MB `PUT`s, respectively). This cost mainly comes from injecting the data stream into a `Docker` container to achieve isolation. We also may consider filter implementation effects, or even the data at hand. To wit, columns `CZ` and `CR` depict the performance of the compression filter for *highly redundant (zeros) and random* data objects. Visibly, the performance of `PUT` requests changes significantly (e.g., objects $\geq$ 1MB) as compression algorithms exhibit different performance depending on the data contents [24]. Conversely, decompression in

[5]We focus on isolated filter execution, as native execution has no additional interception overhead.

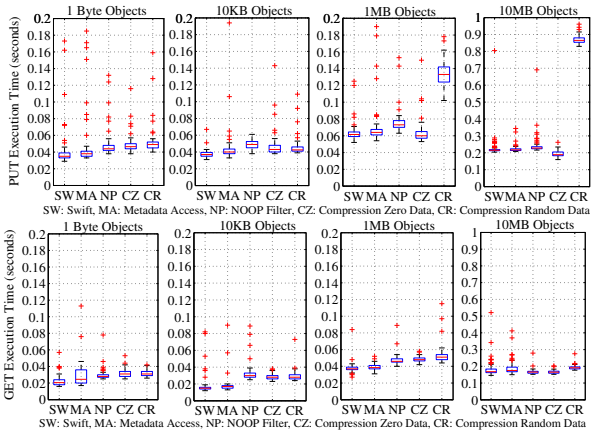


Figure 8: Performance overhead of filter framework metadata interactions and isolated filter enforcement.

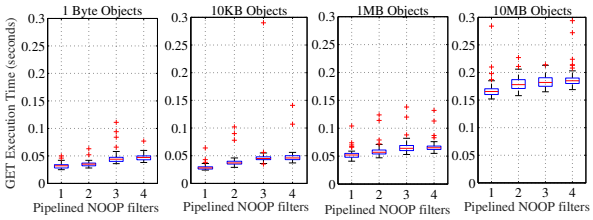


Figure 9: Pipelining performance for isolated filters.

`GET` requests is not significantly affected by data contents. Hence, to improve performance, filters should be enforced in the right conditions.

Finally, our filter framework enables managing extended metadata of objects to store a sequence of data transformations to be undone on retrievals (see Section 4). We measured that reading/writing extended object metadata takes 0.3ms/2ms, respectively, which constitutes modest overhead.

Filter pipelining throughput. Next, we want to further explore the overhead of isolated filter execution. Specifically, Fig. 9 depicts the latency overhead of pipeling multiple `NOOP` Storlet filters. As pipelining is a new feature of Crystal, it required a separate evaluation.

Fig. 9 shows that the latency costs of intercepting a data stream through a pipeline of isolated filters is acceptable. To inform this argument, each additional filter in the pipeline incurs 3ms to 9ms of extra latency in median. This is slightly lower than passing the stream through the `Docker` container for the first time. The reason is that pipelining tenant filters is done within the same `Docker` container, so the costs of injecting the stream into the container are present only once. Therefore, our filter framework is a feasible platform to dynamically compose and pipeline several isolated filters.

Monitoring overheads. To understand the monitoring costs of Crystal, we provide a measurement-based estimation of various configurations of monitoring nodes, workload metrics and controllers. To wit, the monitoring traffic overhead O related to $|\mathscr{W}|$ workload metrics is

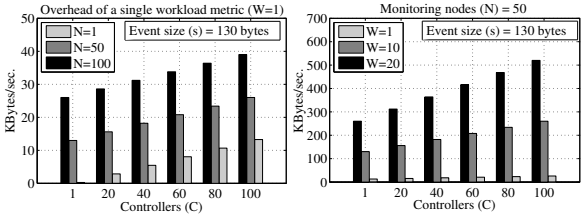


Figure 10: Traffic overhead of Crystal depending on the number of nodes, controllers and workload metrics.

produced by a set of nodes $\mathscr{N}$. Each node in $\mathscr{N}$ periodically sends monitoring events of size s to the MOM broker, which are consumed by $|\mathscr{W}|$ workload metric processes. Then, each workload metric process aggregates the messages of all nodes in $\mathscr{N}$ into a single monitoring message. The aggregated message is then published to a set of subscribed controllers $\mathscr{C}$. Therefore, we can do a worst case estimation of the total generated traffic per monitoring epoch (e.g., 1 second) as: $O = |\mathscr{W}| \cdot [s \cdot (2 \cdot |\mathscr{N}| + |\mathscr{C}|)]$. We also measured simple events (e.g., `PUT_SEC`) to be $s = 130$ bytes in size.

Fig. 10 shows that the estimated monitoring overhead of a single metric is modest; in the worst case, a single workload metric generates less than 40KBps in a 100-machine cluster with $|\mathscr{C}| = 100$ subscribed controllers. Clearly, the dominant factor of traffic generation is the number of workload metrics. However, even for a large number of workload metrics ($|\mathscr{W}| = 20$), the monitoring requirements in a 50-machine cluster do not exceed 520KBps. These overheads seem lower than existing SDS systems with advanced monitoring [33].

8 Related Work

SDS Systems. IOFlow [41], now extended as sRoute [38], was the first complete SDS architecture. IOFlow enables end-to-end (e2e) policies to specify the treatment of IO flows from VMs to shared storage. This was achieved by introducing a queuing abstraction at the data plane and translating high-level policies into queuing rules. The original focus of IOFlow was to enforce e2e bandwidth targets, which was later augmented with caching and tail latency control in [38, 39].

Crystal, however, targets a different scenario. Simply put, it pursues the configuration and optimization of object stores to the evolving needs of tenants/applications, for it needs a richer data plane and a different suite of management abstractions and enforcement mechanisms. For example, tenants require mechanisms to inject custom logic to specify not only system activities but also application-specific transformations on objects.

Retro [33] is a framework for implementing resource management policies in multi-tenant distributed systems. It can be viewed as an incarnation of SDS, because as IOFlow and Crystal, it separates the controller from the mechanisms needed to implement it. A major contribution of Retro is the development of abstractions to enable policies that are system- and resource-agnostic. Crystal shares the same spirit of requiring low develop effort. However, its abstractions are different. Crystal must abstract not only resource management; it must enable the concise definition of policies that enable high levels of programmability to suit application needs. Retro is only extensible to handle custom resources.

IO bandwidth differentiation. Enforcing bandwidth SLOs in shared storage has been a subject of intensive research over the past 10 years, specially in block storage [26, 27, 43, 45, 32, 41, 33]. For instance, mClock [27] achieves IO resource allocation for multiple VMs at the hypervisor level, even in distributed storage environments (dmClock). However, object stores have received much less attention in this regard; vanilla Swift only provides a non-automated mechanism for limiting the "number of requests" [12] per tenant, instead of IO bandwidth. In fact, this problem resembles the one stated by Wang et al. [44] where multiple clients access a distributed storage system with different data layout and access patterns, yet the performance guarantees required are global. To our knowledge, Wu et al. [45] is the only work addressing this issue in object storage. It provides SLOs in Ceph by orchestrating local rate limiters offered by a modified version of the underlying file system (EBOFS). However, this approach is intrusive and restricted to work with EBOFS. In contrast, Crystal transparently intercepts and limits requests streams, enabling developers to design new algorithms that provide distributed bandwidth enforcement [37, 28].

Active storage. The early concept of *active disk* [36, 14, 31, 42], i.e., a HDD with computational capacity, was borrowed by distributed file system designers in HPC environments two decades ago to give birth to active storage. The goal was to diminish the amount of data movement between storage and compute nodes [13, 9]. Piernas et al. [35] presented an active storage implementation integrated in the Lustre file system that provides flexible execution of code near to data in the user space. Crystal goes beyond active storage. It exposes through the filter abstraction a way to inject custom logic into the data plane and manage it via policies. This requires filters to be deployable at runtime, support sandbox execution [7], and be part of complex workflows.

9 Conclusions

Crystal is a SDS architecture that pursues an efficient use of multi-tenant object stores. Crystal addresses unique challenges for providing the necessary abstractions to add new functionalities at the data plane that can be im-

mediately managed at the control plane. For instance, it adds a filtering abstraction to separate control policies from the execution of computations and resource management mechanisms at the data plane. Also, extending Crystal requires low development effort. We demonstrate the feasibility of Crystal on top of OpenStack Swift through two use cases that target automation and bandwidth differentiation. Our results show that Crystal is practical enough to be run in a shared cloud object store.

Acknowledgments

We thank our shepherd Ajay Gulati and the anonymous reviewers. This work has been partly funded by the EU project H2020 "IOStack: Software-Defined Storage for Big Data" (644182) and Spanish research project "Cloud Services and Community Clouds" (TIN2013-47245-C2-2-R) funded by the Ministry of Science and Innovation.

References

[1] Amazon s3. https://aws.amazon.com/en/s3.

[2] Databricks. https://databricks.com.

[3] Docker. https://www.docker.com.

[4] Dropbox. https://www.dropbox.com.

[5] Ifttt. https://ifttt.com.

[6] Mirantis. https://www.mirantis.com.

[7] OpenStack Storlets. https://github.com/openstack/storlets.

[8] Openstack swift. http://docs.openstack.org/ developer/swift.

[9] PVFS Project. http://www.pvfs.org/.

[10] Redis. https://www.redis.io.

[11] Ssbench. https://github.com/swiftstack/ssbench.

[12] Swift performance tunning. https://swiftstack.com/docs/admin/middleware/ratelimit.html.

[13] The Panasas activescale file system (PanFS). http://www.panasas.com/products/panfs.

[14] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. *ACM SIGPLAN Notices*, 33(11):81–91, 1998.

[15] G. A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, The MIT Press, 1985.

[16] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. Taming the cloud object storage with mos. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 7–12, 2015.

[17] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. Mos: Workload-aware elasticity for cloud object stores. In *ACM HPDC'16*, pages 177–188, 2016.

[18] J. Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2007.

[19] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *USENIX OSDI'10*, pages 1–8, 2010.

[20] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[21] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[22] M. Factor, G. Vernik, and R. Xin. The perfect match: Apache spark meets swift. https://www.openstack.org/summit/openstack-paris-summit-2014/session-videos/presentation/the-perfect-match-apache-spark-meets-swift, November 2014.

[23] R. Gracia-Tinedo, P. García-López, M. Sanchez-Artigas, J. Sampé, Y. Moatti, E. Rom, D. Naor, R. Nou, T. Cortes, and W. Oppermann. IOStack: Software-defined object storage. *IEEE Internet Computing*, 2016.

[24] R. Gracia-Tinedo, D. Harnik, D. Naor, D. Sotnikov, S. Toledo, and A. Zuck. SDGen: mimicking datasets for content generation in storage benchmarks. In *USENIX FAST'15*, pages 317–330, 2015.

[25] R. Gracia-Tinedo, Y. Tian, J. Sampé, H. Harkous, J. Lenton, P. García-López, M. Sánchez-Artigas, and M. Vukolic. Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end. In *ACM IMC'15*, pages 155–168, 2015.

[26] A. Gulati, I. Ahmad, C. A. Waldspurger, et al. Parda: Proportional allocation of resources for distributed storage access. In *USENIX FAST'09*, pages 85–98, 2009.

[27] A. Gulati, A. Merchant, and P. J. Varman. mclock: handling throughput variability for hypervisor io scheduling. In *USENIX OSDI'10*, pages 1–7, 2010.

[28] A. Gulati and P. Varman. Lexicographic qos scheduling for parallel i/o. In *ACM SPAA'05*, pages 29–38, 2005.

[29] D. Harnik, R. Kat, D. Sotnikov, A. Traeger, and O. Margalit. To zip or not to zip: Effective resource usage for real-time compression. In *USENIX FAST'13*, pages 229–241, 2013.

[30] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. A pipelined framework for online cleaning of sensor data streams. In *IEEE ICDE'06*, pages 140–140, 2006.

[31] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *ACM SIGMOD Record*, 27(3):42–52, 1998.

[32] N. Li, H. Jiang, D. Feng, and Z. Shi. Pslo: enforcing the x th percentile latency and throughput slos for consolidated vm storage. In *ACM Eurosys'16*, page 28, 2016.

[33] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *USENIX NSDI'15*, 2015.

[34] M. Murugan, K. Kant, A. Raghavan, and D. H. Du. Flexstore: A software defined, energy adaptive distributed storage framework. In *IEEE MASCOTS'14*, pages 81–90, 2014.

[35] J. Piernas, J. Nieplocha, and E. J. Felix. Evaluation of active storage strategies for the lustre parallel file system. In *ACM/IEEE Supercomputing'07*, page 28, 2007.

[36] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia applications. In *VLDB'98*, pages 62–73, 1998.

[37] D. Shue, M. J. Freedman, and A. Shaikh. Fairness and isolation in multi-tenant storage as optimization decomposition. *ACM SIGOPS Operating Systems Review*, 47(1):16–21, 2013.

[38] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska. sRoute: treating the storage stack like a network. In *USENIX FAST'16*, pages 197–212, 2016.

[39] 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 SoCC'15*, pages 174–181, 2015.

[40] R. Stutsman, C. Lee, and J. Ousterhout. Experience with rules-based programming for distributed, concurrent, fault-tolerant code. In *USENIX ATC'15*, pages 17–30, 2015.

[41] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. Ioflow: a software-defined storage architecture. In *ACM SOSP'13*, pages 182–196, 2013.

[42] M. Uysal, A. Acharya, and J. Saltz. Evaluation of active disks for decision support databases. In *IEEE HPCA'00*, pages 337–348, 2000.

[43] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: enabling high-level slos on shared storage systems. In *ACM SoCC'12*, page 14, 2012.

[44] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *USENIX FAST'07*, 2007.

[45] J. C. Wu and S. A. Brandt. Providing quality of service support in object-based file system. In *IEEE MSST'07*, volume 7, pages 157–170, 2007.

[46] E. Zamora-Gómez, P. García-López, and R. Mondéjar. Continuation complexity: A callback hell for distributed systems. In *LSDVE@Euro-Par'15*, pages 286–298, 2015.

WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems

Se Kwon Lee, K. Hyun Lim[†], Hyunsub Song, Beomseok Nam, Sam H. Noh
UNIST (Ulsan National Institute of Science and Technology), [†]Hongik University

Abstract

Recent interest in persistent memory (PM) has stirred development of index structures that are efficient in PM. Recent such developments have all focused on variations of the B-tree. In this paper, we show that the radix tree, which is another less popular indexing structure, can be more appropriate as an efficient PM indexing structure. This is because the radix tree structure is determined by the prefix of the inserted keys and also does not require tree rebalancing operations and node granularity updates. However, the radix tree as-is cannot be used in PM. As another contribution, we present three radix tree variants, namely, WORT (Write Optimal Radix Tree), WOART (Write Optimal Adaptive Radix Tree), and ART+CoW. Of these, the first two are optimal for PM in the sense that they only use one 8-byte failure-atomic write per update to guarantee the consistency of the structure and do not require any duplicate copies for logging or CoW. Extensive performance studies show that our proposed radix tree variants perform considerable better than recently proposed B-tree variants for PM such NVTree, wB+Tree, and FPTree for synthetic workloads as well as in implementations within Memcached.

1 Introduction

Previous studies on indexing structures for persistent memory (PM) have concentrated on B-tree variants. In this paper, we advocate that the radix tree can be better suited for PM indexing than B-tree variants. We present radix tree variant indexing structures that are optimal for PM in that consistency is always guaranteed by a single 8-byte failure-atomic write without any additional copies for logging or CoW.

Emerging persistent memory technologies such as phase-change memory, spin-transfer torque MRAM, and 3D Xpoint are expected to radically change the landscape of various memory and storage systems [4, 5, 7, 9, 10, 14]. In the traditional block-based storage device, the failure atomicity unit, which is the update unit where consistent state is guaranteed upon any system failure, has been the disk block size. However, as persistent memory, which is byte-addressable and non-volatile, will be accessible though the memory bus rather than via the PCI interface, the failure atomicity unit for persistent memory is generally expected to be 8 bytes or no larger than a cache line [5, 6, 12, 13, 15, 19].

The smaller failure atomicity unit, however, appears to be a double-edged sword in the sense that though this allows for reduction of data written to persistent store, it can lead to high overhead to enforce consistency. This is because in modern processors, memory write operations are often arbitrarily reordered in cache line granularity and to enforce the ordering of memory write operations, we need to employ memory fence and cache line flush instructions [21]. These instructions have been pointed out as a major cause of performance degradation [3, 9, 15, 20]. Furthermore, if data to be written is larger than the failure-atomic write unit, then expensive mechanisms such as logging or copy-on-write (CoW) must be employed to maintain consistency.

Recently, several persistent B-tree based indexing structures such as NVTree [20], wB+Tree [3], and FPTree [15] have been proposed. These structures focus on reducing the number of calls to the expensive memory fence and cache line flush instructions by employing an append-only update strategy. Such a strategy has been shown to significantly reduce duplicate copies needed for schemes such as logging resulting in improved performance. However, this strategy does not allow these structures to retain one of the key features of B-trees, that is, having the keys sorted in the nodes. Moreover, this strategy is insufficient in handling node overflows as node splits involve multiple node changes, making logging necessary.

While B-tree based structures have been popular in-memory index structures, there is another such structure,

namely, the radix tree, that has been less so. The first contribution of this paper is showing the appropriateness and the limitation of the radix tree for PM storage. That is, since the radix tree structure is determined by the prefix of the inserted keys, the radix tree does not require key comparisons. Furthermore, tree rebalancing operations and updates in node granularity units are also not necessary. Instead, insertion or deletion of a key results in a single 8-byte update operation, which is perfect for PM. However, the original radix tree is known to poorly utilize memory and cache space. In order to overcome this limitation, the radix tree employs a path compression optimization, which combines multiple tree nodes that form a unique search path into a single node. Although path compression significantly improves the performance of the radix tree, it involves node split and merge operations, which is detrimental for PM.

The limitation of the radix tree leads us to the second contribution of this paper. That is, we present three radix tree variants for PM. For the first of these structures, which we refer to as Write Optimal Radix Tree for PM (WORTPM, or simply WORT), we develop a failure-atomic path compression scheme for the radix tree such that it can guarantee failure atomicity with the same memory saving effect as the existing path compression scheme. For the node split and merge operations in WORT, we add memory barriers and persist operations such that the number of writes, memory fence, and cache line flush instructions in enforcing failure atomicity is minimized. WORT is optimal for PM, as is the second variant that we propose, in the sense that they require only one 8-byte failure-atomic write per update to guarantee the consistency of the structure without any duplicate copies.

The second and third structures that we propose are both based on the Adaptive Radix Tree (ART) that was proposed by Leis et al. [11]. ART resolves the trade-off between search performance and node utilization by employing an adaptive node type conversion scheme that dynamically changes the size of a tree node based on node utilization. This requires additional metadata and more memory operations than the traditional radix trees, but has been shown to still outperform other cache conscious in-memory indexing structures. However, ART in its present form does not guarantee failure atomicity. For the second radix tree variant, we present Write Optimal Adaptive Radix Tree (WOART), which is a PM extension of ART. WOART redesigns the adaptive node types of ART and carefully supplements memory barriers and cache line flush instructions to prevent processors from reordering memory writes and violating failure atomicity. Finally, as the third variant, we present ART+CoW, which is another extension of ART that makes use of CoW to maintain consistency. Unlike B-tree variants where CoW can be expensive, with the radix tree, we show that CoW incurs considerably less overhead.

Through an extensive performance study using synthetic workloads, we show that for insertion and search, the radix tree variants that we propose perform better than recent B-tree based persistent indexes such as the NVTree, wB+Tree, and FPTree [3, 15, 20]. We also implement the indexing structure within Memcached and show that similarly to the synthetic workloads, our proposed radix tree variants perform substantially better than the B-tree variants. However, performance evaluations show that our proposed index structures are less effective for range queries compared to the B-tree variants.

The rest of the paper is organized as follows. In Section 2, we present the background on consistency issues with PM and PM targeted B-tree variant indexing structures. In Section 3, we first review the radix tree to help understand the main contributions of our work. Then, we present the three radix tree variants that we propose. We discuss the experimental environment in Section 4 and then present the experimental results in Section 5. Finally, we conclude with a summary in Section 6.

2 Background and Motivation

In this section, we review background work that we deem most relevant to our work and also necessary to understand our study. First, we review the consistency issue of indexing structures in persistent memory. Then, we present variants of B-trees for PM. As the contribution of our work starts with the radix tree, we review the radix tree in Section 3.

2.1 Consistency in Persistent Memory

Ensuring recovery correctness in persistent indexing structures requires additional memory write ordering constraints. In disk-based indexing, arbitrary changes to a volatile copy of a tree node in DRAM can be made without considering memory write ordering because it is a volatile copy and its persistent copy always exists in disk storage and is updated in disk block units. However, with failure-atomic write granularity of 8 bytes in PM, changes to an existing tree node must be carefully ordered to enforce consistency and recoverability. For example, the number of entries in a tree node must be increased after a new entry is stored. If the system fails after we increase the number of entries but before the new entry is stored in its corresponding space, the garbage entry previously stored in that space will be mistaken as a valid entry resulting in inconsistency.

In order to guarantee consistency between volatile CPU caches and non-volatile memory, we have to ensure the ordering of memory writes via memory fence and

cache line flush instructions. In the Intel x86 architecture, the CLFLUSH instruction is used to flush a dirty cache line back to memory and MFENCE is the load and store fence instruction that prevents the reordering of memory access instructions across the fence. Since CLFLUSH is ordered only with respect to MFENCE, CLFLUSH needs to be used along with MFENCE to prevent reordering of CLFLUSH instructions [21]. These memory fence and cache line flush instructions are known to be expensive [3, 20].

Another important aspect of maintaining consistency is the write size. In legacy B-tree variants, insertion or deletion of a node entry results in modification of a large portion of the node because the entries remain sorted. This is because insertion or deletion of an entry can result in shifts of data within the node. Such shifts are likely to be larger than the failure atomicity unit.

To resolve this problem, legacy systems generally rely on techniques such as logging or CoW. Logs can be used to undo or redo activities such that the system remains in a consistency state. CoW creates a copy and makes updates to the copy. This allows for atomic validation of the copy by overwriting the pointer with an atomic 8-byte store operation. Although logging and CoW guarantee consistency in the presence of failure, they hurt update performance especially when the updated data is large as they both need to duplicate the write operations.

2.2 Persistent B+-Trees

In recent years, several indexing trees for PM such as CDDS B-tree [17], NVTree [20], wB+Tree [3], and FP-Tree [15] have been proposed. To the best of our knowledge, all previously proposed persistent indexes are variants of the B-tree, which has been widely used in various domains including storage systems. We review each of these trees in detail below.

CDDS B-Tree: CDDS (Consistent and Durable Data Structure) B-tree is a multi-version B-tree (MVBT) for PM [17]. When a tree node is updated in the CDDS B-tree, it creates a copy of the updated entry with its version information instead of overwriting the entry, which guarantees recoverability and consistency. However, CDDS B-tree suffers from numerous dead entries and dead nodes. Also, it calls the expensive MFENCE and CLFLUSH instructions as many times as the number of entries in a tree node to sort the entries. Hence, CDDS B-tree is far from satisfactory in terms of both insertion and search performance.

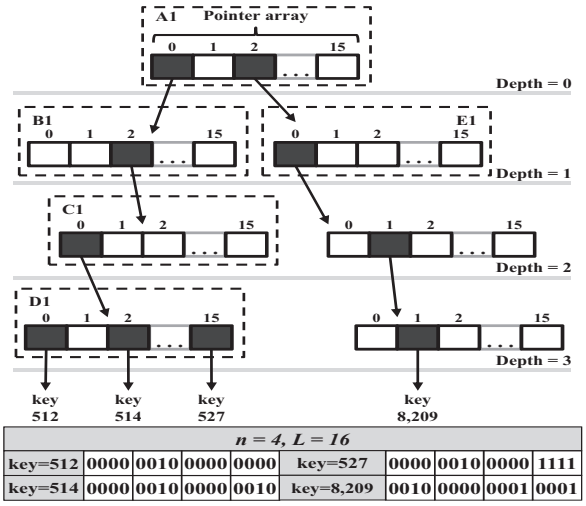
NVTree: NVTree proposed by Yang et al. [20] reduces the number of expensive memory fence and cache line flush instructions by employing an append-only update strategy. Due to this strategy and the fact that only the leaf nodes are kept in PM, NVTree requires only two cache line flushes, one for the entry and the other for the entry count, resulting in improved performance. This results in two consequences; first, the leaf node remains unsorted and second, the internal nodes may be lost upon system failure though the internal nodes can trivially be reconstructed using the leaf nodes in PM. However, NVTree requires all internal nodes to be stored in consecutive memory blocks to exploit cache locality, and within the large memory block, internal nodes are located by offsets instead of pointers. However, because NVTree requires internal nodes to be stored in consecutive blocks, every split of the parent of the leaf node results in the reconstruction of the entire internal nodes. We show in our experiments that due to this reconstruction overhead, NVTree does not perform well for applications that insert data on the fly.

FPTree: FPTree is another persistent index that keeps internal nodes in volatile memory while leaf nodes are kept in PM [15]. By storing the internal nodes in volatile memory, FPTree exploits hardware transactional memory to efficiently handle concurrency of internal node accesses. FPTree also proposes to reduce the cache miss ratio via fingerprinting. Fingerprints are one-byte hashes for keys in each leaf node. By scanning the fingerprints first before a query searches keys, FPTree reduces the number of key accesses and consequently, the cache miss ratio. Although FPTree shows superior performance to NVTree, FPTree also requires reconstruction of internal nodes when a system crashes.

wB+Tree: wB+Tree proposed by Chen and Jin also adopts an append-only update strategy, but unlike NVTree and FPTree, wB+Tree stores both internal and leaf nodes in PM [3]. Since the entries in internal nodes must be sorted, wB+Tree proposes to sort the entries via the slot array, which adds a level of indirection to the actual keys and pointers. That is, the slot array stores the index of keys in sorted order. Since the index is much smaller than the actual key and pointer, seven key indexes can be atomically updated via the 8-byte atomic write operation. wB+Tree also proposes to use an 8-byte bitmap to increase node capacity. When the bitmap is used, wB+Tree requires at least four cache line flushes. If only the slot array is used, the number of cache line flushes decreases to two. Although the number of cache line flushes is dramatically reduced compared to CDDS B-tree, wB+Tree carries the overhead of indirection. Also, wB+Tree still requires expensive logging or CoW for a node split.

3 Radix Trees for PM

In this section, we present the basics of the radix tree. We also discuss the three radix tree variants that we propose, namely, WORTPM (Write Optimal Radix Tree for PM) or simply, WORT (Write Optimal Radix Tree), WOART (Write Optimal Adaptive Radix Tree), and ART+CoW.



n = 4, L = 16									
key=512	0000	0010	0000	0000	key=527	0000	0010	0000	1111
key=514	0000	0010	0000	0010	key=8,209	0010	0000	0001	0001

Figure 1: An example radix tree

3.1 Radix Tree

Traditionally, radix trees come in two versions; one in basic form, which we refer to as the original radix tree, and the other that uses path compression to save memory [2, 26, 27, 28]. For ease of presentation, we first discuss the original version and defer the discussion on the path compressed version to Section 3.3.

The radix tree does not explicitly store keys in its tree nodes. Instead, a node consists of an array of child pointers, each of which is represented by a chunk of bits of a search key as in a hash-based index. Taking the radix tree example in Figure 1, each key is composed of 16 bits, with a chunk of 4 bits in length. Starting from the root node, the most significant leftmost 4 bits of each key is used to determine the subscript within the pointer array. Taking the key=527 case as a walking example, the left 4 bits, 0000, determines that the leftmost pointer points to the child node. In the next level, the next chunk of bits in the search key are used as the subscript of the child pointer array. This would be 0010 for key=527, meaning that the pointer in element 2 points to the next child. In this manner, in the walking example, we see that the next chunk of 4 bits, 0000, determines the next level child and that the least significant bits of the search key, 1111, are used in the leaf node.

There are two key characteristics of the radix tree that are different from B-tree variants. The first is that the height of the radix tree is determined and fixed by the length of the index key and the chunk size. For a maximum key length of L bits and the chunk, which represents the index to the child pointer, of n bits, which allows for a maximum 2^n child node pointers, the search path length is $\lceil L/n \rceil$. An often mentioned weakness of the radix tree is that its height ($\lceil L/n \rceil$) is, in general, taller than that of the B+-tree, which is $log_B N$, where N is the number of keys and B is the node degree [11]. This may result in deeper traversals of the tree.

The second characteristic is that the tree structure is independent of the insertion order but dependent on the distribution of keys. Whereas the B-tree variants maintain a balanced tree growing and shrinking according to the number of data, a radix tree has a fixed number of nodes determined by the maximum key range. How these nodes are used determines its effectiveness in terms of memory usage. For example, when the keys are sparsely distributed, the radix tree makes inefficient use of memory in contrast to when it is dense or skewed.

Due to these limitations and despite proposals to overcome such limitations [2], the radix tree has not been a popular indexing data structure. However, we find that the radix tree possesses features that may be exploited for efficient use with PM. First, with the radix tree, it is possible to traverse the tree structure without performing any key comparison because the positions of the child pointers are static and fixed according to its order. For example, if 527 is indexed as shown in Figure 1, 527 can be easily found by using each 4 bits as a subscript of the index without any comparison operation, i.e., `radix_index[0][2][0][15]`, as 527 is 0000 0010 0000 1111 in binary. In contrast, the B+-tree would require comparing the search key with other keys for each visited node. Such difference in activity can affect cache performance resulting in performance differences.

Also for insertion, the radix tree does not modify any existing entries for the same reason. That is, the number of child pointers in a radix tree node is fixed to 2^n, and it never overflows. Since the radix tree does not store keys, sorting, by nature, is not necessary. This is in contrast to B-tree variants that need to keep the keys sorted and also require expensive split or merge operations accompanied by logging to guarantee consistency.

3.2 Failure Atomic Write in Radix Tree

In this section, we describe the small changes that we made to make radix tree PM efficient. With the changes that we propose, the radix tree will remain consistent upon system failure without requiring any kind of logging or replication mechanism as long as the 8-byte failure atomicity assumption is satisfied. Essentially, there are just two simple changes that need to be made, which we describe in the following.

The first modification is making sure that a write to change a pointer is done in a particular order. Let us elaborate using an example of inserting key=3,884 (0000 1111 0010 1100 in binary) into the radix tree shown in Figure 1. With a given key, we traverse down the path using the partial keys until we find a pointer value that is NULL. For 3,884, as the first partial key is 0000, we follow the leftmost child pointer from the root node. At this level (depth 1), we find that the next partial key 1111 and

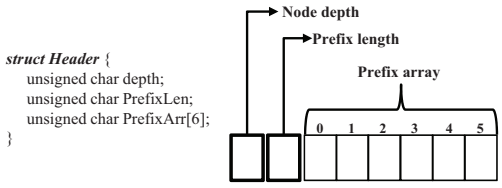


Figure 2: Compression header

that its pointer is NULL. Once we find the child pointer to be NULL, we create a new node and continue doing this until the leaf node is created. For the 3,884 example, we create a new node in depth 2 and a leaf node in depth 3. Finally, we replace the NULL pointers with the addresses of the new nodes.

The first modification that we propose is that the operation to replace the very first NULL pointer (if we have a sequence of nodes created) with the address of the next level node be the last operation. This ensures that the entire operation is failure-atomic. Since the pointer assignment statement is an 8-byte atomic write operation, no form of logging is necessary in the radix tree. However, we do need to call a few memory access serialization instructions to enforce failure atomicity. For example, if 8,209 is indexed as shown in Figure 1, first, we must call memory fence and cache line flush instructions to make sure all nodes leading to the leaf node, including the leaf node, is written to PM. Only then, should we change the very first NULL pointer in the root, that is, element 2 in node A1 to point to node E1. This change is then persisted with the memory fence and cache line flush instructions.

3.3 Radix Tree Path Compression

Thus far, we have described the original radix tree without path compression optimization. In this section, we describe the workings of path compression as our second proposed change is related to this matter.

Although the deterministic structure of the radix tree is the source of its good performance, it is also the weakest point as the key distribution has a high impact on the tree structure and memory utilization. If the distribution of the keys is sparse, the implicit key representation of the radix tree can waste excessive memory space. For example, suppose a string key is stored in a radix tree and there is no other key that shares the prefix with the key. If the number of child pointers in a node is 2^8, each node can implicitly index an 8-bit character but requires an 8-byte pointer per each child. That is, the tree node will use 8×256 bytes for each letter in the key. In order to mitigate this memory space utilization problem, we can consider reducing the number of child pointers in a node. However, this could result in a longer search path, possibly degrading search performance. The path compressed radix tree can save space by removing the internal nodes

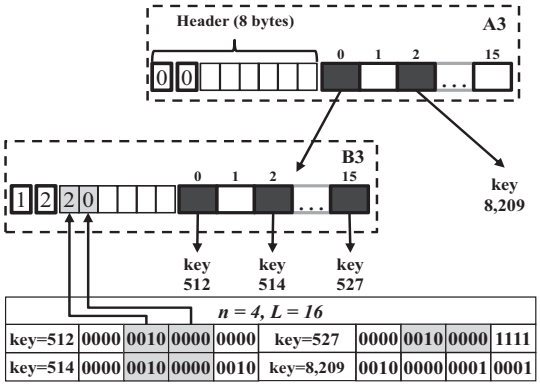


Figure 3: Path compression collapses radix tree nodes

that have only one child per node.

If a node in the radix tree has a single child such as key 8,209 in Figure 1, the node does not have to exist in order to distinguish it from other search paths. Hence, the node can be safely removed and created in a lazy manner until it becomes shared with another child node without hurting correctness. Path compression optimization in the radix tree truncates unique paths in the tree structure. Path compression is known to improve memory utilization especially when the key distribution is sparse. Moreover, path compression helps improve indexing performance by shortening the search path. There are three ways of implementing path compression in the radix tree, that is, the pessimistic, optimistic, and hybrid methods [11].

The pessimistic method explicitly saves the collapsed unique search path as the prefix array in the child node. The pessimistic method requires more memory space in the compressed nodes, but it can prune out unmatching keys instantly. On the other hand, the optimistic method stores the length of the collapsed prefix in the child node, instead of the collapsed prefix itself. Hence, the optimistic method cannot compare the collapsed prefix in the compressed node. Instead, it postpones the comparison of the collapsed keys until we reach a leaf node. The hybrid method combines the two by using the pessimistic method when the collapsed prefix is smaller than a specific length, and the optimistic method, otherwise.

Figure 2 illustrates the structure of a radix tree node header and Figure 3, which is a hybrid compressed version of Figure 1 as it simultaneously stores the length and the collapsed search path, shows how it is used to combine the inner nodes that share the common prefix. Prefix length specifies how many inner nodes are collapsed. In the example shown in Figure 3, 512, 514, and 527 share the second and third partial keys (2 and 0). Hence, the leaf node stores 2 as a prefix length. The prefix array is the collapsed common prefix. In the example, the prefix array stores the common prefix 2 (0010 in binary) and 0 (0000 in binary). Note that we always make use of an 8-

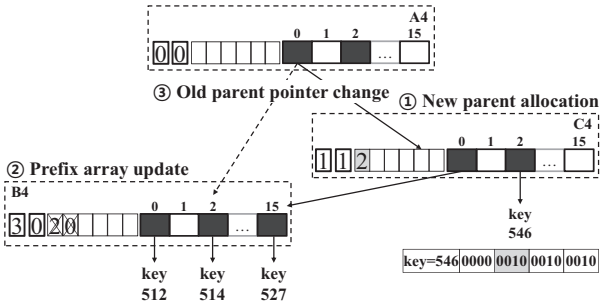


Figure 4: Path compression split

byte compression header. This is to maintain consistency with 8-byte atomic writes, which we elaborate on later.

Algorithm 1 SplitCmp(node,key,value,depth,diffPrfxIdx)

```
1: /*N=node,K=key,V=value,D=depth*/
2: Allocate newParent and newLeaf(K,V)
3: Move a part of header of N to header of newParent
4: Insert newLeaf and N into newParent as children
5: Allocate tmpHdr
6: Record header of N to be updated into tmpHdr
7: *((uint64*)&N.Hdr) = *((uint64*)&tmpHdr;
8: mfence();
9: clflush(&newLeaf);
10: clflush(&newParent);
11: clflush(&N.Hdr);
12: mfence();
13: /*Update old parent pointer*/
14: oldParent.children[]=newParent;
15: mfence();
16: clflush(&oldParent.children[]);
17: mfence();
```

3.4 WORT: Write Optimal Radix Tree

While it is trivial to make the radix tree that is not path compression optimized be failure-atomic as was shown in Section 3.2, doing so with the path compression optimized radix tree is more complicated as nodes are split and merged dynamically. The second modification that we propose makes the path compression optimized radix tree failure-atomic. (Hereafter, the radix tree we refer to are path compression optimized unless otherwise stated.) Note that even though the structure of the radix tree is no longer static, it is nevertheless still deterministic.

The second modification to the radix tree that we propose is the addition of the node depth information to each compression header, which was not needed in the legacy radix tree. Note that in our design, this requires one byte, which should be sufficient even in general deployment, and does not compromise the memory saving effect of the legacy path compression scheme. We now show how the node depth is used to guarantee failure atomicity during insertions. Let us again make use of an example.

Figure 4 is a depiction of how a compressed internal node (*B*3 in Figure 3) splits when the prefix of the key to insert does not match the prefix array, while Algorithm 1 describes the algorithm involved. Let us now go through the process step by step. Assume key=546 (0000 0010 0010 0010 in binary) is being inserted to the radix tree in Figure 3. Since 546 shares only the first partial key (4-bit prefix) with the prefix array of node *B*3, we need to split and create a parent node for the smaller common prefix. This new node is depicted as *C*4 in Figure 4. Once the new node *C*4 stores child pointers to node *B*4 and key 546, node *B*3 (in Figure 3) needs to delete the two partial keys in the prefix array (lines 5-12 in Algorithm 1). Also, node *A*3 (in Figure 3) needs to replace the child pointer to *B*3 to point to *C*4 (lines 12-17) as shown in node *A*4, and this must be done atomically. Otherwise, failure atomicity is not guaranteed and the tree may end up in an inconsistent state. For example, say the system crashes as *B*3 is changed to *B*4, but *A*3 is still unchanged.

Since multiple tree nodes cannot be updated atomically, persistent structures such as the B-tree variants employ expensive logging methods. However, we find that the radix tree can tolerate this temporary inconsistent state by storing the depth in each node and skipping the comparison of the partial key that is currently being expanded to a new parent node. That is, if the node that expands can atomically update its depth, prefix length, and prefix array altogether, then the radix tree can return to its consistent state without relying on logging.

Consider the example of Figure 4 once again. Before node *B*3 is expanded, the depth of node *B*3 is 1 (starting from 0 at root node) and the prefix length is 2, which indicates that node *B*3 indexes the second and third prefixes. After creating node *C*4, the depth and the prefix length need to be updated to 3 and 0, respectively. Assume the system crashes after we update the prefix length and depth, but before we update the child pointer of node *A*3. If we only had the prefix length as in the tradition radix tree, there is no way to detect node *B*4 is in an inconsistent state. However, if the depth is atomically updated along with the prefix length, which is possible with the 8-byte failure atomic write assumption, we can easily detect that there is a missing node between *A*3 and *B*4. Once we detect the inconsistency, the inconsistent node can reconstruct its previous depth, prefix length, and prefix array by selecting two leaf nodes from two arbitrary search paths (lines 3-4 of Algorithm 2) and recomputing the common prefix (lines 5-12 of Algorithm 2). Note that path compression guarantees the existence of at least two leaf nodes in any internal node and that the prefix array has the largest common prefix of every key in a node. In the example of Figure 4, suppose node *B*4 selects key=512 (0000 0010 0000 0000 in binary) and key=527 (0000 0010 0000 1111 in binary) as the two arbitrary leaf nodes. The largest common prefix of those two keys

Algorithm 2 RecoverHeader(node,depth)

```
 1: /*N=node,D=depth*/
 2: /*Select two different arbitrary leaves of N*/
 3: L1 = SelectArbitraryLeaf(N);
 4: L2 = SelectArbitraryLeaf(N);
 5: safeHdr = AllocHeader();
 6: safeHdr.depth=D;
 7: Compute the largest common prefix of L1 and L2
 8: Record it into prfxLen and prfxArr of safeHdr
 9: *((uint64*)&N.Hdr) = *((uint64*)&safeHdr);
10: mfence();
11: clflush(&N.Hdr);
12: mfence();
```

is 0000 0010 0000 in binary. The first prefix 0000 is ignored because we reconstruct the node in depth 1.

We name the radix tree that incorporates the two modifications that we mentioned, one in Section 3.2 and one in this section, WORTPM for Write Optimal Radix Tree for PM, or just WORT. WORT is optimal in that consistency is accomplished by using only 8-byte failure-atomic writes for every operation without requiring any logging or duplication of data under any circumstance.

3.5 Write Optimal Adaptive Radix Tree

Even with path compression, with the radix tree, there is a well known trade-off between tree traversal performance and memory consumption, i.e., if we increase the number of child pointers in a node, the tree height decreases but node utilization is sacrificed. Poor node utilization and high memory consumption have been pointed out as the major disadvantages of the radix tree. In order to resolve these problems, studies such as Generalized Prefix Tree [2] and Adaptive Radix Tree (ART) [11] have been conducted. In this section, we discuss ART and how we adapt ART for PM.

ART is a space efficient radix tree that adaptively changes its node size according to node utilization. In order to reduce the tree height, ART sets the number of child pointers in a node to 2^8 and uses one-byte sized partial keys per node. In parallel, ART reduces memory space consumption by using one of four different node types, namely, NODE4, NODE16, NODE48, and NODE256, according to node utilization. Starting with NODE4, ART adaptively converts nodes into larger or smaller types as the the number of entries exceeds or falls behind the capacity of a node type. Although ART has been shown to outperform other state-of-the-art cache conscious in-memory indexing structures including FAST [11], ART in its current form does not guarantee failure atomicity in PM. Hence, we redesign the node structure of ART, without compromising its memory saving effect, to enforce failure atomicity in PM, which we refer to as Write Optimal Adaptive Radix Tree (WOART). In particular, we find that for PM, NODE4

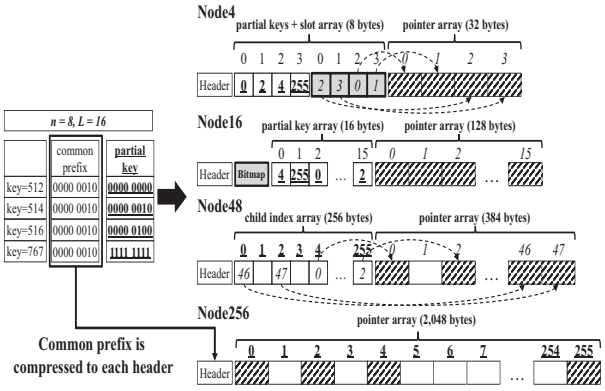


Figure 5: Node structures in WOART

and NODE16 have to be redesigned, NODE48 slightly modified, and NODE256 left unchanged from the original design.

NODE4: In order to reduce the node size when node utilization is low, ART utilizes the node type NODE4, a node that has no more than 4 entries. Since implicit key representation in NODE4 wastes memory space and is not helpful in distinguishing index keys, NODE4 explicitly stores partial keys. Hence, in the original ART scheme, four pairs of partial keys and pointers are kept. The partial keys and pointers are stored at the same index position in parallel arrays and sorted together by the partial key values. With sorting employed, some form of logging becomes a requirement to ensure consistency.

In WOART, we make the following changes to ensure consistency with a single 8-byte atomic write. First, pointers are updated in append-only manner only when an empty entry is available. Then, we add a level of indirection by having a separate slot array that serves as an index to the position of the pointer corresponding to the partial key as shown in NODE4 of Figure 5. Finally, we make use of the fact that the partial key size in the radix tree node is just one byte and that NODE4 only stores four keys per node, plus the fact that the entire slot array size is also only four bytes. Hence, the partial keys and the slot array in NODE4 altogether can be written in a single 8-byte atomic write operation. By performing this operation as the last update, consistency of the tree is guaranteed.

Algorithm 3 shows the details of the insertion algorithm for NODE4. Deletion is omitted as it is analogous to insertion. First, we look for an empty pointer in the slot array (line 1). If there is one (idx is returned), we first make a copy of the 8-byte partial keys and slot array and insert the partial key value and the idx value into the copy (lines 3 and 4). Then, we store the child address in the pointer array entry indexed by idx, and call `mfence` and `clflush` (lines 5-8). Finally, we atomically update the partial keys and slot array by atomically overwriting

Algorithm 3 AddChild4(node4,PartialKey,child)

```
1:  if node4 is not full then
2:      idx = getEmptyIdx(node4.slot);
3:      CopySlot(tmpSlot,node4.slot);
4:      InsertKeytoSlot(tmpSlot,PartialKey,idx);
5:      node4.ptrArr[idx]=child;
6:      mfence();
7:      clflush(&node4.ptrArr[idx]);
8:      mfence();
9:      *((uint64*)node4.slot)=*((uint64*)&tmpSlot);
10:     mfence();
11:     clflush(node4.slot);
12:     mfence();
13: else
14:     Copy and Exchange node4 to node16
```

Algorithm 4 AddChild16(node16,PartialKey,child)

```
1:  if node16 is not full then
2:      idx=getEmptyIdx(node16.bitmap);
3:      node16.partialkeys[idx]=PartialKey;
4:      node16.ptrArr[idx]=child;
5:      mfence();
6:      clflush(&node16.partialkeys[idx]);
7:      clflush(&node16.ptrArr[idx]);
8:      mfence();
9:      node16.bitmap+ = (0x1UL << idx);
10:     mfence();
11:     clflush(&node16.bitmap);
12:     mfence();
13: else
14:     Copy and Exchange node16 to node48
```

the original with the copy of the 8-byte partial keys and slot array and call `mfence` and `clflush` (lines 9-12).

If a node needs more than 4 entries, we expand the node into a NODE16 type through memory allocation and multiple write operations. Note, however, that consistency is always guaranteed during this expansion as the final change to the parent of the new NODE16 is always done with a single 8-byte atomic write.

NODE16: A NODE4 type node becomes a NODE16 type when the number of child pointers grows past four and can have as many as 16 child pointers. Similarly to NODE4, the original NODE16 node keeps the partial keys sorted. However, as previously mentioned, this is an expensive task with PM.

In WOART, with NODE16, we take a similar approach as with NODE4 as we explicitly store keys in append-only manner. However, unlike NODE4, NODE16 does not sort partial keys nor use a slot array to store the indexes to child pointers. Instead, partial keys and pointers are stored as parallel arrays, denoted as partial key array and pointer array, respectively, in Figure 5. Note that there is also a 16-bit bitmap that distinguishes the valid and invalid key and pointer values. The atomic write of this bitmap ensures the consistency of the tree.

Specifically, and in relation to Algorithm 4, we refer to the bitmap (line 2) to find an empty entry to insert the partial key. Then, the partial key and pointer values are placed in the empty entry position of the parallel arrays (lines 3-8). Finally, the bitmap position of the empty entry is set to 1 with an atomic write (lines 9-12). This guarantees consistency of the radix tree.

Note that for deletion, we need to manipulate the bitmap to indicate the invalidness of partial keys and pointers. This is also an atomic write operation, hence, consistency is maintained. Also note that although the partial keys are not sorted, this does not hurt search performance as NODE16 has no more than 16 partial keys. Comparing a given search key with 16 partial keys that fit in a single cache line can be performed very efficiently in modern processors. As this is not true when the number of keys becomes large, WOART explicitly stores partial keys only for NODE4 and NODE16 types.

NODE48 and NODE256: Let us now go over the NODE48 and NODE256 type nodes in WOART. As shown in Figure 5, a NODE256 type node is exactly the same as a node in the original radix tree (see Figure 1). Hence, for NODE256, we simply make use of WORT and do not discuss NODE256 any further.

For NODE48, the details are referred to the original ART [11] as it is essentially what is used. We make use of the open source code provided by Leis et al. [24] but do make slight code changes to make it consistent in PM. However, the essence is the same. Specifically, NODE48 keeps two separate arrays, one that has 256 entries indexed by the partial key and one that has 48 entries, each of which will hold one of the child pointers, respectively denoted child index array and pointer array in Figure 5. Consistency is ensured by making writes to the pointer array first, and then atomically writing the pointer array index value to the child index array, the index of which is determined by the partial key. Note that we search for an available entry in the pointer array by checking the child index array. If we instead search for an available entry by checking for a NULL pointer in the pointer array, as was implemented by Leis et al. [24], system failure may result in leaving non-reusable invalid pointers.

3.6 ART with Copy-on-Write

The third radix tree variant that we consider in this study is one that makes use of copy-on-write (CoW). CoW in the radix tree is much simpler than that with B-tree variants as CoW occurs for only one node upon an update as only the updated node itself is affected upon an update. That is, for any update one can simply create a copy of the node and maintain consistency by replacing the pointer in its parent node with the address of the copy at the final moment. This is in contrast to B-tree variants where node changes can cascade to other nodes, for example, due to splits forcing parent nodes to also split.

In this study, we consider ART with CoW, which we

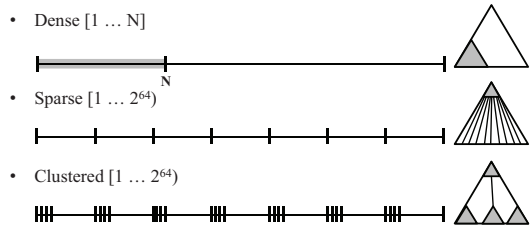


Figure 6: Workloads

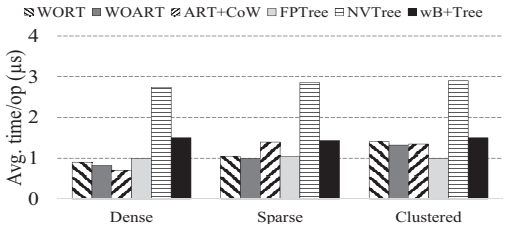


Figure 7: Insertion performance in DRAM latency

refer to as ART+CoW. ART+CoW combines the features of WORT, WOART, and CoW. First, for NODE256, we incorporate the first modification of WORT discussed in Section 3.4. However, for path compression, instead of adding the node depth, we use CoW for both split and merge. Second, for NODE48, we make use of the same mechanism as in WOART. Finally, for NODE16 and NODE4, we simply employ CoW. That is, we make a copy of the node, make changes to it, then change the parent pointer value with the 8-byte failure-atomic write.

4 Experimental Environment

To test the effectiveness of the proposed radix trees, we implement them and compare their performance with state-of-the-art PM indexing structures. The experiments are run on a workstation with an Intel Xeon E5-2620 v3 2.40GHz X 2, 15MB LLC (Last Level Cache), and 256GB DRAM running the Linux kernel version 4.7.0. We compile all implementations using GCC-4.4.7 with the -O3 option.

To observe the effect of PM latency on the performance of the data structure, we emulate PM latency using Quartz [1, 18], a DRAM-based PM performance emulator. Quartz emulates PM latency by injecting software delays per each epoch and throttling the bandwidth of remote DRAM using thermal control registers. We emulate read latency of PM using Quartz while disabling its bandwidth emulation. Since write memory latency emulation is not yet supported in the publicly available Quartz implementation [1], we emulate PM write latency by introducing an additional delay after each `clflush` and `mfence` instructions, as in previous studies [7, 9, 19]. No delays are added for the `store` instruction as the CPU cache hides such delays [25].

For comparison, we implement wB+Tree, NVTree and FPTree [3, 15, 20]. For wB+Tree, we implement both the *slot-only* and *bitmap+slot* schemes, but we present the performance of only the bitmap+slot scheme denoted as `wB+Tree` because we observe that the bitmap+slot scheme has lower node split overhead and search performance is better due to the large node degree. Note that the internal nodes of NVTree and FPTree are designed to be volatile and does not guarantee failure atomicity. As we consider PM latency in our experiments, for NVTree and FPTree, we distinguish latency for internal nodes in DRAM and leaf nodes in PM.

For the workloads, we make use of three synthetically generated distributions of 8-byte integers. Unlike B-tree based indexes, the radix tree is sensitive to the key distribution due to its deterministic nature. To see how the indexes react to extreme cases, we consider three distributions as shown in Figure 6. In Dense key distribution, we generate sequential numbers from 1 to 128M, so that all keys share a common prefix. This workload is the ideal case for the radix tree since overall node utilization is 100%. In Sparse key distribution, keys are uniformly distributed, thus they share a common prefix only in the upper level of the tree structure. For the lower level nodes, the radix tree relies on path compression optimization to improve node utilization. In Clustered key distribution, we merge Dense and Sparse key distributions to model a more realistic workload. Specifically, we generate 2 million small dense distributions, each consisting of 64 sequential keys. In Clustered key distribution, the middle level nodes share common prefixes. For all three distributions, the keys are inserted in random order and experimental results are presented by using a single thread.

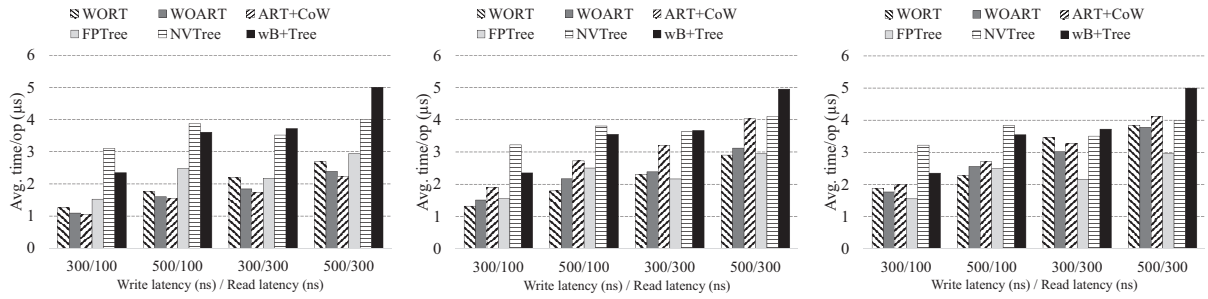
5 Performance Evaluation

In this section, we evaluate the three proposed radix tree variants against the state-of-the-art persistent indexing structures, namely, wB+Tree, NVTree, and FPTree.

5.1 Insertion Performance

Figure 7 shows the average insertion time for inserting 128 million keys for the three different distributions when the entire memory space is of DRAM latency. We set the number of child pointers of WORT to 2^4 so that each node indexes 4-bit partial keys for a maximum of 16 child pointers. We see from the results that in general the radix based trees perform considerably better than NVTree and wB+Tree. The range of benefits and the best radix tree variant depends on the workload.

For NVTree, performance suffers because it requires internal nodes to be pre-allocated in consecutive memory space and a node split results in reconstruction of all the internal nodes. FPTree performs the best among the B-tree variants and, in some cases, better than the radix



(a) Dense (b) Sparse (c) Clustered

Figure 8: Insertion Performance Comparison

Table 1: Average LLC Miss and CLFLUSH per Insertion, LLC Miss per Search, and Leaf Node Depth for Various PM Indexing Structures

D (Dense) **S** (Sparse) **C** (Clustered)	**(a) Average number of LLC miss per insertion**			**(b) Average number of CLFLUSH per insertion**			**(c) Average number of LLC miss per search**			**(d) Average leaf node depth**		
	D	**S**	**C**	**D**	**S**	**C**	**D**	**S**	**C**	**D**	**S**	**C**
WORT	6.3	7.2	17.0	2.2	2.4	2.3	6.5	7.6	11.5	7.0	7.0	8.2
WOART	5.7	7.9	11.2	2.4	3.5	3.7	4.8	7.9	8.9	4.0	4.0	4.2
ART+CoW	5.0	12.7	12.9	2.4	3.8	3.9	3.8	6.2	8.8	4.0	4.0	4.2
FPTree	6.8	6.8	6.8	4.8	4.8	4.8	13.9	14.1	14.1	3.0	3.0	3.0
NVTree	35.0	35.6	33.6	3.3	3.3	3.3	33.5	33.5	33.3	3.0	3.0	3.0
wB+Tree	22.3	22.4	22.4	6.0	6.0	6.0	22.9	22.8	23.3	4.0	4.0	4.0

tree variants. However, this comparison must be made with caution as FPTree assumes that the internal nodes are in DRAM. This has the drawback that when the system recovers from failure or rebooted the internal nodes must be reconstructed incurring considerable overhead.

Considering only the radix trees for the distributions in Figure 7, we see that for Clustered distribution, insertion time is roughly 1.5× higher than for the other two. As shown in column (a) in Table 1, this is due to the higher number of LLC misses incurred as the common prefix of the Clustered distribution is much more fragmented due to the scattered tree nodes than the other two distributions.

Figure 8 shows the insertion results as the latency for reads and writes are changed. The numbers on the *x*-axis represent the latency values in nanoseconds. The default latency, that is of DRAM as reported by Quartz, is 100ns. As PM read and write latency is generally expected to be comparable or slightly worse than those of DRAM, we set the latency to various values as shown in the figure. For these experiments, the internal nodes of NVTree and FPTree are considered to be in DRAM, hence not affected by the latency increase of PM. This should result in more favorable performance for these two trees.

Throughout the results, whether read or write latency is increased, we see that the radix tree variants consistently outperform the B-tree variants, except for FPTree. However, as latency increases, wB+Tree and the radix tree variants that store every node in PM suffer more.

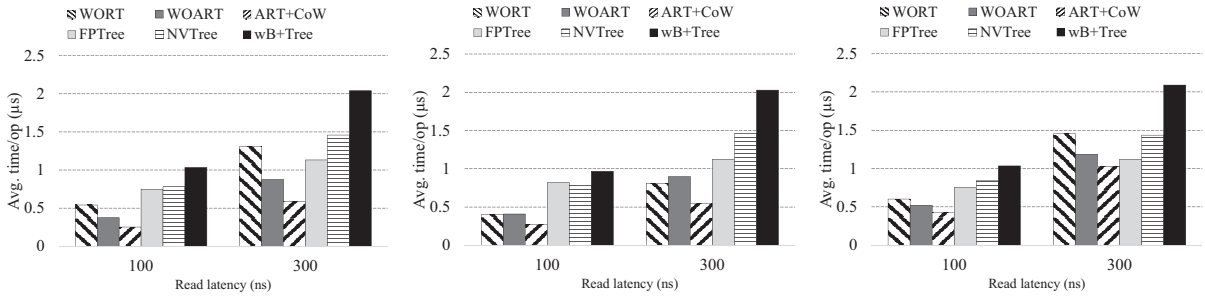
We also see that the B-tree variants are, in general, more sensitive to write latency increases. Column (b) in Table 1, which is the measured average number of cache line flushes per insertion, shows the reason behind this. We see that B-tree variants incur more cache flush instructions than the radix tree variants.

5.2 Search Performance

Figure 9 shows the average search time for searching 128 million keys for the three different distributions. Since search performance is not affected by the write latency of PM, we vary only the read latency using Quartz.

First, observe the left part of each graph, where both read and write latencies of PM are the same as DRAM. We see that the radix tree variants always perform better than the B-tree variants. In particular, ART+CoW performs the best for all cases. Since ART+CoW uses copy-on-write to ensure consistency, there is no additional indirection caused by the append-only strategy and the alignment of partial keys can be maintained. Therefore, ART+CoW is advantageous in tree searching compared to WOART where additional indirection and unsorted keys are employed to support the append-only strategy.

The reason that the radix tree variants perform better can be found in columns (c) and (d) in Table 1 that shows the average number of LLC misses and the average leaf node depth, respectively. We see that the overall performance is inversely proportional to the number of LLC misses and the depth of the tree. Notice that the depth of the tree is slightly higher for the radix tree variants. However, the number of LLC misses is substan-



(a) Dense (b) Sparse (c) Clustered

Figure 9: Search Performance Comparison

tially smaller, which compensates for the higher depth. The reason there are fewer LLC misses is because the radix tree can traverse the tree structure without performing any key comparisons, which incurs less pollution of the cache. Recall that, in contrast, B-tree variants must compare the keys to traverse the tree and that the keys may even be scattered across the entire node due to the append-only strategy. Hence, the B-tree variants more frequently access the entire range of the node causing more LLC misses.

Now, consider the right part of each graph, which is when read latency is increased to 300, in comparison with the left part. We see that WORT stands out in latency increase especially for Dense and Clustered workloads. This is due to the depth of the tree as WORT has the highest depth. Other than WORT, we see that WOART and ART+CoW perform better than the B-tree variants even with the increased read latency even though the internal nodes of FPTree and NVTree are still seeing DRAM latency.

5.3 Range Query Performance

Traditionally, B+-trees have an edge over radix trees on range queries as keys are sorted within the nodes and the leaf nodes are linked with sibling pointers. In contrast, in the radix tree, no sibling pointers exist, but the leaf nodes essentially contain keys that are implicitly sorted. Hence, in the radix tree, one may have to traverse up the descendant node(s) in order to find the next leaf node.

Our proposed radix tree variants are no different from traditional radix trees and do not do well for range queries. However, we note that the B-tree variants for PM also do not keep the keys sorted to reduce the overhead for ensuring consistency, which harms one of the key features of B+-trees. To see the effect of such changes we perform experiments for range queries.

Figure 10 shows the range query performance when keys in the range consisting of 0.001% and 0.01% of the 128M keys are queried. Here, we also present the performance of the original B+-tree for reference. We observe that the average time per operation of the three radix tree variants is over 5.8× and 6.4× than B+-tree for the 0.001% and 0.01% range, respectively. However, the performance gap declines for PM indexes. With respect to FPTree, NVTree, and wB+Tree, the average time per operation of the three radix variants is 3.0× and 2.8×, 1.8× and 1.8×, and 4.8× and 5.3× higher for 0.001% and 0.01% range queries, respectively. The reduction in difference is because B-tree variants need to rearrange the keys when servicing range queries.

5.4 Experiments with Memcached

In order to observe the performance of our proposed index structures for real life workloads, we implement all the tree structures used in the previous experiments within Memcached. Memcached is an in-memory caching system for key-value based database systems [22]. We remove the hash function and table of Memcached and embed the indexing structures. We also replace the bucket locking mechanism of the hash table with the global tree locking mechanism. The global tree locking mechanism locks the root of the tree in order to prevent conflicts between threads whenever an insertion operation is executed. We run mc-benchmark, which performs a series of insert queries (SET) followed by a series of search queries (GET) [23]. The key distribution is uniform, which randomly chooses a key from a set of string keys.

For these experiments, we use two connected machines with a 10Gbps Ethernet switch, one for Memcached and the other for running the mc-benchmark. We execute 128 million SET and GET queries with 4 threads and 50 threads, respectively. The machine used for Memcached is the same as described in Section 4 and the machine that runs the mc-benchmark is an Intel i7-4790 3.60GHz, 32GB DRAM and with Linux version 4.7.0.

For these experiments, all indexing structures are assumed to run entirely on PM even for the internal nodes of FPTree. This is because to consider the hybrid configuration, considerable changes must be made to Memcached, which we wanted to avoid as such changes may affect the outcome of the results. To support variable-sized string keys, wB+Tree and FPTree replace the keys in nodes with 8-byte pointers to the keys stored in a sep-

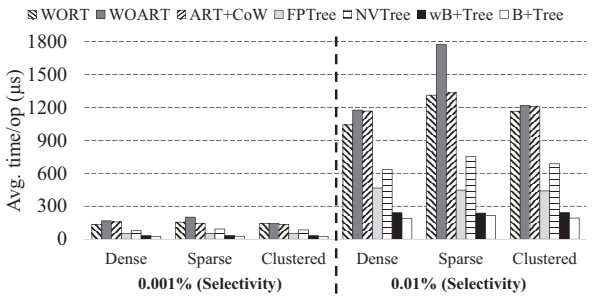


Figure 10: Range query over 128M

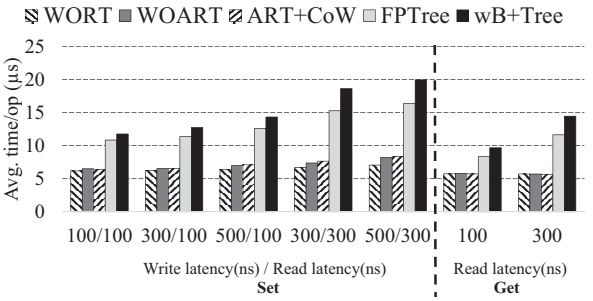


Figure 11: Memcached mc-benchmark performance

arate location [3, 15]. We follow this design in our experiments. For NVTree, as there is no mention on how to handle variable-sized keys, we omit its evaluation for Memcached [20].

The left part of Figure 11 shows the results for SET operations. We observe that the radix tree variants perform considerable better than the B-tree variants by roughly 50%. Other than the difference in structure, there are a couple of other factors that influences the difference. One is that there is the additional indirection and large key-comparing overhead, which was also observed by Chen and Jin [3]. Note that for radix tree variants, the overhead for key comparison is minimal. The other is the additional cache line flush required to store the keys in a separate PM area in the case of B-tree variants. This overhead does not exist for radix tree as variable-sized strings can be handled in essentially the same manner as integers. We also see that the effect of increased PM latency is also more profound for the B-tree variants.

The right part of Figure 11 shows the results for GET queries. Similarly to the SET query, the radix tree variants perform better than the B-tree variants. However, we also notice that the radix tree variants' results are the same for both 100 and 300 read latencies. We conjecture that this actually represents the network communication bottleneck. In spite of this, however, we see that the radix tree variants reduce wB+Tree latency by 41% and 60% and FPTree latency by 31% and 51% for 100 and 300 read latencies, respectively.

6 Summary and Conclusion

With the advent of persistent memory (PM), several persistent B-tree based indexing structures such as NVTree [20], wB+Tree [3], and FPTree [15] have recently been proposed. While B-tree based structures have been popular in-memory index structures, there is another such structure, namely, the radix tree, that has been less popular. In this paper, as our first contribution, we showed that the radix tree can be more appropriate as an indexing structure for PM. This is because its structure is determined by the prefix of the inserted keys dismissing the need for key comparisons and tree rebalancing. However, we also show that the radix tree as-is cannot be used in PM.

As our second contribution, we presented three radix tree variants adapted for PM. WORT (Write Optimal Radix Tree), which is the first variant that we proposed, employs a failure-atomic path compression scheme that we develop. WORT is optimal for PM, as is WOART, in the sense that they only require one 8-byte failure-atomic write per update to guarantee the consistency of the structure totally eliminating the need to make duplicates typically done via logging or copy-on-write (CoW) in traditional structures. WOART (Write Optimal Adaptive Radix Tree) and ART+CoW, the second and third variants, are both based on the Adaptive Radix Tree (ART) that was proposed by Leis et al. [11]. ART resolves the trade-off between search performance and node utilization found in traditional radix trees by employing an adaptive node type conversion scheme that dynamically changes the size of a tree node based on node utilization. However, ART in its present form does not guarantee failure atomicity. WOART redesigns the adaptive node types of ART and supplements memory barriers and cache line flush instructions to prevent processors from reordering memory writes and violating failure atomicity. ART+CoW, the third variant, extends ART to make use of CoW to maintain consistency.

Extensive performance studies showed that our proposed radix tree variants perform considerable better than recently proposed B-tree variants for PM such as NVTree, wB+Tree, and FPTree for synthetic workloads as well as in implementations within Memcached.

Acknowledgement

We would like to thank our shepherd Vasily Tarasov and the anonymous reviewers for their invaluable comments. We also thank our colleagues from the NECSST lab and Wook-Hee Kim for their numerous discussions that helped shape this research. We are also grateful to Ismail Oukid for his feedback in implementing the FPTree. This work was supported by Samsung Research Funding Centre of Samsung Electronics under Project Number SRFC-IT1402-09.

References

[1] GitHub. Quartz: A DRAM-based performance emulator for NVM *https://github.com/HewlettPackard/quartz*.

[2] M. Boehm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient In-Memory Indexing with Generalized Prefix Trees. In *Proceedings of the Database Systems for Business, Technology, and Web (BTW)*, 2011.

[3] S. Chen and Q. Jin. Persistent B+-Trees in Non-Volatile Main Memory. In *Proceedings of the VLDB Endowment (PVLDB)*, 2015.

[4] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[5] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[6] 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 9th ACM European Conference on Computer Systems (EuroSys)*, 2014.

[7] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware Logging in Transaction Systems. In *Proceedings of the VLDB Endowment (PVLDB)*, 2014.

[8] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010.

[9] W. -H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, 2016.

[10] E. Lee, H. Bahn, and S. H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[11] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the IEEE 29th International Conference on Data Engineering (ICDE)*, 2013.

[12] 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 ACM Conference on Timely Results in Operating Systems (TRIOS)*, 2013.

[13] D. Narayanan and O. Hodson. Whole-System Persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[14] J. Ou, J. Shu, and Y. Lu. A High Performance File System for Non-Volatile Main Memory. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, 2016.

[15] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.

[16] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, 1999.

[17] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. 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)*, 2011.

[18] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 15th Annual Middleware Conference (Middleware)*, 2015.

[19] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[20] J. Yang, Q. Wei, C. Chen, C. Wang, and K. L. Yong. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[21] Intel Corporation Intel® 64 and IA-32 Architectures Software Developer's Manual. *http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html*.

[22] MEMCACHED What is Memcached? *https://memcached.org*.

[23] GitHub Memcache port of Redis benchmark. *https://github.com/antirez/mc-benchmark*.

[24] Technische Universität München The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. *https://db.in.tum.de/ leis/*.

[25] J. Hyun Kim, Young Je Moon, and Sam H. Noh. An Experimental Study on the Effect of Asymmetric Memory Latency of New Memory on Application Performance. In *Proceedings of the 24th IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2016.

[26] J. Corbet Trees I: Radix trees. *https://lwn.net/Articles/175432/*.

[27] D. R. Morrison PATRICIA –Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM (JACM)*, 1968.

[28] D. E. Knuth The Art in Computer Programming: Sorting and Searching, Vol. 3. *Pearson Education*, 1998.

SHRD: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizng in Device

Hyukjoong Kim[1], Dongkun Shin[1], Yun Ho Jeong[2], and Kyung Ho Kim[2]

[1]Sungkyunkwan University, Korea
[2]Samsung Electronics, Korea

Abstract

Recent advances in flash memory technology have reduced the cost-per-bit of flash storage devices such as solid-state drives (SSDs), thereby enabling the development of large-capacity SSDs for enterprise-scale storage. However, two major concerns arise in designing SSDs. The first concern is the poor performance of random writes in an SSD. Server workloads such as databases generate many random writes; therefore, this problem must be resolved to enable the usage of SSDs in enterprise systems. The second concern is that the size of the internal DRAM of an SSD is proportional to the capacity of the SSD. The peculiarities of flash memory require an address translation layer called flash translation layer (FTL) to be implemented within an SSD. The FTL must maintain the address mapping table in the internal DRAM. Although the previously proposed demand map loading technique can reduce the required DRAM size, the technique aggravates the poor random performance. We propose a novel address reshaping technique called sequentializing in host and randomizing in device (SHRD), which transforms random write requests into sequential write requests in the block device driver by assigning the address space of the reserved log area in the SSD. Unlike previous approaches, SHRD can restore the sequentially written data to the original location without requiring explicit copy operations by utilizing the address mapping scheme of the FTL. We implement SHRD in a real SSD device and demonstrate the improved performance resulting from SHRD for various workloads.

1 Introduction

In recent times, the proliferation of flash-memory-based storage such as solid-state drives (SSDs) and embedded multimedia cards (eMMCs) has been one of the most significant changes in computing systems. The cost-per-bit of flash memory has continued to fall owing to semiconductor technology scaling and 3-D vertical NAND flash technology [32]. As a result, SSD vendors currently provide up to 16 TB of capacity.

Flash memory has several characteristics that must be carefully handled. In particular, its "erase-before-write" constraint does not permit in-place update. In order to handle the peculiarities of flash memory, special software–called a flash translation layer (FTL) [14, 20]–is embedded within flash storage systems. When a page of data must be updated, the FTL writes the new data to a clean physical page and invalidates the old page because in-place overwrite is prohibited in flash memory. Therefore, the logical address and physical address of a flash page will be different. The FTL maintains a mapping table for the logical-to-physical (L2P) address translation. When the SSD has an insufficient number of clean pages, garbage collection (GC) is triggered to reclaim invalid pages. GC selects victim blocks to be recycled, copies all the valid pages in the recycling blocks to another block, and erases the recycling blocks.

Recent flash storage devices adopt page-level address mappings instead of block-level schemes in order to provide higher performance. Page-level mappings permit requests to be serviced from any physical page on flash memory, whereas block-level mappings restrict the physical page location of a request based on its logical address. However, the finer-grained mapping scheme requires a large L2P mapping table. Typically, the size of a page-level mapping table is 0.1% of the total storage capacity because the length of the address translation data for a 4 KB page is 4 bytes.

The mapping table is accessed by every I/O request; therefore, in order to achieve high performance, the entire table must be loaded into an internal DRAM of the SSD. Thus, as an example, 8 TB of SSD requires 8 GB of DRAM for the mapping table. More than 4 GB of DRAM requires a 64-bit processor, which is hardly acceptable to embedded systems. In addition, a larger capacity of DRAM system composed of multiple DRAM modules requires DRAM controller to handle more ad-

dress lines, which will increase the controller cost and DRAM access latency. Furthermore, a large DRAM-equipped SSD will have a high power consumption and product cost. Therefore, the large mapping table is the most critical hurdle in increasing the capacity of SSDs.

In order to resolve this problem, on-demand map loading schemes such as DFTL [14] were proposed. Instead of maintaining all the address translation entries in DRAM, DFTL dynamically loads/unloads the page-level mapping entries to/from a small DRAM according to the workload access patterns. The entire image of the page-level mapping table is stored in reserved flash memory blocks called *map blocks*. If storage workloads exhibit significant temporal locality, the performance of the demand-loading approach will be similar to the performance of the traditional page-level mapping scheme that loads all the mapping entries in DRAM. In addition, DFTL can utilize spatial locality by loading multiple logically contiguous mapping entries at the miss of a mapping entry. However, real scenarios have many random workloads with low localities. DFTL is vulnerable to these workloads; this limitation is a critical drawback of DFTL in real computing systems.

In this paper, we focus on random write rather than read because writes occupy about 70% at server storage workloads as observed in [28]. In addition, whereas read requests are controllable by several optimization techniques such as page caching or prefetching, the immediate handling of write requests are unavoidable in many cases because they are generated for data durability.

One possible solution for random write workloads is to use log-structured file systems (LFSs) (e.g., NILFS [21] and F2FS [22]) or copy-on-write file systems (e.g., btrfs [33]) because they generate sequential write requests using an out-of-place update scheme. The key-value stores based on LSM-Trees (e.g., LevelDB [7] and RocksDB [9]) and log-structured databases (e.g., RethinkDB [8]) also remove random writes. However, such log-structured file systems or applications suffer from garbage collection or compaction overhead and require many metadata block updates owing to their out-of-place update scheme. Moreover, file system and SSD perform duplicated garbage collections [35].

Another solution is to translate random write requests into sequential write requests in the block device driver. ReSSD [26] and LSDM [37] sequentialize random write requests by providing an additional address translation layer in the storage device driver and temporarily writing the sequentialized data to a reserved area in the storage device. We call this operation *sequentializing*. However, when the reserved area becomes full, the temporarily written data must be moved to the original location; this operation is called *randomizing (restoring)*. The randomizing operation results in additional storage traffic. Further, it eventually sends random writes to storage.

In this paper, we propose a novel scheme, called sequentializing in host and randomizing in device (SHRD), to reshape the storage access pattern. In order to reduce the map-handling overhead in a DFTL-based SSD, the proposed scheme sequentializes random write requests into sequential requests at the block device driver by redirecting random write requests to the reserved storage area. Therefore, it is analogous to the previous device-driver-level sequentializing approach. However, in the previous schemes, randomizing is achieved by performing explicit move operations in the host system; thus, they use the "sequentializing in host and randomizing in host (SHRH)" approach. Our scheme conducts the randomizing in storage device by changing only the logical addresses of the sequentialized data.

SHRD has several advantages. It can improve random write performance by reducing map-loading or command-handling overheads and by increasing the utilization of parallel units in an SSD. It can also improve the lifetime of an SSD. The reduction in the number of map update operations results in a reduction in the number of program and erase operations on flash memory blocks, thus minimizing the write amplification ratio.

This study makes the following specific contributions. (1) We propose and design a novel request reshaping scheme called SHRD, which includes a storage device driver and the FTL of an SSD. The idea of SHRD is to improve the spatial locality for FTL mapping table accesses by logging random requests in the storage and reordering these requests. (2) We implement the proposed SHRD scheme by modifying the firmware of an SSD device and the Linux SCSI device driver. Unlike many other studies based on SSD simulators and I/O traces, our scheme is demonstrated and verified by using a real SSD device. (3) We use several enterprise-scale workloads to demonstrate the improved performance achieved by SHRD. We observe that in comparison with DFTL, the performance of SHRD is 18 times better for a random-write dominant I/O workload and 3.5 times better for a TPC-C workload.

The remainder of this paper is organized as follows: In Section 2, the FTL schemes are introduced; in Section 3, the motivation and main idea are presented; in Section 4, the proposed SHRD scheme is described in detail; the experimental results are presented in Section 5; previous studies on improving random write performance are presented in Section 6; and the conclusion of this study is described in Section 7.

2 Backgrounds

Generally, the page-level mapping FTL maintains the entire L2P mapping table in the internal DRAM of an

SSD. In order to service incoming write requests, FTL allocates active flash blocks, writes incoming data sequentially in the physical pages of the active blocks, and updates the L2P mapping table in DRAM. If the active blocks are full with user data, the FTL flushes dirty mapping entries into the reserved area of flash memory called map blocks and then allocates new active blocks. If a sudden power-off occurs before the map flush operation, the power-off recovery (POR) operation of an SSD scans only the active blocks and rebuilds the L2P mapping table with the logical page numbers (LPNs) stored in the out-of-bound (OOB) area of flash pages; this OOB area is a hidden space reserved for FTL-managed metadata.

In order to reduce the mapping table size, several solutions have been proposed such as hybrid mapping schemes [20, 25] and extent mapping schemes [27, 16, 31]. Although these solutions can reduce the mapping table size significantly, they are vulnerable to random write workloads. Hybrid mapping scheme shows considerable performance degradation when the log blocks and normal data blocks are merged. Extent mapping scheme must split a large extent into multiple smaller extents when the random updates are requested.

Rather than reducing the size of the mapping table, we can use an on-demand map loading scheme such as DFTL [14]. This scheme uses a small amount of DRAM for the cached mapping table (CMT), which has only a subset of the entire page-level mapping table that is maintained in the map blocks of flash chips, as shown in Figure 1. For each I/O request, DFTL determines the physical page number (PPN) to be accessed based on the mapping entries in the CMT. The map blocks must be read and written as page units; therefore, multiple contiguous mapping entries constitute a *map page*, and DFTL loads/unloads mapping entries in map page units. For example, if each mapping entry has a size of 4 bytes, 4 KB of map page can contain 1024 logically contiguous mapping entries. Owing to the page-level loading scheme, DFTL requires spatial locality as well as temporal locality in the storage access workload. In order to handle a map miss, one victim map page must be written in the map block if the map page is dirty and one demanded map page must be read from the map block. Therefore, the demand map loading scheme demonstrates poor performance for a random workload due to additional storage traffic.

The map-miss handling in DFTL decreases the utilization of the parallel units of SSD. In order to provide high I/O bandwidth, multiple flash chips in SSD should be accessed simultaneously via parallel channels and chip interleaving. However, if several requests generate map misses simultaneously and the missed map entries must be read from a same chip, the handling of the requests must be serialized and thus several flash chips will be idle during the map-miss handling.

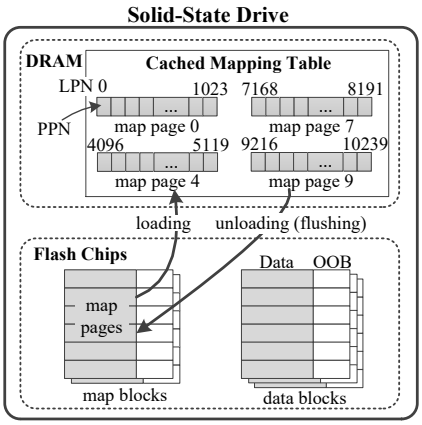


Figure 1: Demand map loading.

3 Main Idea

The poor random write performance of SSD can be attributed to various reasons. The first reason is the mapping-table handling overhead. If the SSD uses a demand map loading scheme such as DFTL, random requests will result in frequent CMT misses, as explained in Section 2. Even if the SSD can load all the mapping entries into the DRAM, the random writes will generate many dirty map pages in the DRAM. When the SSD periodically flushes dirty map pages, many pages will be updated in the map blocks.

The second reason is the request-handling overhead. The occurrence of many small requests increases the request traffic between the host computer and the SSD and increases the interrupt-handling overhead of the host computer. In order to solve this problem, eMMC adopts the packed command from the version 4.5 standard; thus, multiple requests can be merged into one packed command [1]. However, the current SATA/SAS protocol does not support the request merging.

The final reason is the cost of GC. GC selects a victim flash block having a small number of valid pages in order to reduce page-copy operations. While a sequential write workload generates many completely invalid blocks, a random write workload distributes invalid pages among several flash blocks, thus making it difficult to find a low-cost GC victim block. The overhead of GC can be mitigated by using hot and cold separation algorithms [10, 24].

From among the several reasons for poor random performance of flash storage, we focus on the mapping-table handling overhead and the request-handling overhead because they are the major causes and do not have any solutions currently. SHRD can reduce the mapping-table handling overhead by improving the spatial locality of a workload, and can reduce the request-handling overhead

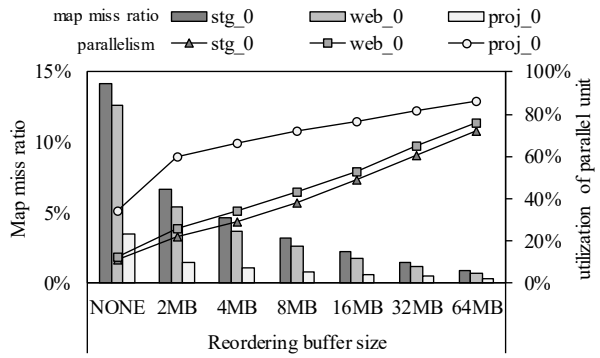


Figure 2: Effect of request reordering.

by packing multiple requests into one large request.

Even if a workload has low spatial locality within a short time interval, it can have high spatial locality within a long time interval. If a memory space is available for request buffering, the map miss ratio can be reduced by reordering requests in the order of LPN to improve spatial locality. Figure 2 shows the simulation results for the map miss ratio of DFTL and the utilization of parallel flash chips in an SSD. The MSR-Cambridge server workloads [4] are used for input traces. We assume that the CMT size of DFTL is 128 KB and a maximum of 64 flash pages can be accessed in parallel via multi-channel, multi-bank, and multi-plane mechanisms. The x-axis shows the reordering buffer size. In each experiment, all the I/O requests are partitioned into several groups in the order of request arrival time such that the total request size of each group is equal to the reordering buffer size. Then, the requests within a group are sorted in the order of LPN. Such transformed input traces are provided to an SSD simulator, which uses the DFTL algorithm. In comparison with the original trace, the map miss ratios significantly decrease as the size of the reordering buffer increases because the spatial locality improves. The utilization of the parallel units of the SSD also improves.

In order to support such a request reordering, we must allocate the reordering buffer in either host system or SSD. The host-level buffering has several problems. First, a large memory space is required for data buffering. Second, applications may invoke synchronous operations such as *fsync*() in order to ensure instant data durability. Most database systems rely on the fsync system call to be assured of immediate data durability. Therefore, host-level data buffering is impractical. The large memory allocation within SSD also does not correspond with our motivation.

Our idea is to buffer only mapping entries instead of request data. The mapping entries are buffered in a small size of host memory, and the data are directly written at SSD without buffering. To obtain the same effect of request reordering, SHRD writes random write requests in a reserved space in the SSD called random write log buffer (RWLB). This step is *sequentializing*. The RWLB is a logical address space; therefore, an SSD can allocate any physical flash blocks for the RWLB. SHRD assigns a temporary logical page number (tLPN) sequentially for each original logical page number (oLPN). tLPNs are allocated from the RWLB address space. The write operations to the RWLB can be performed with large and sequential write requests that invoke little mapping-table handling overhead because the original addresses are sequentially mapped to the addresses of the RWLB. The storage device driver in host computer maintains the mapping information between oLPN and tLPN in order to redirect read requests, and thus SHRD does not require any change on host file systems.

When the logical address space of the RWLB is exhausted, the buffered mapping entries in host system are sent to SSD after being reordered based on their oLPNs. SSD restores the sequentialized data into the original addresses with the mapping entries. This step is *randomizing (restoring)*, which modifies only the L2P mapping table of the SSD instead of moving the sequentialized data. Although the randomizing operation updates many L2P mapping entries, the map-loading overhead is minimized because the randomizing operations of sequentialized pages are performed in the order of oLPN, thus improving the spatial locality during map update operations.

4 SHRD Scheme

4.1 Overall Architecture

The SHRD architecture consists of an SHRD device driver (D/D) in the host system and SHRD-supporting firmware in the SSD, as shown in Figure 3. The file system sends a write request, which consists of the target logical page number (oLPN), the size, and the memory pointer to user data. The SHRD D/D receives the write request and checks whether sequentializing is required based on the write request size. The sequentializing and randomizing involve special operations; therefore, they result in some overhead. Considering the trade-off between performance gain and the overhead caused by SHRD, only small and random requests must be sequentialized. If the request size does not exceed a predefined threshold value called *RW threshold*, the `sequentializer` assigns sequential temporary addresses (tLPNs) to the request.

The sequentialized write requests are sent to the SHRD-supporting SSD via a special command, called *twrite*, which sends both the oLPN and the assigned tLPN. The SSD writes the sequentialized data into the blocks assigned to the RWLB. The

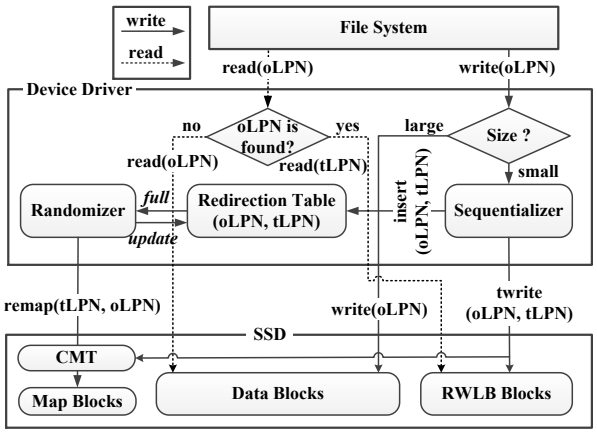


Figure 3: SHRD architecture.

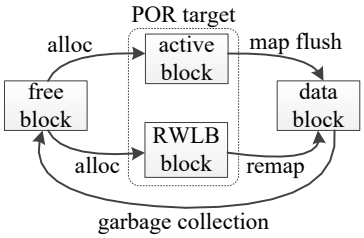


Figure 4: Life cycle of flash memory block.

`sequentializer` also inserts the mapping information into the `redirection table`; read requests use this table to redirect the original address to the sequentialized temporary address.

If the logical address space of the RWLB is exhausted, the `randomizer` restores the original logical addresses of the sequentialized data via the *remap* command. When the SSD receives the *remap* command, it changes only the internal L2P mapping table without changing the physical location of the data.

The SHRD-supporting FTL is similar to DFTL, but it can handle SHRD-supporting special commands such as *twrite* and *remap*. It also manages several different types of flash memory blocks such as data block, active block, RWLB block, and map block. Figure 4 shows the life cycle of a flash memory block. The FTL allocates active blocks and RWLB blocks for the normal data region and RWLB, respectively. The mapping entries of the pages in these regions are updated only in the CMT; therefore, the POR operation must scan the OOB area of these blocks to rebuild the mapping information. When the mapping entries of an active block are flushed into the map blocks, the active block is changed to a data block and it can be a victim of GC. If all the pages in an RWLB block are remapped to its original logical address, the block is changed to a data block. An RWLB block cannot be a victim for garbage collection because the mapping entries of its pages are not fixed yet.

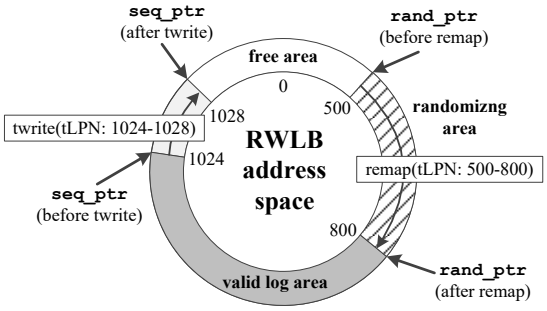


Figure 5: RWLB address management.

4.2 Sequentializing in Host

In order to manage the address space of the RWLB, which is hidden from the file system, SHRD maintains two address pointers to the RWLB address space, as shown in Figure 5. The RWLB is managed in a circular manner. `seq_ptr` is the start location for sequential address allocation, and it is incremented by sequentializing. `rand_ptr` is the start location of sequentialized pages that must be randomized, and it is incremented by randomizing. Therefore, the redirection table in the host has the mapping information of the pages that are written to the address range from `rand_ptr` to `seq_ptr`.

The sequentialized write requests are not immediately sent to the SSD. In order to minimize the command-handling overhead, multiple sequentialized write requests are packed into one write request, as shown in the example in Figure 6. The request packing can also reduce the interrupt handling overhead. The sequentialized write requests have contiguous address values; therefore, the packed write request has only one start address value. The block layer of the Linux operating system supports a maximum of 512 KB of write requests; therefore, the requests can be packed until the total packed request size does not exceed 512 KB. If no request is present in the I/O scheduler queue, the sequentializer sends the packed requests to the storage device immediately instead of waiting for more requests even though the size of packed requests is less than 512 KB. The request packing also halts if a higher priority request such as *flush* arrives. During the random write packing operation, if an examined request is not sequentialized (e.g., a read or large write request), SHRD sends first the normal request to storage and then the packing operation for random write requests continues. As more requests are packed into a single request, the command-handling overhead will be reduced. However, the request packing does not affect the map-handling overhead in SSD because subsequent *twrite* requests will be assigned with sequential tLPNs.

SHRD-supporting SSD must handle the sequentialized and packed requests differently from normal write requests; therefore, we need to implement *twrite* as a special command. The SATA storage interface provides

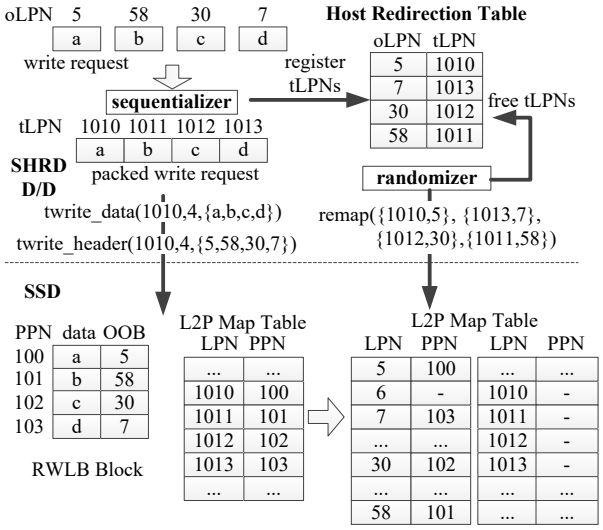


Figure 6: Example of SHRD operations.

vendor commands to support the extension of its command set. However, a vendor command cannot be used in the command queueing mode, and thus, the performance can degrade significantly. In order to overcome such a problem, we implemented several special commands of SHRD operations by utilizing the legacy write command. The SHRD-supporting special commands are exactly same as the normal write command except that their target address values are beyond the logical address space of storage device. Depending on the address value of write command, SSD firmware interprets it as a particular special command. The information of the special command (e.g., oLPN and tLPN) is transferred via data blocks of the write command. Therefore, no hardware or software changes are required in the SATA interface protocol.

Two SHRD commands (i.e., *twrite_header* and *twrite_data*) are used to write the sequentialized data to the RWLB of the SSD. *twrite_header* contains the address mapping information in its 4 KB of data, i.e., start tLPN, page count, and array of oLPNs, as shown in Figure 6. The sequentializing information of a maximum of 128 pages can be transferred via 4 KB of data.

After a *twrite_header* command is sent, the corresponding *twrite_data* command is sent in order to transfer the packed user data (maximum of 512 KB), which will be written to the RWLB blocks. The *twrite_data* command contains the target logical address value of tLPN, which is assigned by the sequentializer. The SSD firmware determines a PPN for each tLPN and inserts the mapping entry (tLPN, PPN) into the L2P mapping table. Each oLPN value transferred by the *twrite_header* command is written into the OOB area of the corresponding flash page. The oLPN will be used for POR when the page is in the RWLB block, and it will be used by GC after its block is changed to a data block.

After the completion of the *twrite* command, the sequentializer inserts the address translation entry between oLPN and tLPN into the redirection table. Because the remap entry will be used by subsequent read requests and remap operations, the redirection table maintains both the oLPN-to-tLPN and tLPN-to-oLPN map entries. If an old mapping entry for the same oLPN exists, the redirection table is updated and a *trim* command for the previous tLPN can be optionally sent to the SSD in order to inform it about the invalidation of the data at the address of the tLPN. The size of the redirection table is determined by the size of the RWLB. For example, in our implementation, when the RWLB size is 64 MB, the size of the redirection table is 256 KB and the table can contain 16K mapping entries.

During the handling of *twrite* commands, normal read or write commands can be transferred to SSD if their target addresses are not related with a pending *twrite* command. However, any dependent read request must wait for the completion of the related *twrite* command. In addition, any dependent write request can be processed only after the redirection map table is updated by *twrite*.

4.3 Read Redirection

For a read request from the file system, the SHRD D/D searches for the target logical page numbers, oLPNs, in the redirection table. If the oLPNs are found, the sequentializer redirects the read request to the RWLB by changing the target address to the corresponding tLPNs. Otherwise, the original addresses are used. The search time in the redirection table can increase read request latencies. In order to minimize the search time, the oLPN-to-tLPN mapping entries in the redirection table are maintained with a red-black (RB) tree; thus, the search time grows at the rate of $O(\log n)$. A complex case of read handling is the scenario in which the target logical address range has been partially sequentialized. In this case, the read request must be split into multiple sub-read requests, and only the sub-reads for sequentialized data must be sent to the RWLB. After the completion of all the sub-read requests, the original read request will be completed.

4.4 Randomizing in Device

When the RWLB address space is exhausted by sequentializing operations, the SHRD D/D performs the randomizing operation to reclaim the allocated addresses of the RWLB by sending a special command called *remap*, which restores the oLPNs of sequentialized pages. First, the randomizer selects the address range to be reclaimed; this address range starts from the `rand_start` pointer of the RWLB. Then, the randomizer searches the redirection table for the mapping entries whose tLPN values are included in the randomizing address range and creates

the remapping entries for randomizing. The search operation accesses the tLPN-to-oLPN mapping entries in the redirection table. The generated remapping entries, each of which is represented by (tLPN, oLPN), are sorted. By sending the oLPN-sorted remapping entries to the SSD, the spatial locality of CMT access is improved and the CMT miss ratio can be reduced.

The remapping entries are sent as a 4 KB of data in the *remap* command. The size of one remapping entry is 8 bytes, and one remap command can transfer a maximum of 511 remapping entries. The remaining space is used to store the number of remapping entries. Therefore, the SHRD D/D sends multiple remap commands incrementally during randomizing, and other irrelevant read/write requests can be sent to the SSD between remap commands. However, normal requests can be delayed within the SSD if there are many pending remap requests, because each remap command modifies a large number of address mapping entries of the SSD and normal requests cannot be handled simultaneously with the remap command. To solve this problem, two optimization techniques are used. First, we can reduce the maximum number of remapping entries for a single remap command. Second, the maximum number of remap commands which are pending in the SSD can be limited. In our implementation, these two numbers are limited to 128 and 1, respectively. Using these techniques, the delay of normal request can be limited. If a normal request is relevant to a remap command, it must wait for the completion of the remap command.

When the SSD receives a remap command, for each remapping entry (tLPN, oLPN), it inserts the mapping entry (oLPN, PPN) into the L2P mapping table. The PPN is the physical location of the data to be randomized, and it can be obtained by reading the mapping entry (tLPN, PPN) from the L2P mapping table. Therefore, two map pages must be accessed for randomizing one physical page–the map page that contains (tLPN, PPN) and the map page that contains (oLPN, PPN). However, the sorted remapping entries in a remap command will modify only a small number of map pages because it is quite probable that consecutive remapping entries will access a same map page. After all the pages in an RWLB block are randomized, the block is changed to a normal data block. In order to change the block information, dirty mapping entries of the CMT must be flushed into the map blocks, and then, the block change information must be written to the map blocks. After the completion of the remap command, the randomizer removes the mapping entry (oLPN, tLPN) from the redirection table. Optionally, host can send a *trim* command for tLPN.

The special commands of SHRD have ordering constraints. The *twrite_data*(tLPN) command must be sent after the corresponding *twrite_header*(oLPN, tLPN) command is completed. The *remap*(oLPN, tLPN) command must be sent after the corresponding *twrite_data*(tLPN) command is completed. If the SHRD D/D issues a command before its dependent command is completed, these commands can be reordered by the command queueing scheme of the storage interface, thus potentially violating the consistency of the SSD data. Owing to the ordering constraints, SHRD operations result in a small amount of performance overhead in our current implementation. If the SSD firmware can guarantee the ordering between dependent commands, the overhead can be reduced. We will consider this requirement in future work.

4.5 Power-Off-Recovery and Garbage Collection

The redirection table is maintained in the host DRAM; therefore, a sudden power-off can result in a loss of all the sequentializing mapping information. In order to address this issue, we use the *autoremap* technique during POR. As shown in Figure 6, each flash page of sequentialized data contains the oLPN in its OOB area. Therefore, the mapping between PPN and oLPN can be obtained. The POR operation scans all the RWLB blocks and performs the randomizing operation by using the PPN-to-oLPN mappings. The autoremap operation is similar to the randomizing by the SHRD D/D, except that autoremap is invoked internally by the SSD POR firmware. Therefore, only the SSD mapping table is modified without changing the physical locations. The POR operation must scan all the RWLB blocks; therefore, the RWLB size influences the POR time. However, the increased POR time is not critical because sudden-power-offs are rare and the POR is performed at the device booting time. In addition, in our implementation, the RWLB block scan time is less than 0.7 seconds for 64 MB of RWLB. After the autoremap operation, the blocks in the RWLB are changed to normal data blocks and they can be victims for GC.

For each valid page in a GC victim block, GC must copy the pages to a free block and modify the corresponding mapping entry in the L2P mapping table. GC uses the oLPN in the OOB area in order to access the L2P mapping entry of the copied page. If a selected GC victim block was initially allocated as a RWLB block, there can be many map misses in the CMT during GC, because the valid pages of the victim block were written by random write requests. To handle the map misses during GC, we use a map logging technique. If the mapping entry of a copied page is missed from the CMT, the proposed map logging technique does not load the corresponding map page from a map block immediately. Instead, the new mapping entry is written in the map log table (MLT) in DRAM. For the purpose, a small amount

of memory space is reserved in DRAM. After several GC victim blocks are reclaimed, the MLT will have many mapping entries to be written to the map pages. If the MLT is full, GC sorts the mapping entries based on the LPN value to improve the spatial locality when accessing map pages, and flushes all the mapping entries of the MLT into the map pages.

The idea of map-logging technique is similar to that of SHRD technique. Therefore, the map logging may be also applied to normal write request handling. However, SSD must reserve a large amount of memory space for the MLT to support normal write requests. In addition, the map logging technique cannot reduce the request handling overhead. Therefore, it is better to use SHRD technique for normal requests.

5 Experiments

5.1 Experimental Setup

We implemented an SHRD-supporting SSD by modifying the firmware of the Samsung SM843 SSD, which is designed for data center storage. For our experiments, the parallel units in the SSD were partially enabled–4 channels, 4 banks/channel, and 4 planes/bank. We implemented two 4-KB-page-level mapping FTLs, demand-loading FTL (DFTL) and SHRD-supporting FTL (SHRD-FTL). They use the CMT scheme with a DRAM whose size is less than that of the entire mapping table. Although the total storage capacity is 120 GB, the device provides only 110 GB of address space to the host system. The remaining over-provisioning space is used for GC and for the RWLB. The host computer system used a Linux 3.17.4 kernel and was equipped with Intel Core i7-2600 3.4 GHz CPU and 8 GB DRAM. The SHRD device driver was implemented by modifying the SCSI device driver of the Linux kernel. For the simplicity of the system, the trim/discard commands are not enabled.

In order to demonstrate the performance improvement achieved by the SHRD scheme, several server benchmarks were used: fio [3], tpcc [6], YCSB [13], postmark [19], and fileserver/varmail workloads of filebench [2]. In the case of the fio random write workload, four concurrent threads were generated and each thread wrote 8 GB of data with 4 KB of random write requests in 32 GB of address space. The tpcc workload was generated by percona's tpcc-mysql. The DB page size was configured to 4 KB, the number of warehouses was 120, and the number of connections was 20. In the case of the YCSB workload, MySQL system and the update-heavy workload (i.e., Workload A), which has 20% reads and 80% updates, were used. The number of transactions of postmark workload was 100,000. In the cases of the fileserver and varmail workloads, the number of files was configured as 200,000. Other parameters used the default options. The benchmarks were run at EXT4 filesystem by default. Table 1 presents the characteristics of each workload, which includes the logical address space, the average size of write requests, the portion of write requests, and the frequency of *flush* command generated by *fsync* calls (the average number of write requests between two flush commands).

Table 1: Workload Characteristics

benchmark	logical space (GB)	avg. write size (KB)	write portion	writes btwn flushes
fio(RW)	32	4.8	100%	61.6
tpcc	16	14.8	60%	18.9
YCSB	24	24.1	63%	5.7
postmark	20	72.0	90%	2743.7
fileserver	24	65.6	65%	2668.0
varmail	10	12.8	44%	1.0

Table 2: Statistics on SHRD Operations

benchmark	small req. portion	requests /twrite	pages /twrite	updated map pages /remap
fio(RW)	100%	30.3	31.42	5.74
tpcc	58%	3.95	6.76	5.33
YCSB	57%	2.45	9.71	9.76
postmark	88%	11.54	57.38	1.34
fileserver	33%	9.33	57.3	1.66
varmail	97%	1.19	3.25	1.24

For all the following experiments, the default sizes of RWLB, RW threshold, and CMT are 64 MB, 128 KB, and 1 MB, respectively, unless they are specified.

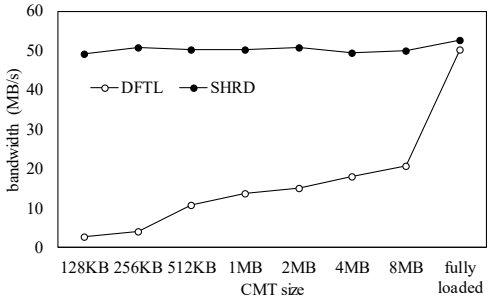
5.2 Experimental Results

Performance improvement with SHRD Table 2 shows several statistics on SHRD operations for each workload, i.e., the portion of sequentialized small write requests, the average number of packed requests/pages per single twrite command, and the average number of updated map pages per single remap command. Although each remap command contains a maximum of 128 remapping entries, it updated less than 10 map pages at all workloads due to the oLPN-sorted map access. Table 3 shows the portions of three reasons for request packing interruption during sequentializing, and the portion of small twrite requests that have less than 32 KB of packed requests.

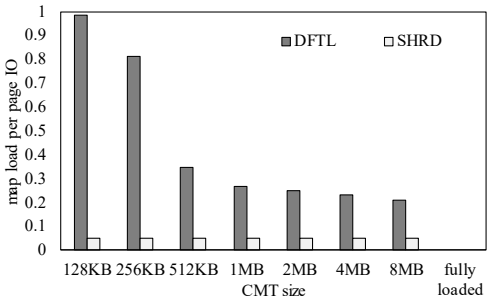
Figure 7(a) compares the performance of fio random write benchmark under the SHRD and DFTL schemes for different values of the CMT size. Because the logical address space of fio benchmark is 32 GB, if the CMT size is 32 MB, all the mapping entries of the workloads can be fully loaded into the CMT; thus, there is no map-miss handling overhead. If CMT size is less than 32 MB, the CMT can cache the mapping entries for 32 GB of address space partially. For example, 1 MB of CMT can contain only 3.1% of the entire mapping entries of the workload. Figure 7(b) shows the average number of

Table 3: Reasons for packing interruption

benchmark	reasons			small twrites
	no_req	flush	full	(< 32 KB)
fio(RW)	66%	31%	4%	6%
tpcc	50%	50%	0%	53%
YCSB	56%	39%	5%	86%
postmark	81%	0%	19%	1%
fileserver	51%	0%	49%	1%
varmail	21%	79%	0%	83%



(a) performance comparison

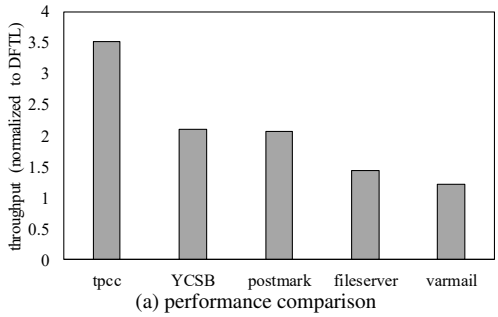


(b) map miss comparison

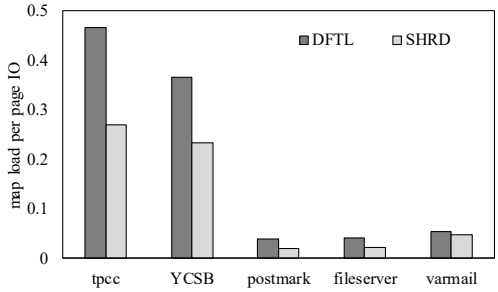
Figure 7: Fio random write workload results.

map page misses per write operation. A value of 1 indicates that every write operation triggers one additional map page load operation. As the CMT size decreases, DFTL shows worse performance because the number of map page loads increases. However, the performance of SHRD is similar to the performance when map pages are fully loaded into DRAM, irrespective of the CMT size. Even when the mapping table is fully loaded, SHRD shows a better performance than DFTL owing to the reduction on request handling overhead.

Figure 8(a) compares the performance of the SHRD and DFTL schemes for several real workloads under a fixed size of CMT (i.e., 1 MB). For all benchmark workloads, SHRD improved the I/O performance in comparison with DFTL. For the database workloads (i.e., tpcc and YCSB), SHRD demonstrated significantly better performance than DFTL because they are write-dominant workloads. In addition, the average size of write requests is small at these workloads; thus many write requests were sequentialized as shown in Table 2. However, DB workloads generate *fsync* calls frequently; therefore, the request packing at sequentializing was frequently interrupted by the *flush* command, as shown in



(a) performance comparison



(b) map miss comparison

Figure 8: Real workload results.

Table 3, and a smaller number of requests were packed for the DB workloads in comparison with other workloads.

The postmark workload is more write-dominant than the DB workloads, and many requests were sequentialized. However, the performance gain was similar to that of YCSB workload because its average write request size is larger. The fileserver workload generates many large requests; therefore, DFTL also show good performance. In addition, the fileserver workload includes many read requests. Because SHRD cannot directly improve read performance, the performance gain was small. The varmail workload generates *fsync* frequently; therefore, a smaller number of requests were packed by the sequentializer as shown in Table 2. Then, many special commands of SHRD must be transferred, thus degrading the performance owing to the ordering constraints explained in Section 4.4. In addition, the varmail workload is read-dominant, thus its performance improvement was small. We used the varmail workload as an adverse workload to check the SHRD overhead. Nevertheless, SHRD showed a better performance than DFTL even for the workload.

Figure 9 shows the map entries accessed by SHRD-FTL during the execution of postmark benchmark. Although the original postmark workload has many small and random write requests, SHRD changed them to sequential write requests to the RWLB address region (blue dots). Therefore, only a small number of accesses occurred on the normal address region by large sequential requests (black dots). During the periodic randomizing operations, the original addresses and the temporary addresses were accessed (read dots). The map entry ac-

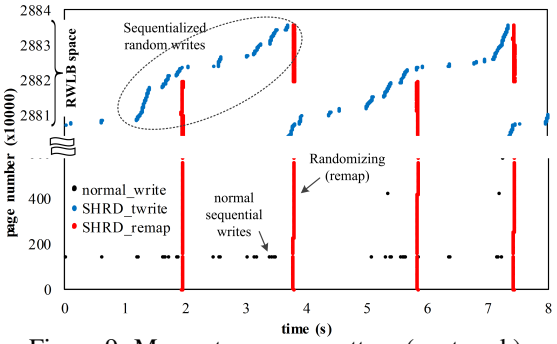
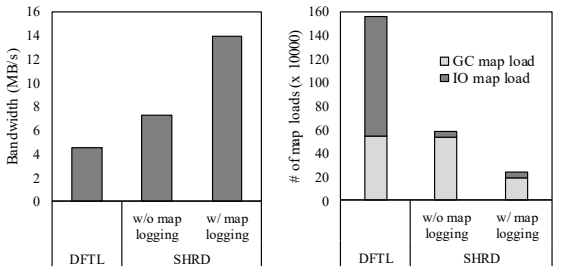


Figure 9: Map entry access pattern (postmark).



(a) performance improvement (b) map page miss reduction

Figure 10: Effect of map logging at GC.

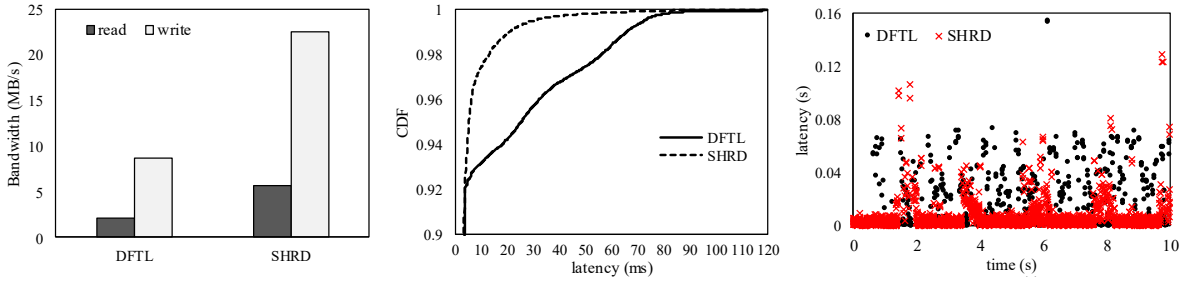
cess pattern during randomizing is sequential owing to the LPN-sorted accesses.

Map logging GC In order to examine the effect of map logging technique explained in Section 4.5, we initialized the SSD with an aging operation. First, 60% of the logical area of SSD was filled with sequential writes. Then, 80% of the sequentially-written data were overwritten with random writes. Finally, we ran the fio workload. The MLT size is 96 KB. Because the aging operation consumes all the physical blocks of SSD, the GC was triggered at the start of the fio test. As shown in Figure 10(a), the performance was further improved by using the map logging technique in addition to SHRD. As shown in Figure 10(b), SHRD can reduce only the number of map misses by normal IO requests. When the map logging technique is used additionally, the number of map misses by GC decreases.

SHRD overhead In the SHRD scheme, the redirection table must be searched for each read request. We measured the read performance degradation due to the redirection table searching overhead. First, we wrote two 4 GB files, one using 4 KB of random write requests and the other using 512 KB of sequential write requests, respectively. Then, the sequentially-written file was read with 4 KB of random read requests. The RWLB size is 64 MB; hence, the redirection table was filled with the mapping entries of the randomly-written file. Although, for each random read request to the sequentially-written file, its mapping entries cannot be found from the redirection table, the read requests must traverse the RB tree of the redirection table until the searching reaches the leaf nodes. In the worst-case scenario, we observed that the read performance degradation is less than 2%.

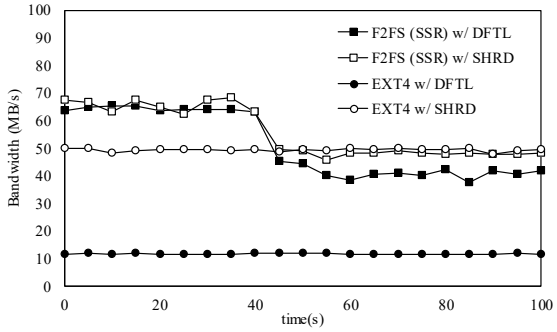
In the case of real workloads, read and write requests are mixed, where SHRD may show a worse read performance due to the sequentializing and randomizing operations on write requests. In particular, the remap command can delay the handling of read request, as explained in Section 4.4. Figure 11 compares the read performance of DFTL and SHRD. The fio mixed workload was used, which has four threads generating 4 KB of random read and write requests. The ratio of write requests is 80%; hence, it is a write-dominant workload. SHRD improved the write performance and thus the read performance was also improved by minimizing the waiting time for write request, as shown in Figures 11(a) and 12(b). SHRD showed long read latencies when SSD handles remap commands. However, the read latencies delayed by remap commands are similar to those delayed by map misses at DFTL, as shown in Figure 11(c).

Performance improvement at EXT4 and F2FS In order to compare the performance gains at different file systems, EXT4 and F2FS were used. F2FS is a log-structured file system, and it supports the slack space recycling (SSR) which permits overwrite operations for invalid blocks [22]. Compared to the garbage collection of LFS, the SSR operation can prevent significant performance degradation when the file system utilization is high. Figure 12(a) shows the performance improvement by SHRD for a random workload under the EXT4 and F2FS file systems. The file system utilizations were initialized to 75% by creating 1,024 files with sequential requests and updating 20% of the data with 4 KB of random write requests. Then, the fio random write workload was run. EXT4 showed a significant performance degradation for the random workload when DFTL was used. SHRD improved the performance of EXT4 significantly by reducing the map handling overhead. For F2FS, many free segments were generated by a background garbage collection before running the fio workload. Therefore, F2FS showed significantly better performance than the original EXT4 file system until 40 seconds elapsed because F2FS generated sequential write requests to the free segments. However, the free segments were exhausted and SSR operations were triggered starting from 40 seconds. The SSR operations generated small random write requests to utilize invalid blocks; thus, the performance of F2FS plummeted. However, by adopting the SHRD scheme, the performance of F2FS was improved even when the SSR operation was triggered. Consequently, EXT4 and F2FS showed similar performance when they adopted the SHRD scheme. As shown in Figure 12(b), F2FS showed worse sequential read/write performance than EXT4 at an aged condition. Although EXT4 showed worse random write performance than

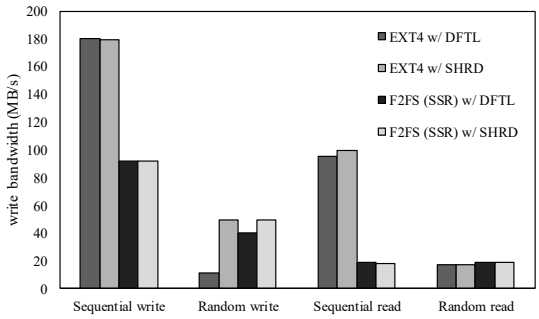


(a) bandwidth (b) CDF of read latency (c) read latency fluctuation

Figure 11: Read performance.



(a) performance change over time



(b) performance comparison at different workloads

Figure 12: Effect of SHRD at EXT4 and F2FS.

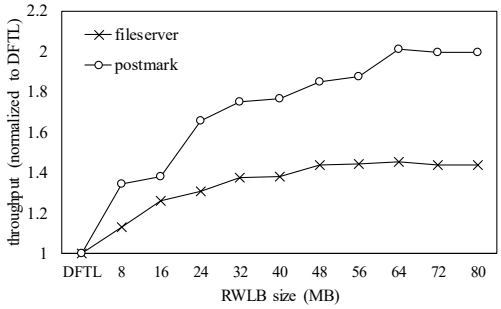


Figure 13: Effect of RWLB sizes.

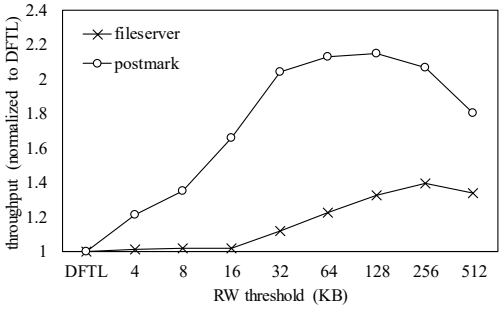


Figure 14: Effect of RW threshold.

F2FS under the DFTL scheme, SHRD removed the random write performance gap between EXT4 and F2FS. Therefore, the combination of EXT4 and SHRD can provide better performance for all types of workloads.

RWLB size and RW threshold When the SHRD scheme is implemented, several factors must be determined by considering the tradeoffs. As a larger size of RWLB is used, more number of remapping entries can share each map page, thus improving the spatial locality on accessing the map pages. In addition, the overwrite operations can invalidate more number of sequentialized pages in RWLB before they are randomized. However, a large RWLB requires a large redirection table and a large amount of table searching overhead. Figure 13 shows the performance changes for various sizes of the RWLB. A large RWLB provides better performance; however, the performance reaches a saturation point. Therefore, the RWLB size must be selected considering the drawbacks of a large RWLB.

The RW threshold determines the amount of data to be sequentialized by SHRD. As we increase the threshold, the performance can be improved by reducing the map handling overhead in the SSD as shown in Figure 14. However, a too large threshold can degrade performance by increasing the overhead of SHRD operations.

6 Related Work

Several studies have investigated approaches to handle the performance gap between sequential writes and random writes at flash storage. LFSs can eliminate random writes at the file system layer; SFS [29] and F2FS [22] are examples of such LFSs. SFS separates hot and cold data into different segments to reduce the cleaning overhead of a traditional LFS. F2FS arranges the on-disk layout from the perspective of the FTL on the SSDs and adopts adaptive logging to limit the maximum latency of segment cleaning. However, despite all the efforts, these

flash-based LFSs continue to suffer from write amplification in the segment cleaning phase. Further, LFSs show poor read performance as shown in Figure 12(b).

DFS [17], open-channel SSD [5], and application-managed flash [23] elevate the flash storage software into the OS layer and directly manage flash block allocation according to the flash address space. Therefore, the address mapping table is managed by host computer, and SSD will receive only sequential write requests. However, they can be used only for a specific SSD design and burdens the OS with excessive flash management overhead such as wear-leveling and GC.

Nameless Write [36] permits the storage device to choose the location of a write and inform the OS about the chosen address. Therefore, Nameless Write could eliminate the need for address indirection in SSDs. However, this scheme requires burdensome callback functions to communicate the chosen address to the host OS and necessitates significant changes to the conventional storage interface.

ReSSD [26] and LSDM [37] log random writes sequentially in a pre-reserved storage area and maintain the redirection map table in host memory. However, similar to the previous LFSs, these schemes must copy the data when the log space is reclaimed, thus causing write amplification. Further, they do not consider the POR issue; therefore, when a sudden power-off occurs, the logged data can be lost because the host memory is volatile.

The NVMe standard has a new interface called host memory buffer (HMB) [11], which permits NVMe SSD to utilize the DRAM of the host system via PCIe interface; thus, the vendor can build DRAM-less SSDs by maintaining the entire mapping table in the host DRAM. However, the latency of the host DRAM will be greater than the latency of the internal DRAM for SSD controller. In addition, the volatile mapping table must be flushed periodically to the SSD. On the contrary, SHRD minimizes the flushing overhead of the mapping table and requires only a small size of host memory.

Meanwhile, several studies adopt the FTL-level remap concept, in a manner similar to SHRD. JFTL [12] remaps the addresses of journal data to the addresses of home locations, thus eliminating redundant writes to flash storage. X-FTL [18] supports transactional flash storage for databases by leveraging the address mapping scheme of FTL. ANViL [34] proposes a storage virtualization interface based on FTL-level address remapping by permitting the host system to manipulate the address map using three operations-clone, move, and delete. SHARE [30] also utilizes the address remapping to enable host-side database engines to achieve write atomicity without causing write amplification. Ji *et al.* [15] proposed to use the remap operation for file system defragmentation.

Although the concept of address remapping was introduced by the mentioned studies, it is not trivial to implement the remap operation. SSD maintains two directions of address mappings, i.e., L2P mapping and its reverse P2L mapping. The P2L mapping is used by GC to identify the LPN of a physical page. The remap operation must change both the mappings, and the changed mapping information must be stored in flash memory blocks in order to ensure the durability. The P2L map of each physical page is generally stored in the OOB area of flash page, and thus it cannot be modified without copying the remapped physical page into another flash page. Otherwise, SSD must maintain a separate P2L mapping table. This problem is not easy, and any solution can involve P2L map handling overhead exceeding the benefits of remap operation. In the case of SHRD, we require a "restore" operation to the predetermined address rather than a remap operation to any address because we know the original logical address of a sequentialized page. We can easily implement the restore operation by storing the original logical address in the OOB area at sequentializing without modifying the P2L mapping.

7 Conclusion

We proposed a novel address reshaping technique, SHRD, in order to reduce the performance gap between random writes and sequential writes for SSDs with DRAM resource constraints. The sequentializer of the SHRD technique transforms random write requests into sequential write requests in the block device driver by assigning the address space of a reserved log area in the SSD. Read requests can access the sequentialized data by using a redirection table in the host DRAM. Unlike the previous techniques, SHRD can restore the original logical addresses of the sequentialized data without requiring copy operations, by utilizing the address indirection characteristic of the FTL. We also resolved the POR issue of the redirection table on the host DRAM. We developed a prototype of an SHRD-supporting SSD and a Linux kernel device driver to verify the actual performance benefit from SHRD, and demonstrated the remarkable performance gain. SHRD will be an effective solution for DRAM size reduction in large-capacity enterprise-class SSDs.

Acknowledgements

We thank the anonymous reviewers and our shepherd Sungjin Lee for their valuable comments. We also thank Dong-Gi Lee and Jaeheon Jeong in Samsung Electronics for supporting our work on Samsung SSD device. This research was supported by Samsung Electronics and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2016R1A2B2008672).

References

[1] Embedded Multi-media card (eMMC), Electrical standard (4.5 Device). http://www.jedec.org/standards-documents/results/jesd84-b45.

[2] Filebench. http://filebench.sourceforge.net/.

[3] Flexible I/O Tester. https://github.com/axboe/fio.

[4] MSR Cambridge Traces. http://iotta.snia.org/traces/388.

[5] Open-Channel Solid State Drives. http://lightnvm.io/.

[6] tpcc-mysql. https://github.com/Percona-Lab/tpcc-mysql.

[7] LevelDB. http://leveldb.org, 2016.

[8] RethinkDB: The open-source database for the realtime web. https://www.rethinkdb.com/, 2016.

[9] RocksDB: A persistent key-value store for fast storage environments. http://rocksdb.org, 2016.

[10] CHANG, L.-P., AND KUO, T.-W. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proc. of 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '02, pp. 187–196.

[11] CHEN, M. C. Host Memory Buffer (HMB) based SSD System. http://www.flashmemorysummit.com/, 2015.

[12] 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.

[13] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proc. of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pp. 143–154.

[14] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proc. of the 14th international Conference on Architectural Support For Programming Languages and Operating Systems*, ASPLOS '09, pp. 229–240.

[15] JI, C., CHANG, L., SHI, L., WU, C., AND LI, Q. An empirical study of file-system fragmentation in mobile storage systems. In *Proc. of 8th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '16.

[16] 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 *Proc. of IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST '11, pp. 23–27.

[17] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. DFS: A file system for virtualized flash storage. In *Proc. of the 8th USENIX Conference on File and Storage Technologies*, FAST '10.

[18] KANG, W.-H., LEE, S.-W., MOON, B., OH, G.-H., AND MIN, C. X-FTL: Transactional FTL for sqlite databases. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pp. 97–108.

[19] KATCHER, J. Postmark: A new file system benchmark. Tech. rep., TR3022, Network Appliance, 1997.

[20] KIM, J., KIM, J. M., NOH, S. H., MIN, S. L., AND CHO, Y. A space-efficient flash translation layer for compact flash systems. *IEEE Transactions on Consumer Electronics 48*, 2 (2002), 366–375.

[21] 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.

[22] LEE, C., SIM, D., HWANG, J.-Y., AND CHO, S. F2FS: A new file system for flash storage. In *Proc. of the 13th USENIX Conference on File and Storage Technologies*, FAST '15.

[23] LEE, S., LIU, M., JUN, S., XU, S., KIM, J., AND ARVIND. Application-managed flash. In *Proc. of 14th USENIX Conference on File and Storage Technologies*, FAST '16, pp. 339–353.

[24] LEE, S., SHIN, D., KIM, Y.-J., AND KIM, J. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review 42*, 6 (2008), 36–42.

[25] LEE, S.-W., PARK, D.-J., CHUNG, T.-S., LEE, D.-H., PARK, S., AND SONG, H.-J. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems 6*, 3 (2007).

[26] LEE, Y., KIM, J.-S., AND MAENG, S. ReSSD: A software layer for resuscitating SSDs from poor small random write performance. In *Proc. of the ACM Symposium on Applied Computing*, SAC '10, pp. 242–243.

[27] LEE, Y.-G., JUNG, D., KANG, D., AND KIM, J.-S. μFTL: A memory-efficient flash translation layer supporting multiple mapping granularities. In *Proc. of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, pp. 21–30.

[28] LI, Q., SHI, L., XUE, C. J., WU, K., JI, C., ZHUGE, Q., AND SHA, E. H.-M. Access characteristic guided read and write cost regulation for performance improvement on flash memory. In *Proc. of 14th USENIX Conference on File and Storage Technologies*, FAST '16, pp. 125–132.

[29] MIN, C., KIM, K., CHO, H., LEE, S.-W., AND EOM, Y. I. SFS: Random write considered harmful in solid state drives. In *Proc. of the 10th USENIX Conference on File and Storage Technologies*, FAST '12.

[30] OH, G., SEO, C., MAYURAM, R., KEE, Y.-S., AND LEE, S.-W. SHARE interface in flash storage for relational and NoSQL databases. In *Proc. of the 2016 International Conference on Management of Data*, SIGMOD '16, pp. 343–354.

[31] PARK, D., DEBNATH, B., AND DU, D. CFTL: A convertible flash translation layer adaptive to data access patterns. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '10, pp. 365–366.

[32] PARK, K.-T., ET AL. Three-dimensional 128 Gb MLC vertical NAND flash memory with 24-WL stacked layers and 50 MB/s high-speed programming. *IEEE JOURNAL OF SOLID-STATE CIRCUITS 50*, 1 (2015), 204–213.

[33] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage 9*, 3 (2013), 9:1–9:32.

[34] WEISS, Z., SUBRAMANIAN, S., SUNDARARAMAN, S., TALAGALA, N., ARPACI-DUSSEAU, S. A. C., AND ARPACI-DUSSEAU, R. H. ANViL: Advanced virtualization for modern non-volatile memory devices. In *Proc. of the 13th USENIX Conference on File and Storage Technologies*, FAST '15.

[35] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., AND SUNDARARAMAN, S. Don't stack your log on my log. In *Proc. of 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, INFLOW '14.

[36] ZHANG, Y., ARULRAJ, L. P., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. De-indirection for flash-based SSDs with nameless writes. In *Proc. of the 10th USENIX Conference on File and Storage Technologies*, FAST '12.

[37] ZUCK, A., KISHON, O., AND TOLEDO, S. LSDM: Improving the performance of mobile storage with a log-structured address remapping device driver. In *Proc. of 8th International Conference on Next Generation Mobile Applications, Services and Technologies* (2014), pp. 221–228.

Graphene: Fine-Grained IO Management for Graph Computing

Hang Liu H. Howie Huang
The George Washington University
{asherliu, howie}@gwu.edu

Abstract

As graphs continue to grow, external memory graph processing systems serve as a promising alternative to in-memory solutions for low cost and high scalability. Unfortunately, not only does this approach require considerable efforts in programming and IO management, but its performance also lags behind, in some cases by an order of magnitude. In this work, we strive to achieve an ambitious goal of achieving ease of programming and high IO performance (as in-memory processing) while maintaining graph data on disks (as external memory processing). To this end, we have designed and developed *Graphene* that consists of four new techniques: an IO request centric programming model, bitmap based asynchronous IO, direct hugepage support, and data and workload balancing. The evaluation shows that Graphene can not only run several times faster than several external-memory processing systems, but also performs comparably with in-memory processing on large graphs.

1 Introduction

Graphs are powerful data structures that have been used broadly to represent the relationships among various entities (e.g., people, computers, and neurons). Analyzing massive graph data and extracting valuable information is of paramount value in social, biological, healthcare, information and cyber-physical systems [14, 15, 17, 24, 29].

Generally speaking, graph algorithms include reading the *graph data* that consists of a list of neighbors or edges, performing calculations on vertices and edges, and updating the *graph (algorithmic) metadata* that represents the states of vertices and/or edges during graph processing. For example, breadth-first search (BFS) needs to access the adjacency lists (data) of the vertices that have just been visited at the prior level, and mark the statuses (metadata) of previously unvisited neighbors as visited. Accesses of graph data and metadata come hand-in-hand in many algorithms, that is, reading one vertex or edge will be accompanied with access to the corresponding metadata. It is important to note that in this paper we use the term *metadata* to refer to the key data structures in graph computing (e.g., the statuses in BFS and the ranks in PageRank).

To tackle the IO challenge in graph analytics, prior research utilizes in-memory processing that stores the whole graph data and metadata in DRAM to shorten the latency of random accesses [20, 35, 40, 44, 47]. In-memory processing brings a number of benefits including easy programming and high-performance IOs. However, this approach is costly and difficult to scale, as big graphs continue to grow drastically in size. On the other hand, the alternative approach of external memory graph processing focuses on accelerating data access on storage devices. However, this approach suffers not only from complexity in programming and IO management but also slow IO and overall system performance [40, 62].

To close the gap between in-memory and external memory graph processing, we design and develop *Graphene*, a new semi-external memory processing system that efficiently reads the graph data on SSDs while managing the metadata in DRAM. Simply put, Graphene incorporates graph data awareness in IO management behind an IO centric programming model, and performs fine-grained IOs on flash-based storage devices. This is different from current practice of issuing large IOs and relying on operating system (OS) for optimization [40, 47, 62]. Figure 1 presents the system architecture. The main contributions of Graphene are four-fold:

IO (request) centric graph processing. Graphene advocates a new paradigm where each step of graph processing works on the data returned from an IO request. This approach is unique from four types of existing graph processing systems: (1) vertex-centric programming model, e.g., Pregel [36], GraphLab [35], PowerGraph [20], and Ligra [47]; (2) edge-centric, e.g., X-stream [44] and Chaos [43]; (3) embedding-centric, e.g., Arabesque [50]; and (4) domain-specific language, e.g.,

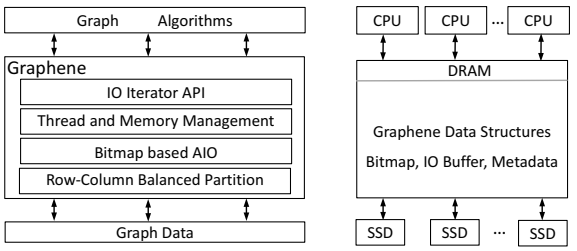


Figure 1: Architecture overview.

Galois [40], Green-Marl [27] and Trinity [46]. All these models are designed to address the complexity of the computation, including multi-threaded processing [27, 40], workload balancing [10, 20], inter-thread (node) communication [38] and synchronization [36]. However, in order to achieve good IO performance, these models require a user to explicitly manage the IOs, which is a challenging job by itself. For example, FlashGraph needs user input to sort, merge, submit and poll IO requests [62].

In Graphene, IO request centric processing (or IO centric for short) aims to simplify not only graph programming but also the task of IO management. To this end, we design a new *IoIterator* API that consists of a number of system and user-defined functions. As a result, various graph algorithms can be written in about 200 lines of code. Behind the scenes, Graphene translates high-level data accesses to fine-grained IO requests for better optimization. In short, IO centric processing is able to retain the benefit of easy programming while delivering high-performance IO.

Bitmap based, asynchronous IO. Prior research aims to read a large amount of graph data as quickly as possible, even when only a portion of it is needed. This design is justified because small random accesses in graph algorithms are not the strong suit of rotational hard drives. Notable examples include GraphChi [32] and X-stream [44], which read the entire graph data sequentially from the beginning to the end during each iteration of the graph calculation. In this case, the pursuit of high IO bandwidth overshadows the usefulness of data accesses. Besides this full IO model, the IO on-demand approach loads only the required data in memory, but again requires significant programming effort [25, 56, 62].

With the help of IO centric processing, Graphene pushes the envelope of the IO on-demand approach. Specifically, Graphene views graph data files as an array of 512-byte blocks, a finer granularity than more commonly used 4KB, and uses a *Bitmap*-based approach to quickly reorder, deduplicate, and merge the requests. While it incurs 3.4% overhead, the Bitmap approach improves the *IO utility* by as much as 50%, and as a result runs more than four times faster than a typical list based IO. In this work, IO utility is defined as the ratio between the amount of data that is loaded and useful for graph computation, and that of all the data loaded from disk. Furthermore, Graphene exploits *Asynchronous IO* (AIO) to submit as many IO requests as possible to saturate the IO bandwidth of flash devices.

Direct hugepage support. Instead of using 4KB memory pages, Graphene leverages the support of *Direct HugePage* (DHP), which preallocates the (2MB and 1GB) hugepages at boot time and uses them for both graph data and metadata structures, e.g., IO buffer and Bitmap. For example, Graphene designs a hugepage based memory buffer which enables multiple IO requests to share one hugepage. This technique eliminates the runtime uncertainty and high overhead in the transparent hugepage (THP) method [39], and significantly lowers the TLB miss ratio by 177×, leading to, on average, 12% performance improvement across different algorithms and graph datasets.

Balanced data and workload partition. Compared to existing 2D partitioning methods which divide vertices into equal ranges, Graphene introduces a row-column balanced 2D partitioning where each partition contains an equal number of edges. This ensures that each SSD holds a balanced data partition, especially in the cases of highly skewed degree distribution in real-world graphs. However, a balanced data partition does not guarantee that the workload from graph processing is balanced. In fact, the computation performed on each partition can vary drastically depending on the specific algorithm. To address this problem, Graphene utilizes dedicated IO and computing threads per SSD and applies a work stealing technique to mitigate the imbalance within the system.

We have implemented Graphene with different graph algorithms and evaluated its performance on a number of real world and synthetic graphs on up to 16 SSDs. Our experiments show that Graphene outperforms several external memory graph systems by 4.3 to 20×. Furthermore, Graphene is able to achieve similar performance to in-memory processing with the exception of BFS.

This paper is organized as follows: Section 2 presents the IO centric programming model. Section 3 discusses bitmap-based, asynchronous IO and Section 4 presents data and workload balancing techniques, and Section 5 describes hugepage support. Section 6 describes a number of graph algorithms used in this work. Section 7 presents the experimental setup and results. Section 8 discusses the related work and Section 9 concludes.

2 IO Request Centric Graph Processing

Graphene allows the system to focus on the data, be it a vertex, edge or subgraph, returned from an IO request at a time. This new IO (request) centric processing aims to provide the illusion that all graph data resides in mem-

Table 1: IoIterator API

Type	Name	Return Value	Description
System provided	Iterator->Next()	io_block_t	Get the next in-memory data block
	Iterator->HasMore()	bool	Check if there are more vertices available from IO
	Iterator->Current()	vertex	Get the next available vertex *v*
	Iterator->GetNeighbors(vertex *v*)	vertex array	Get the neighbors for the vertex *v*
User defined	IsActive(vertex *v*)	bool	Check if the vertex *v* is active
	Compute(vertex *v*)		Perform algorithm specific computation

```
while true do
    foreach vertex v do
        if IsActive(v) then
            handle = IO_Submit(v);
            IO_Poll(handle);
            Compute(the neighbors of v);
        end
    end
    level++;
end
```

Algorithm 1: BFS with user-managed IO.

```
while true do
    block = IoIterator→Next();
    while block→HasMore() do
        vertex v = block→Current();
        if IsActive(v) then
            Compute(block→GetNeighbors(v));
        end
    end
    level++;
end
```

Algorithm 2: IoIterator-based BFS.

ory, and delivers high IO performance through applying various techniques behind the scenes which will be described in next three sections.

To this end, Graphene develops an *IoIterator* framework, where a user only needs to call a simple *Next()* function to retrieve the needed graph data for processing. This allows the programmers to focus on graph algorithms without worrying about the IO complexity in semi-external graph processing. At the same time, by taking care of graph IOs, the IoIterator framework allows Graphene to perform disk IOs more efficiently in the background and make them more cache friendly. It is worth noting that the IO centric model can be easily integrated with other graph processing paradigms including vertex or edge centric processing. For example, Graphene has a user-defined *Compute* function that works on vertices.

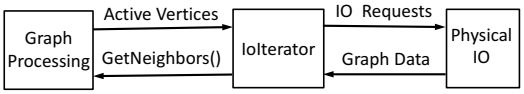


Figure 2: IoIterator programming model.

At a high level shown in Figure 2, we insert a new IoIterator layer between the algorithm and physical IO. In this architecture, the processing layer is responsible for the control flow, e.g., computing what vertices of the graph should be active, and working on the neighbors of those active vertices. The IO layer is responsible for serving the IO requests from storage devices. Graph processing can start as soon as the IOs for the adjacency lists of the active vertices are complete, i.e., when the data for the neighbors become available. The new abstraction of IoIterator is responsible for translating the requests for the adjacency lists into the IO requests for data blocks.

Internally, Graphene applied a number of IO optimizations behind the IoIterator, including utilizing a Bitmap per device for sorting and merging, submitting large amounts of non-blocking requests via asynchronous IO, using hugepages to store graph data and metadata, and resolving the mismatch between IO and processing across devices.

The IoIterator layer consists of a set of APIs listed in Table 1. There are four system-defined functions for the IoIterator, *Next*, *HasMore*, *Current*, and *GetNeighbors*, which work on the list of the vertices returned from the underlying IO layer. In addition, two functions *IsActive* and *Compute* should be defined by the users. For example, in BFS, the IsActive function should return *true* for any frontier if a vertex *v* has been visited in the preceding iteration, and Compute should check the status of each neighbor of *v*, and mark any unvisited neighbors as frontiers for the next iteration. Detailed description of BFS and other algorithms can be found in Section 6.

An example of BFS pseudocode written with the current approach of user-managed selective IO vs. the IoIterator API can be found in Algorithms 1 and 2. In the first approach, the users are required to be familiar with the Linux IO stack and explicitly manage the IO requests such as IO submission, polling, and exception handling. The main advantage of the IoIterator is that it completely removes such a need. On the other hand, in both approaches, the users need to provide two similar functions, *IsActive* and *Compute*.

It is important to note that the pseudocode will largely stay the same for other algorithms, but with different *IsActive* and *Compute*. For example, in PageRank, IsActive returns *true* for vertices that have delta updates, and Compute accumulates the updates from different source vertices to the same destination vertex. Here, Compute may be written in vertex or edge centric model.

3 Bitmap Based, Asynchronous IO

Graphene achieves high-performance IO for graph processing through a combination of techniques including fine-grained IO blocks, bitmap, and asynchronous IO. Specifically, Graphene favors small, 512-byte IO blocks to minimize the alignment cost and improve the IO utility, and utilizes a fast bitmap-based method to reorder and produce larger IO requests, which will be submitted to devices asynchronously. As a result, the performance of graph processing improves as a higher fraction of useful data are delivered to CPUs at high speed.

In Graphene, graph data are stored on SSDs in *Compressed Sparse Row* (CSR) format which consists of two data structures: the *adjacency list array* that stores the IDs of the destination vertices of all the edges ordered by the IDs of the source vertices, and the *beginning position array* that maintains the index of the first edge for each vertex.

3.1 Block Size

One trend in modern operating systems is to issue IOs in larger sizes, e.g., 4KB by default in some Linux distributions [8]. While this approach is used to achieve high sequential bandwidth from underlying storage devices like hard drives, doing so as in prior work [62] would lead to low IO utility because graph algorithms inherently issue small data requests. In this work, we have studied the IO request size when running graph algorithms on Twitter [2] and Friendster [1]. Various graph datasets that are used in this paper is summarized in Section 7. One can see that most (99%) of IO requests are much smaller than 4KB as shown in Figure 3. Thus, issuing 4KB IOs would waste a significant amount of IO bandwidth.

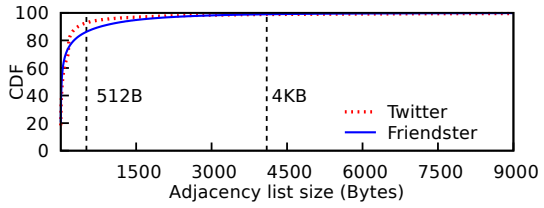


Figure 3: Distribution of IO sizes.

In Graphene, we choose to use a small IO size of 512 bytes as the basic block for graph data IOs. Fortunately, new SSDs are capable of delivering good IOPS for 512-byte read requests for both random and sequential IOs. For example, Samsung 850 SSD [49], which we use in the experiments, can achieve more than 20,000 IOPS for 512-byte random read.

Another benefit of using 512-byte blocks is to lower the cost of the alignment for multiple requests. Larger block size like 4KB means the offset and size of each IO request should be a multiple of 4KB. In the example presented in Figure 4, requesting the same amount of

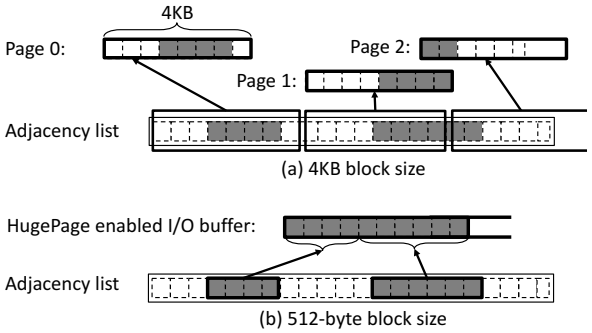


Figure 4: IO alignment cost: 4KB vs. 512-byte blocks, where one dotted box represents one 512-byte block.

data will lead to the different numbers of IOs when using 4KB (top) and 512-byte (bottom) block sizes. One can see that the former will load 2.2× more data, i.e., 12KB vs. 5KB in this case. In addition, combined with hugepage support that will be presented shortly, 512-byte block IO will need only one hugepage-based IO buffer, compared to three 4KB pages required in the top case.

3.2 Bitmap-Based IO Management

At each iteration of graph processing, graph algorithms compute and generate the requests for the adjacency lists (i.e., the neighboring vertices) of *all* active vertices for the following iteration. In particular, Graphene translates such requests into a number of 512-byte aligned IO blocks, which are quickly identified in a new *Bitmap* data structure. In other words, Graphene maintains a Bitmap per SSD, one bit for each 512-byte block on the disk. For each request, Graphene marks the bits for the corresponding blocks, that is, should a block need to be loaded, its bit is marked as "1", and "0" otherwise. Clearly, the Bitmap offers a *global* view of IO operations and enables optimization opportunities which would not otherwise be possible.

For a 500GB SSD as we have used in this work, the size of the bitmap is merely around 128MB, which we can easily cache in CPUs and store in DRAM with a number of hugegages. Because Graphene combines Bitmap-based management with asynchronous IO, it is also able to utilize one IO thread per SSD. Therefore, since there is only one thread managing the Bitmap for each SSD, no lock is required on the Bitmap structures.

Issues with local IO optimization. Traditionally, the OS takes a *local* view of the IO requests by immediately issuing the requests for the neighbors of one or a group of active vertices. In addition, the OS performs several important tasks such as IO batching, reordering and merging at the block layer. Unfortunately, these techniques have been applied only to IO requests that have been buffered in certain data structures. For instance, Linux exploits a linked list called *pluglist* to batch and submit the IO requests [8], in particular, the most recent Linux kernel 4.4.0 supports 16 requests in a batch.

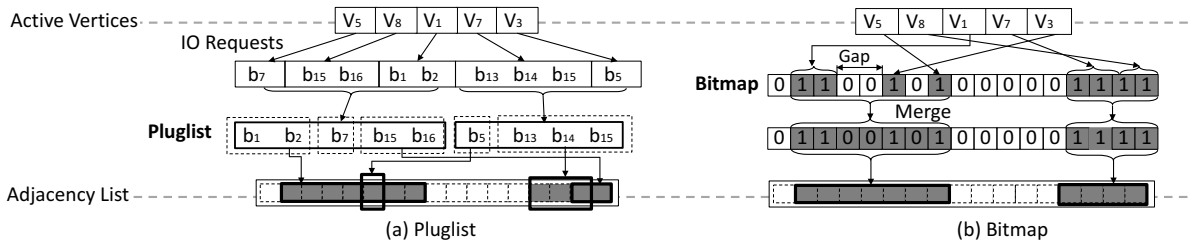


Figure 5: Pluglist vs. bitmap IO management, (a) Pluglist where sorting and merging are limited to IO requests in the pluglist. (b) Bitmap where sorting and merging are applied to all IO requests.

Figure 5(a) presents the limitations of the pluglist based approach. In this example, vertices $\{v_5, v_8, v_1, v_7, v_3\}$ are all active and the algorithm needs to load their neighbors from the adjacency list file. With a fixed-size pluglist, some of the requests will be *batched* and enqueued first, e.g., the requests for the first three vertices $\{v_5, v_8, v_1\}$. In the second step, *sorting* is applied across the IO requests in the pluglist. Since the requests are already grouped, sorting happens within the boundary of each group. In this case, the requests for the first three vertices are reordered from $\{b_7, b_{15}, b_{16}, b_1, b_2\}$ to $\{b_1, b_2, b_7, b_{15}, b_{16}\}$. In the third step, if some IO blocks present good spatial locality, *merging* will be applied to form a larger IO request, e.g., blocks $\{b_1, b_2, b_7\}$ are merged into one IO transaction. And later, a similar process happens for the IOs on the rest of vertices $\{v_7, v_3\}$.

In this case, there are four independent IO requests to the disk, (a) blocks b_1 - b_7, (b) blocks b_{15} - b_{16}, (c) block b_5, and (d) blocks b_{13} - b_{15}. The first request loads seven sequential blocks in one batch, which takes advantage of prefetching and caching and is preferred by the disks and OS. As a result, the third request for block b_5 will likely hit in the cache. On the other hand, although the second and fourth requests have overlapping blocks, they will be handled as two separate IO requests.

Bitmap and global IO optimization. Graphene chooses to carry out IO management optimizations, including IO deduplication, sorting and merging, on a *global* scale. This is motivated by the observation that although graph algorithms tend to present little or no locality in a short time period, there still exists a good amount of locality within the entire processing window. Bitmap-based IO management is shown in Figure 5(b). Upon receiving the requests for all active vertices, Graphene will convert the needed adjacency lists into the block addresses and mark those blocks in the Bitmap.

Sorting. The process of marking active blocks in the corresponding locations in the Bitmap naturally sorts the requests in the order of physical addresses on disks. In other words, the order of the requests is simply that of the marked bits in the Bitmap.

IO deduplication is also easily achieved in the process. Bitmap-based IO ensures that only one IO request will be sent even when the data block is requested multiple times, achieving the effect of IO deduplication. This is common in graph computation. For example, in the single source shortest path algorithm, one vertex may have many neighboring vertices, and if more than one neighbors need to update the distance of this vertex, it will need to be enqueued multiple times for the next iteration. In addition, different parts of the same IO block may need to be loaded at the same time. In the prior example, as the block b_{15} is shared by the requests from vertices v_7 and v_8, it will be marked and loaded once. Our study shows that the deduplication enabled from Bitmap can save up to 3× IO requests for BFS, compared to a pluglist based method.

IO merging. Bitmap is very easy to use for merging the requests in the vicinity of each other into a larger request, which reduces the total number of IO requests submitted to disks. For example, as shown in Figure 5(b), IO requests for vertices v_1, v_3, v_5 (and similarly for vertices v_7 and v_8) are merged into one. As a result, there are only two non-overlapping requests instead of four as in the pluglist case.

How to merge IO requests is guided by a number of rules. It is straightforward that consecutive requests should be merged. When there are multiple non-consecutive requests, we can merge them when the blocks to be loaded are within a pre-defined *maximum gap*, which determines the largest distance between two requests. Note that this rule directly evaluates the Bitmap by bytes to determine whether eight consecutive blocks are needed to be merged.

This approach favors larger IO sizes and has proven to be effective in achieving high IO performance. Figure 6 shows the performance when running BFS on the Twitter and UK graphs. Interestingly, the performance peaks for both graphs when the maximum gap is set to 16 blocks (i.e., 8KB). Graphene also imposes an upper bound for IO size, so that the benefit of IO merging would not be dwarfed by handling of large IO requests. We will discuss this upper bound shortly.

In conclusion, Bitmap provides a very efficient method to manage IO requests for graph processing. We will show later that while the OS already provides similar functionality, this approach is more beneficial for dealing with random IOs to a large amount of data.

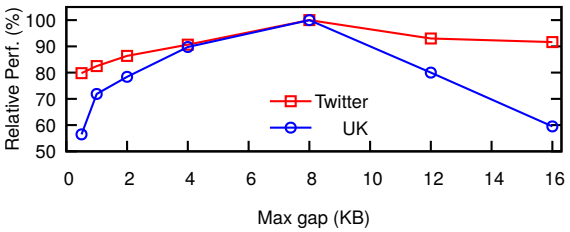
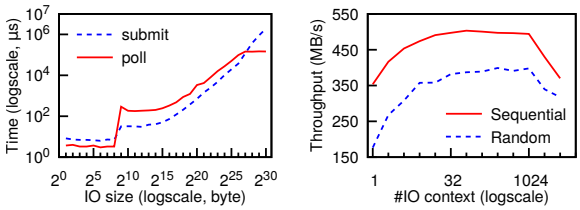


Figure 6: Graphene BFS Performance of maximum gap.



(a) IO size (b) IO context

Figure 7: AIO performance w.r.t. IO size and IO context

Besides Bitmap-based IO, we have also implemented a Pluglist based approach that extends the pluglist to support sorting, deduplication and merging in a global scale. As shown in Section 7, compared to a list, the Bitmap approach incurs smaller overhead and runs four times faster. It is important to note that although we focus on using Bitmap for graph processing in this work, it can also be applied to other applications. We will demonstrate this potential in Section 7.

3.3 Asynchronous IO

Asynchronous IO (AIO) is often used to enable a user-mode thread to read or write a file, while simultaneously carrying out the computation [8]. The initial design goal is to overlap the computation with non-blocking IO calls. However, because graph processing is IO bound, Graphene exploits AIO for a different goal of submitting as many IO requests as possible to saturate the IO bandwidth of flash devices.

There are two popular AIO implementations, i.e., user-level *POSIX AIO* and kernel-level *Linux AIO*. We prefer the latter in this work, because POSIX AIO forks child threads to submit and wait for the IO completion, which in turn has scalability issues while submitting too many IO requests [8]. In addition, Graphene leverages direct IO to avoid the OS-level page cache during AIO, and the possible blocks introduced by the kernel [19].

Upper bound for IO request. Although disks favor large IO sizes in tens or hundreds of MBs, it is not always advantageous to do so, especially for AIO. Typically, an AIO consists of two steps, submitting the IO request to an IO context and polling the context for completion. If IO request sizes are too big, the time for IO submission would take longer than polling, at which point AIO would essentially become blocking IO. Figure 7(a) studies the AIO submission and polling time. As the size goes beyond 1MB, submission time increases quickly. And once it reaches 128MB, it becomes blocked IO as submission time eventually becomes longer then polling time. In this work, we find that a modest IO size, such as 8, 16, and 32 KB, is able to deliver good performance for various graph algorithms. Therefore, we set the default upper bound of IO merging as 16KB.

IO context. In AIO, each IO context loads the IO requests sequentially. Graphene uses multiple contexts to handle the concurrent requests and overlap the IO with the computation. For example, while a thread is working on the request returned from one IO context, another IO context can be used to serve other requests from the same SSD. Given its intensive IO demand, graph computation would normally need to create a large number of IO contexts. However, without any constraints, too many IO contexts would hurt the performance because every context needs to register in the kernel and may lead to excessive overhead from polling and management.

Figure 7(b) evaluates the disk throughput with respect to the number of total IO contexts. As one can see that each SSD could achieve the peak performance with 16 contexts but the performance drops once the total IO context goes beyond 1,024 contexts. In this work, depending on the number of available SSDs, we utilize different numbers of IO contexts, by default using 512 contexts for 16 SSDs.

3.4 Conclusion

In summary, combining 512-byte block and Bitmap-based IO management allows Graphene to load a smaller amount of data from SSDs, about 21% less than the traditional approach. Together with AIO, Graphene is able to achieve high IO throughput of upto 5GB/s for different algorithms on an array of SSDs.

4 Balancing Data and Workload

Taking care of graph data IO only solves half of the problem. In this section, we present data partitioning and workload balancing in Graphene.

4.1 Row-Column Balanced 2D Partition

Given highly skewed degree distribution in power-law graphs, existing graph systems, such as GridGraph [63], TurboGraph [25], FlashGraph [62], and PowerGraph [20], typically apply a simple 2D partitioning method [9] to split the neighbors of each vertex across multiple partitions. The method is presented in Figure 8(a), where each partition accounts for an equal range of vertices, P number of vertices in this case, on both row and column-wise. This approach needs to scan the graph data once to generate the partitions. The main

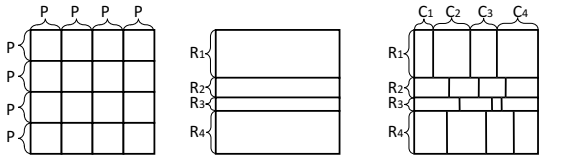


Figure 8: Graphene Balanced 2D partition.

drawback of this approach is that an equal range of vertices in each data partition do not necessarily lead to an equal amount of edges, which can result in workload imbalance for many systems.

To this end, Graphene introduces a row-column balanced 2D partitioning method, as shown in Figure 8(b-c), which ensures each partition contains an equal number of edges. In this case, each partition may have different numbers of rows and columns. This is achieved through three steps: (1) the graph is divided by the row major into R number of partitions, each of which has the same numbers of edges with potentially different number of rows; (2) Each row-wise partition is further divided by the column major into C number of (smaller) partitions, each of which again has the equal amount of edges. As a result, each partition may contain different number of rows and columns. Although it needs to read the graph one more time, it produces "perfect" partitions with the equal amount of graph data, which can be easily distributed to a number of SSDs.

Figure 9 presents the benefits of row-column balanced 2D partition for two social graphs, Twitter and Friendster. On average, the improvements are 2.7× and 50% on Twitter and Friendster, respectively. The maximum and minimum benefits for Twitter are achieved on SpMV for 5× and k-Core 12%. The speedups are similar for Friendster. While each SSD holds a balanced data partition, the workload from graph processing is not guaranteed to be balanced. Rather, the computation performed on each partition can vary drastically depending on the specific algorithm. In the following, we present the workflow of Graphene and how it balances the IO and processing.

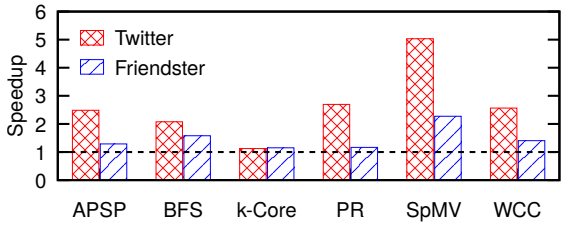


Figure 9: Benefit of row-column balanced 2D partition.

4.2 Balancing IO and Processing

Although AIO, to some extent, enables the overlapping between IO and computation, we have observed that a single thread doing both tasks would fail to fully saturate

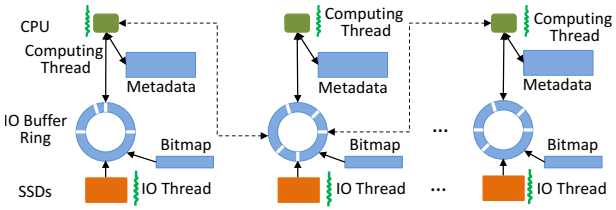


Figure 10: Graphene scheduling management.

the bandwidth of an SSD. To address this problem, one can assign multiple threads to work on a single SSD in parallel. However, if each thread would need to juggle IO and processing, this can lead to contention in the block layer, resulting in a lower performance.

In Graphene, we assign two threads to collaboratively handle the IO and computation on each SSD. Figure 10 presents an overview of the workflow. Initially upon receiving updates to the Bitmap, a dedicated *IO thread* formulates and submits IO requests to the SSD. Once the data is loaded in memory, the *computing thread* retrieves the data from the *IO buffer* and works on the corresponding metadata. Using PageRank as an example, for currently active vertices, the IO thread would load their in-neighbors (i.e., the vertices with a directed edge to active vertices) in the IO buffer, further store them in the ring buffer. Subsequently, the computing thread uses the rank values of those in-neighbors to update the ranks of active vertices. The metadata of interest here is the rank array.

Graphene pins IO and computing threads to the CPU socket that is close to the SSD they are working on. This NUMA-aware arrangement reduces the communication overhead between IO thread and SSD, as well as IO and computing threads. Our test shows that this can improve the performance by 5% for various graphs.

Graphene utilizes a work stealing technique to mitigate computational imbalance issue. As shown in Figure 10, each computing thread first works on the data in its own IO buffer ring. Once it finishes processing its own data, this thread will check the IO buffer of other computing threads. As long as other computing threads have unprocessed data in IO buffers, this thread is allowed to help process them. This procedure repeats until all data have been consumed.

Figure 11 presents the performance benefit from work stealing. On average, PageRank, SpMV, WCC and APSP

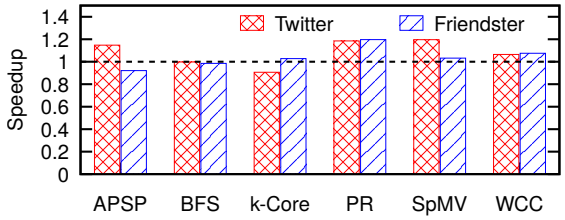


Figure 11: Benefit of workload stealing.

achieve various speedup of 20%, 11%, 8% and 4%, respectively, compared to the baseline of not using workload stealing. On the other hand, BFS and k-Core suffer slowdown of 1% and 3%. This is mostly because the first four applications are more computation intensive while BFS and k-Core are not. One drawback of workload stealing is lock contention at the IO buffer ring, which can potentially lead to performance degradation, e.g., 8% for APSP on Friendster and k-Core on Twitter.

5 HugePage Support

Graphene leverages the support of *Direct HugePages* (DHP), which preallocates hugepages at boot time, to store and manage graph data and metadata structures, e.g., IO buffer and Bitmap, shown as blue boxes in Figure 10. This is motivated by our observation of high TLB misses, as the number of memory pages continues to grow for large-scale graph processing. Because a TLB miss typically requires hundreds of CPU cycles for the OS to go through the page table to figure out the physical address of the page, this would greatly lower the graph algorithm performance.

In Graphene, the OS creates and maintains a pool of hugepages at machine boot time when memory fragmentation is at the minimum. This is because any memory fragmentation would break physical space into pieces and disrupt the allocation of hugepages. We choose this approach over transparent hugepage (THP) in Linux [39] for a couple of reasons. First, we find that THP introduces undesirable uncertainty at runtime, because such a hugepage could be swapped out from memory [42]. Second, THP does not always guarantee successful allocation and may incur high CPU overhead. For example, when there were a shortage, the OS would need to aggressively compress the memory in order to provide more hugepages [54].

Data IO. Clearly, if each IO request were to consume one hugepage, a large portion of memory space would be wasted, because Graphene, even with IO merging, rarely issues large (2MB) IO requests. Alternatively, Graphene allows multiple IO requests to share hugepages. This consolidation is done through IO buffers in the IO Ring Buffer. Given a batch of IO requests, Graphene first claims a buffer that contains a varied number of continuous 2MB hugepages. As the IO thread works exclusively with a buffer, all IO requests can in turn use any portion of it to store the data. Also, consecutive IO requests will use continuous memory space in the IO buffer so that there is no fragmentation. Note that the system needs to record the begin position and length of each request within the memory buffer, which is later parsed and shared with the user-defined Compute function in the IoIterator. In addition, *direct IO* is utilized for loading disk blocks directly into hugepages. Comparing to buffered IO, this method skips the step of copying data to system pagecache and further to user buffer, i.e., double copy.

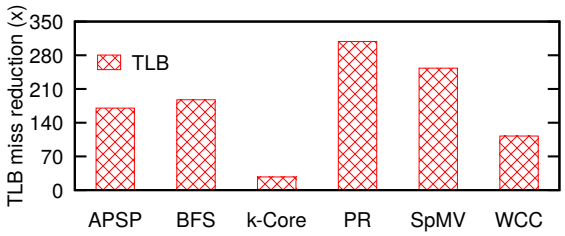


Figure 12: TLB misses reduced by hugepage-enabled buffer.

Metadata has been the focus of several prior works [9, 12, 59] to improve the cache performance of various graph algorithms. As a first attempt, we have investigated the use of page coloring [16, 60] to resolve cache contention, that is, to avoid multiple vertices being mapped to the same cache line. With 4KB pages, we are able to achieve around 5% improvement across various graphs. However, this approach becomes incompatible when we use 2MB hugepages for metadata, as the number of colors is determined by the LLC size (15MB), associativity (20) and page size.

To address this challenge, we decide to use hugepages for the metadata whose size is at the order of $O(|V|)$. In this work, we use 1GB hugepages, e.g., for PageRank, a graph with one billion vertices will need 4GB memory for metadata, that is, four 1GB hugepages.

This approach brings several benefits. Figure 12 illustrates the reduction in TLB miss introduced by this technique when running on a Kronecker graph. Across six algorithms, we observe an average 177× improvement with the maximum of 309× for PageRank. In addition, as prefetching is constrained by the page size, hugepages also enables more aggressive hardware prefetching in LLC, now that the pages are orders of magnitude bigger (1GB vs. 4KB). The test shows that this technique provides around 10% speedup for these graph algorithms.

6 Graph Algorithms

Graphene implements a variety of graph algorithms to understand different graph data and metadata, and their IO patterns. For all the algorithms, the sizes of data and metadata are $O(|E|)$ (total count of edges) and $O(|V|)$ (total count of vertices), respectively.

Breadth First Search (BFS) [4, 33] performs random reads of the graph data, determined by the set of most recently visited vertices in the preceding level. The statuses (visited or unvisited) of the vertices are maintained in the status array, a key metadata in BFS. It is worthy to note that status array may experience more random IOs, because the neighbors for a vertex tend to have different IDs, some of which are far apart.

PageRank (PR) [26,41] can calculate the popularity of a vertex by either pulling the updates from its in neighbors

or pushing its rank to out neighbors. The former performs random IO on the rank array (metadata), whereas the latter requires sequential IO for graph data but needs locks while updating the metadata. In this work, we adapt delta-step PageRank [61], where only vertices with updated ranks should push their delta values to the neighbors, yet again requiring random IOs.

Weakly Connected Component (WCC) is a special type of subgraph whose vertices are connected to each other. For directed graphs, a strongly connected component exists if a directed path can be found between all pairs of vertices in the subgraph [28]. In contrast, a WCC exists if such a path can be found regardless of the edge direction. We implement the hybrid WCC detection algorithm presented in [48], that is, it uses BFS to detect the largest WCC then uses label propagation to compute remaining smaller WCCs. In this algorithm, the label array serves as the metadata.

k-Core (KC) [37, 45] is another type of subgraph where each vertex has the degree of at least k. Iteratively, a k-Core subgraph is found by removing the vertices from the graph whose degree is less than k. As the vertices are removed, their neighbors are affected, where the metadata – degree array – will need to be updated. Similar to aforementioned algorithms, since the degree array is indexed by the vertex IDs, the metadata IO in k-Core also tends to be random. k-Core is chosen in this work as it presents alternating graph data IO patterns across different iterations. Specifically, in the initial iterations, lots of vertices would be affected when a vertex is removed, thus the graph data is retrieved likely in the sequential order. However at the later iterations, fewer vertices will be affected, resulting in random graph data access.

All Pairs Shortest Path (APSP) calculates the shortest paths from all the vertices in the graph. With APSP, one can further compute Closeness Centrality and Reachability problems. Graphene combines multi-source traversals together, to reduce the total number of IOs needed during processing and the randomness exposed during the metadata access [34, 51]. Similar to FlashGraph, we randomly select 32 source vertices for evaluation to reduce APSP execution time on large graphs.

Sparse Matrix Vector (SpMV) multiplication exhibits sequential access when loading the matrix data, and random access for the vector. In this algorithm, the matrix and vector serve the role as graph data and metadata, respectively. As a comparison to BFS, SpMV is more IO friendly but equally challenging on cache efficiency.

7 Evaluations

We have implemented a prototype of Graphene in 3,300 lines of C++ code, where the IoIterator accounts for 1,300 lines and IO functions 800 lines. Six graph algorithms are implemented with average 200 lines of code. We perform our experiments on a server with a dual-socket Intel Xeon E5-2620 processor (total 12 cores and 24 threads with hyperthreading), 128GB memory, 16 500GB Samsung 850 SSDs connected with two LSI SAS 9300-8i host bus adapters, and Linux kernel 4.4.0.

Table 2: Graph Datasets.

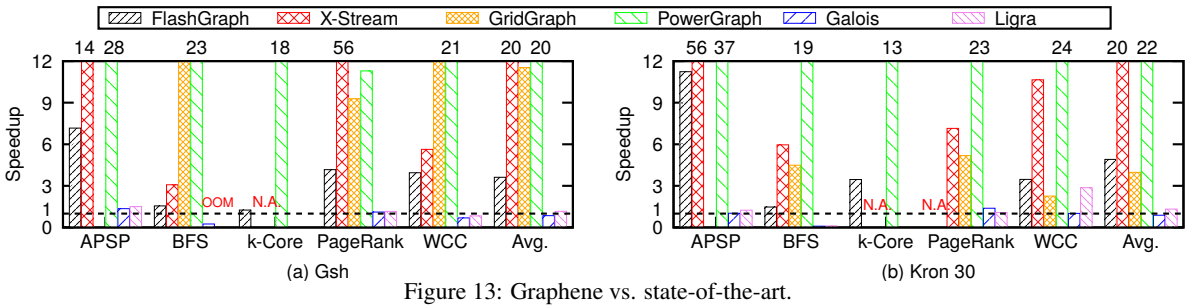
Name	# Vertices	# Edges	Size	Preprocess (seconds)
Clueweb	978M	42.6B	336GB	334
EU	1071M	92B	683GB	691
Friendster	68M	2.6B	20GB	3
Gsh	988M	33.8B	252GB	146
Twitter	53M	2.0B	15GB	2
UK	788M	48B	270GB	240
Kron30	1B	32B	256GB	141
Kron31	2B	1T	8TB	916

Table 2 lists all the graphs used in this paper. Specifically, Twitter [2] and Friendster [1] are real-world social graphs. In particular, Twitter contains 52,579,682 vertices and 1,963,263,821 edges, and Friendster is an online gaming network with 68,349,466 vertices and 2,586,147,869 edges. In addition, Clueweb [13], EU [18], Gsh [23] and UK [55] are webpage based graphs provided by webgraph [5–7]. Among them, EU is the largest with over one billion of vertices and 90 billion of edges. On the other hand, two Kronecker graphs are generated with the Graph500 generator [22] with scale 30 and 31, which represent the number of vertices as 1 billion (2^{30}) and 2 billion (2^{31}), with number of edges of 32 billion and 1 trillion. This paper, by default uses 8 bytes to represent a vertex ID unless explicitly noted. We run the tests five times and report the average values.

In addition, Table 2 presents the time consumption of the preprocessing step of the row-column balanced 2D partition. On average, our partition method takes 50% longer time than the conventional 2D partition method, e.g., preprocessing the largest Kron31 graph takes 916 seconds. Note that except X-Stream, many graph systems, including FlashGraph, GridGraph, PowerGraph, Galois and Ligra, also require similar or longer preprocessing to prepare the datasets. In the following, we report the runtime of graph algorithms, excluding the preprocessing time for all graph systems.

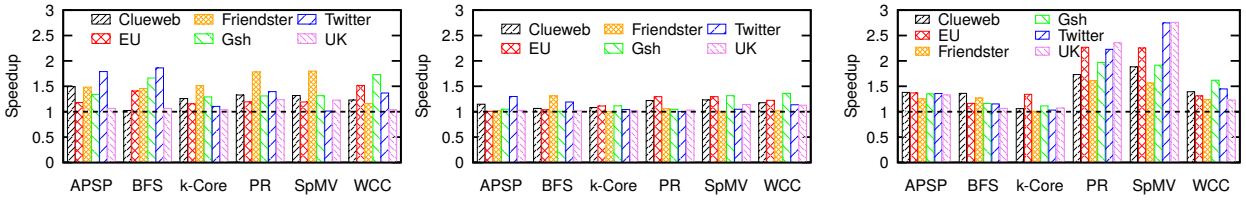
7.1 Comparison with the State of the Art

We compare Graphene against FlashGraph (semi-external memory), X-Stream (external memory), GridGraph (external memory), PowerGraph (in-memory), Galois (in-memory), and Ligra (in-memory) when running various algorithms. Figure 13 reports the speedup of Graphene over different systems for all five algorithms. SpMV is currently not supported in other systems except our Graphene, and k-Core is only provided by FlashGraph, PowerGraph and Graphene. In the figure the label "NA" indicates lack of support in the system. In this test, we choose one real graph (Gsh) and one synthetic graph



(a) Gsh (b) Kron 30

Figure 13: Graphene vs. state-of-the-art.



(a) Bitmap vs. Pluglist (b) HugePage vs. 4K Page (c) Dedicated IO and computing threads

Figure 14: Overall performance benefits of IO techniques.

(Kron30). Note that Gsh is the largest graph that is supported by in-memory systems. We have observed similar performance on other graphs.

In general, Graphene outperforms external memory systems FlashGraph, GridGraph and X-Stream by 4.3×, 7.8× and 20×, respectively. Compared to in-memory systems PowerGraph, Galois and Ligra where all graph data are stored in DRAM, Graphene keeps the data on SSDs and reads on-demand, outperforming PowerGraph by 21× and achieving a comparable performance with the other two (90% for Galois and 1.1× for Ligra). Excluding BFS which is the most IO intensive and favors in-memory data, Graphene outperforms Galois and Ligra by 10% and 45%, respectively. We also compare Graphene with an emerging Differential Dataflow system [53] and Graphene is able to deliver an order of magnitude speedup on BFS, PageRank and WCC.

For the Gsh graph, as shown in Figure 13, Graphene achieves better performance than other graph systems for different algorithms with exceptions for BFS and WCC. For example, for APSP, Graphene outperforms PowerGraph by 29×, Galois by 35%, Ligra by 50%, FlashGraph by 7.2× and X-Stream by 14×. For BFS and WCC, Graphene runs faster than GridGraph, PowerGraph, FlashGraph and X-Stream, but is slower than the two in-memory systems, mostly due to relatively long access latency on SSDs compared to DRAM. Similar performance benefits can also be observed on the syntheic Kron30 graph.

Table 3: Graphene runtime on Kron31 (seconds).

Name	APSP	BFS	k-Core	PageRank	WCC	SpMV
Kron31	7,233	2,630	318	25,023	3,023	5,706

Trillion-edge graph. We further evaluate the performance of Graphene on Kron31 as presented in Table 3. On average, all algorithms take around one hour to finish, with the maximum from PageRank of 6.9 hours while k-Core can be completed in 5.3 minutes. To the best of our knowledge, this is among the first attempts to evaluate trillion-edge graphs on a external-memory graph processing system.

7.2 Benefits of IO Techniques

This section examines the impacts on the overall system performance brought by different techniques independently, including Bitmap, hugepage, and dedicated IO and computing threads. We run all six algorithms on all six real-world graphs.

The Bitmap provides an average 27% improvement over using the pluglist as presented in Figure 14(a). Clearly, Bitmap favors the algorithms with massive random IOs such as WCC and BFS and low diameter graphs such as Gsh, EU, and Friendster. For example, Bitmap achieves about 70% speedup on Gsh on both BFS and WCC, and 30% for other algorithms.

Figure 14(b) compares the performance of hugepages and 4KB pages. Hugepages provides average 12% improvement and the speedup varies from 17% for WCC to 6% for k-Core. Again, two largest improvements are achieved on the (largest) Gsh graph for SpMV and WCC.

The benefit introduced by dedicated IO and computing threads is presented in Figure 14(c), where the baseline is using one thread for both IO and computing. In this case, Graphene achieves an average speedup of 54%. Particularly, PageRank and SpMV enjoy significant higher improvement (about 2×) than the other algorithms.

7.3 Analysis of Bitmap-based IO

We study how Bitmap-based IO affects the IO and computing ratio of different algorithms in Figure 15. Without

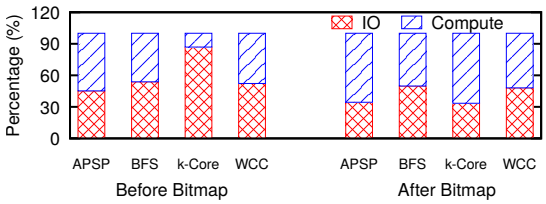
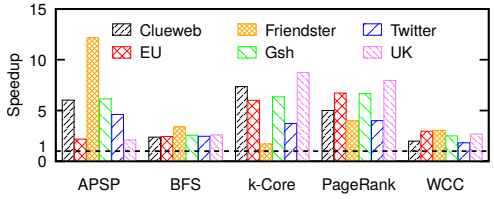


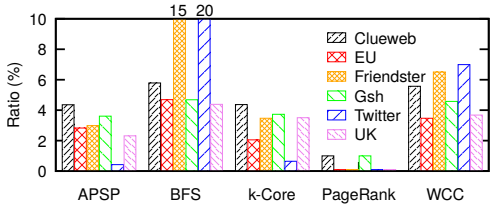
Figure 15: Runtime breakdown of IO and computing with Bitmap-based IO.

bitmap, all four algorithms spend about 60% on IO and 40% on computation. In comparison, the distribution of runtime reverses with bitmap, where computation takes average 60% of the time and IO 40%. Because the IO time is significantly reduced, faster IO as a result accelerates the execution of the algorithms. In particular, the biggest change comes from k-Core where IO accounts for 87% and 34% before and after bitmap.

As shown in Figure 16, when compared to a pluglist-based approach, the Bitmap-based IO runs 5.5×, 2.6×, 5.6×, 5.7× and 2.5× faster on APSP, BFS, k-Core, PageRank, and WCC, respectively. Note that here we only evaluate the time consumption of preparing the bitmap and pluglist, which is different from overall system performance presented in Figure 14. On the other hand, in most cases, adding Bitmap incurs a small increase of about 3.4% of total IO time. However, for a few cases with relatively high overhead, it is most likely caused by the small size of the graph data (e.g., Friendster and Twitter), as well as random IOs of the algorithms (e.g., BFS). The time spent on Bitmap varies from about 60 milliseconds for PR and SpMV (less than 1% of total IO time), to 100 seconds for APSP (2.3% of IO time).



(a) Preparing Bitmap vs. Pluglist.



(b) Overhead.

Figure 16: Bitmap performance and overhead.

Bitmap-based IO can be applied to other applications beyond graph processing. Figure 17 examines the time consumption differences between Bitmap based IO and Linux IO. Here we replay the reads in five IO traces

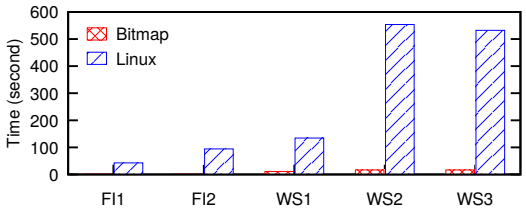
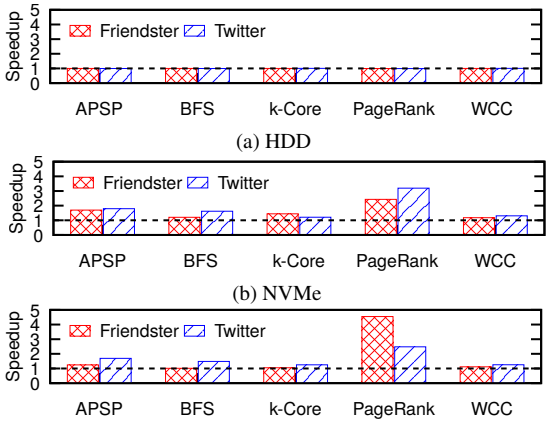


Figure 17: Bitmap-based IO performance on traces.

as quickly as possible, namely Financial 1-2 and Web-Search 1-3 from UMass Trace Repository [3]. On average, the Bitmap is 38× faster than Linux IO, with the maximum speedup of 74× obtained on Financial2 (from 94.2 to 1.26 seconds). The improvement comes mostly from more (9.3×) deduplicated IOs and more aggressive IO merging.

Figure 18 further studies the impacts of bitmap based IO on hard disk (HDD), NVMe and Ramdisk. In this test, we use five Seagate 7200RPM SATA III hard drives in a Raid-0 configuration, and one Samsung 950 Pro NVMe device. One can see that compared to the pluglist based method, although bitmap improves hard disk performance only marginally (1% on average), faster storage devices such as NVMe and Ramdisk are able to achieve about 70% improvement in IO performance.



(a) HDD

(b) NVMe

(c) Ramdisk

Figure 18: Bitmap performance on HDD, NVMe and Ramdisk.

7.4 Scalability, Utility, and Throughput

This section studies the scalability of Graphene with respect to the number of SSDs. Recall that Graphene uses two threads per SSD, one IO and another compute. Using a single thread would fail to fully utilize the bandwidth of an SSD. As shown in Figure 19, Graphene achieves an average 3.3× speedup on the Kron30 graph when scaling from a single SSD (two threads) to eight SSDs (16 threads). Across different applications, SpMV enjoys the biggest 3.7× speedup and PageRank the smallest 2.6×. The small performance gain from 8 to 16 SSDs is due to the shift of the bottleneck from IO to CPU.

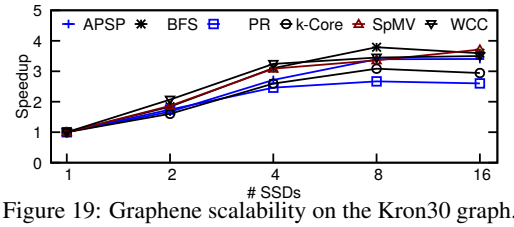


Figure 19: Graphene scalability on the Kron30 graph.

Recall that IO utility is defined as the ratio of useful data and total data loaded, we evaluate the IO utility when using 512-byte IO vs. 4KB IO on various algorithms and graph datasets. As presented in Figure 20, Graphene achieves 20% improvement on average. For APSP and BFS, one can see about 30% improvement with the best benefit of 50% on UK. Similar speedups can also be observed for K-Core and WCC. In contrast, PageRank and SpMV present minimal benefit because the majority of their iterations load the whole graph.

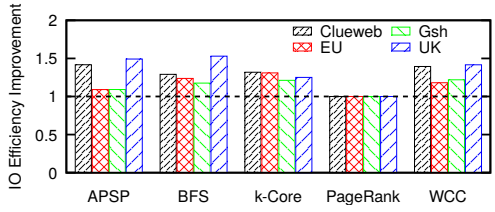


Figure 20: Utility of 512-byte vs. 4KB IO.

To demonstrate the IO loads of different disks in Graphene, we further examine the throughput of 16 SSDs for two applications, BFS and PageRank. Figure 21 show the throughput for the fastest (max) and slowest (min) SSDs, as well as the median throughput. Clearly, the 16 SSDs are able to deliver similar IO performance for most of run, with an average difference of 6 to 15 MB/s (5-7% for PageRank and BFS). For both algorithms, the slowest disk does require extra time to complete the processing, which we leave for future research to close the gap.

8 Related Work

Recent years have seen incredible advances in graph computation, to name a few, in-memory systems [27, 40, 47], distributed systems [10, 11, 20, 38, 46, 61], external-memory processing [21, 25, 31, 32, 35, 36, 43, 44, 57, 62, 63], and accelerator-based systems [30, 33, 58]. In this section, we compare Graphene with existing projects from three aspects: programming, IO, and partitioning.

Programming. Prior projects, regardless of *Think like a vertex* [10, 32, 36, 58], *Think like an edge* [31, 43, 44], *Think like an embedding* [50], or *Think like a graph* [52], center around simplifying computation related programming efforts. In comparison, Graphene aims for ease of IO management with the new IO iterator API.

IO optimization is the main challenge for external

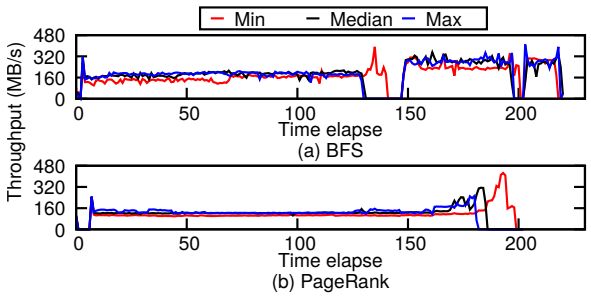


Figure 21: Throughputs of the fastest (max) and slowest (min) SSDs, and median throughput out of 16 SSDs.

memory graph engines, for which Graphene develops a set of fine-grained IO management techniques, including using 512-byte IO block and bitmap-based selective IO. Our approach achieves high efficiency compared to full IO [32, 36, 43, 44]. Compared to GridGraph [63] and FlashGraph [62], Graphene introduces a finer grained method that supports global range IO adjustment and reduces IO requests by 3×. Also, Graphene shows that asynchronous IOs, when carefully managed, are very beneficial for external memory systems. While hugepages are not new to graph systems [40, 62], Graphene addresses the issue of potentially low memory utilization by constructing IO buffers to share hugepages.

Partition optimization. A variety of existing projects [12, 20, 62, 63] rely on conventional 2D partitioning [9] to balance the workload. In contrast, Graphene advocates that it is the amount of edges, rather than vertices, in a partition that determines the workload. The new row-column balanced partition can help achieve up to 2.7× speedup on a number of graph algorithms.

9 Conclusion and Future work

In this paper, we have designed and developed Graphene that consists of a number of novel techniques including IO centric processing, Bitmap-based asynchronous IO, hugepage support, data and workload balancing. It allows the users to treat the data as in-memory, while delivering high-performance on SSDs. The experiments show that Graphene is able to perform comparably against in-memory processing systems on large-scale graphs, and also runs several times faster than existing external-memory processing systems.

10 Acknowledgments

The authors thank the anonymous reviewers and our shepherd Brad Morrey for their valuable suggestions that help improve the quality of this paper. The authors also thank Da Zheng, Frank Mcsherry, Xiaowei Zhu, and Wenguang Chen for their help and discussion. This work was supported in part by National Science Foundation CAREER award 1350766 and grant 1618706.

References

[1] Friendster Network Dataset – KONECT. http://konect.uni-koblenz.de/networks/friendster, 2016.

[2] Twitter (MPI) Network Dataset – KONECT. http://konect.uni-koblenz.de/networks/twitter_mpi, 2016.

[3] UMASS Trace Repository. http://traces.cs.umass.edu/, 2016.

[4] Scott Beamer, Krste Asanović, and David Patterson. Direction-Optimizing Breadth-First Search. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[5] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. BUbiNG: Massive Crawling for the Masses. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web (WWW)*, 2014.

[6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web (WWW)*, 2011.

[7] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW)*, Manhattan, USA, 2004.

[8] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[9] Aydin Buluç and Kamesh Madduri. Parallel Breadth-First Search on Distributed Memory Systems. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[10] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems (Eurosys)*, 2015.

[11] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Faking the Pulse of A Fast-Changing and Connected World. In *Proceedings of the european conference on Computer Systems (Eurosys)*, 2012.

[12] Jatin Chhugani, Nadathur Satish, Changkyu Kim, Jason Sewall, and Pradeep Dubey. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[13] Clueweb dataset from WebGraph. http://law.di.unimi.it/webdata/clueweb12/, 2012.

[14] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. Graph-Based Technologies For Intelligence Analysis. *Communications of the ACM*, 2004.

[15] Antonio Del Sol, Hirotomo Fujihashi, and Paul O'Meara. Topology of Small-World Networks of Protein-Protein Complex Structures. *Bioinformatics*, 2005.

[16] Xiaoning Ding, Kaibo Wang, and Xiaodong Zhang. ULCC: A User-Level Facility for Optimizing Shared Cache Performance on Multicores. In *Proceedings of the SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2011.

[17] Christian Doerr and Norbert Blenn. Metric Convergence in Social Network Sampling. In *Proceedings of the 5th ACM workshop on HotPlanet*, 2013.

[18] EU dataset from WebGraph. http://law.di.unimi.it/webdata/eu-2015/, 2015.

[19] Fixing asynchronous I/O, again. https://lwn.net/Articles/671649/, 2016.

[20] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2012*.

[21] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[22] Graph500. http://www.graph500.org/.

[23] Gsh dataset from WebGraph. http://law.di.unimi.it/webdata/gsh-2015/, 2015.

[24] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran,

Wenguang Chen, and Enhong Chen. Chronos: A Graph Engine For Temporal Graph Analysis. In *Proceedings of the european conference on Computer systems (Eurosys)*, 2014.

[25] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in A Single PC. In *Proceedings of the 19th SIGKDD international conference on Knowledge discovery and data mining (KDD)*, 2013.

[26] Taher H Haveliwala. Topic-Sensitive Pagerank. In *Proceedings of the 11th international conference on World Wide Web (WWW)*, 2002.

[27] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL For Easy and Efficient Graph Analysis. In *Proceedings of the international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[28] Sungpack Hong, Nicole C Rodia, and Kunle Olukotun. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[29] Hawoong Jeong, Sean P Mason, A-L Barabási, and Zoltan N Oltvai. Lethality and Centrality in Protein Networks. *Nature*, 2001.

[30] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the international symposium on High performance distributed computing (HPDC)*, 2014.

[31] Pradeep Kumar and H Howie Huang. G-Store: High-Performance Graph Store for Trillion-Edge Processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[32] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2012*.

[33] Hang Liu and H Howie Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[34] Hang Liu, H. Howie Huang, and Yang Hu. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.

[35] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment*, 2012.

[36] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the SIGMOD International Conference on Management of data (SIGMOD)*, 2010.

[37] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-Core Decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[38] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth Symposium on Operating Systems Principles (SOSP)*, 2013.

[39] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI)*, 2002.

[40] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2013.

[41] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order To the Web. In *Stanford InfoLab Technical Report*, 1999.

[42] Performance Issues with Transparent Huge Pages (THP). `https://blogs.oracle.com/linux/entry/performance_issues_with_transparent_huge`, 2013.

[43] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[44] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2013*.

[45] Ahmet Erdem Sariyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. Streaming Algorithms for k-Core Decomposition. *Proceedings of the VLDB Endowment*, 2013.

[46] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, 2013.

[47] Julian Shun and Guy E Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2013.

[48] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. BFS and Coloring-Based Parallel Algorithms For Strongly Connected Components and Related Problems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[49] Samsung 850 EVO SSD. `http://www.samsung.com/semiconductor/minisite/ssd/product/consumer/850evo.html`, 2015.

[50] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. Arabesque: A System For Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[51] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. The More the Merrier: Efficient Multi-Source Graph Traversal. *Proceedings of the VLDB Endowment*, 2014.

[52] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From Think Like a Vertex to Think Like a Graph. *Proceedings of the VLDB Endowment*, 2013.

[53] Timely Dataflow Blog. `https://github.com/frankmcsherry/timely-dataflow`, 2016.

[54] Transparent huge pages in 2.6.38. `http://lwn.net/Articles/423584/`, 2011.

[55] UK dataset in WebGraph. `http://law.di.unimi.it/webdata/uk-2014/`, 2014.

[56] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX Annual Technical Conference (ATC)*, 2016.

[57] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. GraphQ: Graph Query Processing with Abstraction Refinement—Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *Proceedings of the Usenix Annual Technical Conference (ATC)*, 2015.

[58] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[59] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. GRAM: Scaling Graph Computation to the Trillions. In *Proceedings of the Sixth Symposium on Cloud Computing (SoCC)*, 2015.

[60] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards Practical Page Coloring-Based Multicore Cache Management. In *Proceedings of the European conference on Computer systems (Eurosys)*, 2009.

[61] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An Asynchronous Graph Processing Framework For Delta-Based Accumulative Iterative Computation. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[62] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[63] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX Annual Technical Conference (ATC)*, 2015.

vNFS: Maximizing NFS Performance with Compounds and Vectorized I/O

Ming Chen, Dean Hildebrand*, Henry Nelson+, Jasmit Saluja,
Ashok Sankar Harihara Subramony, and Erez Zadok
*Stony Brook University, *IBM Research - Almaden, +Ward Melville High School*

Abstract

Modern systems use networks extensively, accessing both services and storage across local and remote networks. Latency is a key performance challenge, and packing multiple small operations into fewer large ones is an effective way to amortize that cost, especially after years of significant improvement in bandwidth but not latency. To this end, the NFSv4 protocol supports a *compounding* feature to combine multiple operations. Yet compounding has been underused since its conception because the synchronous POSIX file-system API issues only one (small) request at a time.

We propose *vNFS*, an NFSv4.1-compliant client that exposes a vectorized high-level API and leverages NFS *compound procedures* to maximize performance. We designed and implemented vNFS as a user-space RPC library that supports an assortment of bulk operations on multiple files and directories. We found it easy to modify several UNIX utilities, an HTTP/2 server, and Filebench to use vNFS. We evaluated vNFS under a wide range of workloads and network latency conditions, showing that vNFS improves performance even for low-latency networks. On high-latency networks, vNFS can improve performance by as much as two orders of magnitude.

1 Introduction and Background

Modern computer hardware supports high parallelism: a smartphone can have eight cores and a NIC can have 256 queues. Although parallelism can improve throughput, many standard software protocols and interfaces are unable to leverage it and are becoming bottlenecks due to serialization of calls [8, 16]. Two notable examples are HTTP/1.x and the POSIX file-system API, both of which support only one synchronous request at a time (per TCP connection or per call). As Moore's Law fades [44], it is increasingly important to make these protocols and interfaces parallelism-friendly. For example, HTTP/2 [5] added support for sending multiple requests per connection. However, to the best of our knowledge little progress has been made on the file-system API.

In this paper we similarly propose to batch multiple file-system operations. We focus particularly on the Network File System (NFS), and study how much performance can be improved by using a file-system API friendly to NFSv4 [34, 35]; this latest version of NFS supports *compound procedures* that pack multiple operations into a single RPC so that only one round trip is needed to process them. Unfortunately, although NFS compounds have been designed, standardized, and implemented in most NFS clients and servers, they are underused—mainly because of the limitations of the low-level POSIX file-system interface [8].

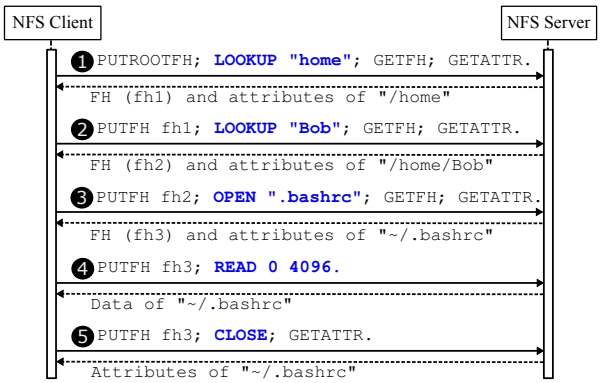


Figure 1: NFS compounds used by the in-kernel NFS client to read a small file. Each numbered request is one compound, with its operations separated by semicolons. The operations use an NFSv4 server-side state, the current filehandle (CFH). PUTROOTFH *sets the CFH to the FH of the root directory;* PUTFH *and* GETFH *set or retrieve the CFH;* LOOKUP *and* OPEN *assume that the CFH is a directory, find or open the specified name inside, and set it as the CFH;* GETATTR, READ, *and* CLOSE *all operate on the file indicated by the CFH.*

To explain the operations and premise of NFS4's compound procedures, we discuss them using several instructive figures. We start with Figure 1, which shows how reading a small file is limited by the POSIX API. This simple task involves four syscalls (`stat`, `open`, `read`, and `close`) that generate five compounds, each incurring a round trip to the server. Because compounds are initiated by low-level POSIX calls, each compound contains only one significant operation (in bold blue), with the rest being trivial operations such as PUTFH and GETFH. Compounds reduced the number of round trips slightly by combining the syscall operations (LOOKUP, OPEN, READ) with NFSv4 state-management operations (PUTFH, GETFH) and attribute retrieval (GETATTR), but the syscall operations themselves could not be combined due to the serialized nature of the POSIX API.

Ideally, a small file should be read using only one NFS compound (and one round trip), as shown in Figure 2. This would reduce the read latency by 80% (by removing four of the five round trips). We can even read multiple files using a single compound, as shown in Figure 3. All these examples use the standard (unmodified) NFSv4 protocol. SAVEFH and RESTOREFH operate on

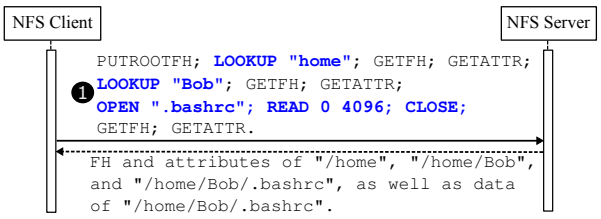


Figure 2: Reading `/home/Bob/.bashrc` using only one compound. This single compound is functionally the same as the five in Figure 1, but uses only one network round trip.

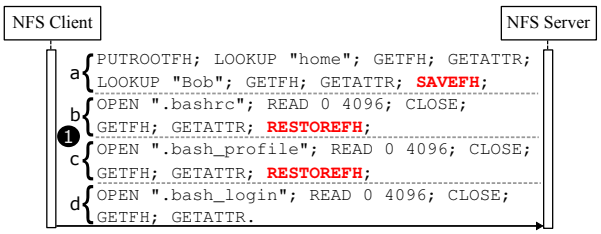


Figure 3: One NFS compound that reads three files. The operations can be divided into four groups: (a) sets the current and saved filehandle to `/home/Bob`; (b), (c), and (d) read the files `.bashrc`, `.bash_profile`, and `.bash_login`, respectively. SAVEFH *and* RESTOREFH *(in red) ensure that the CFH is `/home/Bob` when opening files. The reply is omitted.*

the *saved filehandle* (SFH), an NFSv4 state similar to the *current filehandle* (CFH). SAVEFH copies the CFH to the SFH; RESTOREFH restores the CFH from the SFH.

For compounds to reach their full potential, we need a file-system API that can convey high-level semantics and batch multiple operations. We designed and developed *vNFS*, an NFSv4 client that exposes a high-level vectorized API. vNFS complies with the NFSv4.1 standard, requiring no changes to NFS servers. Its API is easy to use and flexible enough to serve as a building block for new higher-level functions. vNFS is implemented entirely in user space, and thus easy to extend.

vNFS is especially efficient and convenient for applications that manipulate large amounts of metadata or do small I/Os. For example, vNFS lets `tar` read many small files using a single RPC instead of using multiple RPCs for each; it also lets `untar` set the attributes of many extracted files at once instead of making separate system calls for each attribute type (owner, time, etc.).

We implemented vNFS using the standard NFSv4.1 protocol, and added two small protocol extensions to support file appending and copying. We ported GNU's Coreutils package (`ls`, `cp`, and `rm`), `bsdtar`, `nghttp2` (an HTTP/2 server), and Filebench [15, 40] to vNFS. In general, we found it easy to modify applications to use vNFS. We ran a range of micro- and macro-benchmarks on networks with varying latencies, showing that vNFS can speed such applications by 3–133× with small network latencies (≤5.2ms), and by up to 263× with a 30.2ms latency.

The rest of this paper is organized as follows. Section 2 summarizes vNFS's design. Section 3 details the vectorized high-level API. Section 4 describes the implementation of our prototype. Section 5 evaluates the performance and usability of vNFS by benchmarking applications we ported. Section 6 discusses related work and Section 7 concludes.

2 Design Overview

In this section we summarize vNFS's design, including our goals, choices we made, and the architecture.

2.1 Design Goals

Our design has four goals, in order of importance:

- **High performance**: vNFS should considerably outperform existing NFS clients and improve both latency and throughput, especially for workloads that emphasize metadata and small I/Os. Performance for other workloads should be comparable.
- **Standards compliance**: vNFS should be fully compliant with the NFSv4.1 protocol so that it can be used with any compliant NFS server.
- **Easy adoption**: vNFS should provide a general API that is easy for programmers to use. It should be familiar to developers of POSIX-compliant code to enable smooth and incremental adoption.
- **Extensibility**: vNFS should make it easy to add functions to support new features and performance improvements. For example, it should be simple to add support for Server Side Copy (a feature in the current NFSv4.2 draft [17]) or create new application-specific high-level APIs.

2.2 Design Choices

The core idea of vNFS is to improve performance by using the compounding feature of standard NFS. We discuss the choices we faced and justify those we selected to meet the goals listed in Section 2.1.

Overt vs. covert coalescing. To leverage NFS compounds, vNFS uses a high-level API to overtly express the intention of compound operations. An alternative would be to covertly coalesce operations under the hood while still using the POSIX API. Covert coalescing is a common technique in storage and networking; for example, disk I/O schedulers combine many small requests into a few larger ones to minimize seeks [3]; and Nagle's TCP algorithm coalesces small outbound packets to amortize overhead for better network utilization [21].

Although overt compounding changes the API, we feel it is superior to covert coalescing in four important respects: **(1)** By using a high-level API, overt compounding can batch dependent operations, which are impossible to coalesce covertly. For example, using the

POSIX API, we cannot issue a `read` until we receive the reply from the preceding `open`. **(2)** Overt compounding can use a new API to express high-level semantics that cannot be efficiently conveyed in low-level primitives. NFSv4.2's Server Side Copy is one such example [17]. **(3)** Overt compounding improves both throughput and latency, whereas covert coalescing improves throughput at the cost of latency, since accumulating calls to batch together inherently requires waiting. Covert coalescing is thus detrimental to metadata operations and small I/Os that are limited by latency. This is important in modern systems with faster SSDs and 40GbE NICs, where latency has been improving much slower than raw network and storage bandwidth [33]. **(4)** Overt compounding allows implementations to use all possible information to maximize performance; covert coalescing depends on heuristics, such as timing and I/O sizes, that can be suboptimal or wrong. For example, Nagle's algorithm can interact badly with Delayed ACK [10].

Vectorized vs. start/end-based API. Two types of APIs can express overt compounding: a vectorized one that compounds many desired low-level NFS operations into a single high-level call, or an API that uses calls like `start_compound` and `end_compound` to combine all low-level calls in between [32]. We chose the vectorized API for two reasons: **(1)** A vectorized API is easier to implement than a start/end-based one. Users of a start/end-based API might mix I/Os with other code (such as looping and testing of file-system states), which NFS compounds cannot support. **(2)** A vectorized API logically resides at a high level and is more convenient to use, whereas using a low-level start/end-based API is more tedious for high-level tasks (similar to C++ programming vs. assembly).

User-space vs. in-kernel implementation. A kernel-space implementation of vNFS would allow it to take advantage of the kernel's page and metadata caches and use the existing NFS code base. However, we chose to design and implement vNFS in user space for two reasons: **(1)** Adding a user-space API is much easier than adding system calls to the kernel and simplifies future extensions; and **(2)** User-space development and debugging is faster and easier. Although an in-kernel implementation might be faster, prior work indicates that the performance impact can be minimal [39], and the results in this paper demonstrate substantial performance improvements even with our user-space approach.

2.3 Architecture

Figure 4 shows the architecture of vNFS, which consists of a library and a client. Instead of using the POSIX API, applications call the high-level vectorized API provided by the vNFS library, which talks directly to the vNFS client. The vNFS library facilitates application adoption,

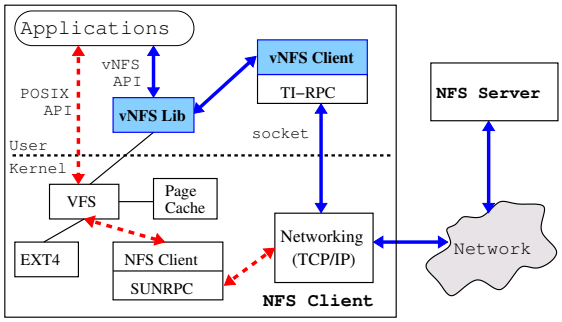


Figure 4: vNFS Architecture. The blue arrows show vNFS's data path, and the dashed red arrows show the in-kernel NFS client's data path. The vNFS library and client (blue shaded boxes) are new components we added; the rest already existed.

since most modern applications are developed using libraries and frameworks instead of OS system calls [2]. To provide generic support and encourage incremental adoption, the library detects when compound operations are unsupported, and in that case converts vNFS operations into standard POSIX primitives. Thus, the vNFS library can also be used with file systems that do not support compounding, e.g., as a utility library for batching file-system operations.

The vNFS client accepts vectorized operations from the library, puts as many of them into each compound as possible, sends them to the NFS server using Transport-Independent RPC (TI-RPC), and finally processes the reply. Note that existing NFSv4 servers already support compounds and can be used with vNFS without change. TI-RPC is a generic RPC library without the limitations of Linux's in-kernel SUNRPC (e.g., supporting only a single data buffer per call); TI-RPC can also run on top of TCP, UDP, and RDMA. Like the in-kernel NFS client, the vNFS client also manages NFSv4's client-side states such as sessions, etc.

3 vNFS API

This section details vNFS's vectorized API (listed in Table 1). Each API function expands its POSIX counterparts to operate on a vector of file-system objects (e.g., files, directories, symbolic links). vNFS functions handle errors in a standard manner: return results for successful operations, report the index of the first failed operation in a compound (if any), and ignore any remaining operations that were not executed by the server. Figure 5 demonstrates the use of vNFS API to read three small files in one NFS compound. To simplify programming, vNFS also provides utility functions for common tasks such as recursively removing a whole directory, etc.

`vread/vwrite`. These functions can read or write multiple files using a single compound, with automatic on-demand file opening and closing. These calls boost throughput, reduce latency, and simplify programming. Both accept a vector of I/O structures, each containing

Function	Description
vopen vclose	Open/close many files.
vread vwrite	Read/write/create/append files with automatic file opening and closing.
vgetattrs vsetattrs	Get/set multiple attributes of file-system objects.
vsscopy vcopy	Copy files in whole or in part with/out Server Side Copy.
vmkdir	Create directories.
vlistdir	List (recursively) objects and their attributes in directories.
vsymlink	Create many symbolic links.
vreadlink	Read many symbolic links.
vhardlink	Create many hard links.
vremove	Remove many objects.
vrename	Rename many objects.

Table 1: vNFS vectorized API functions. Each function has two return values: an error code and a count of successful operations. NFS servers stop processing the remaining operations in a compound once any operation inside failed. To facilitate gradual adoption, vNFS also provides POSIX-like scalar API functions, omitted here for brevity. Each vNFS function has a version that does not follow symbolic links, also omitted.

a `vfile` structure (Figure 5), offset, length, buffer, and flags. Our vectorized operations are more flexible than the `readv` and `writev` system calls, and can operate at many (discontinuous) offsets of multiple files in one call. When generating compound requests, vNFS adds OPENs and CLOSEs for files represented by paths; files represented by descriptors do not need that since they are already open. OPENs and CLOSEs are coalesced when possible, e.g. when reading twice from one file.

The length field in the I/O structure also serves as an output, returning the number of bytes read or written. The structure has several flags that map to NFS's internal Boolean arguments and replies. For example, the flag `is_creation` corresponds to the NFS `OPEN4_CREATE` flag, telling `vwrite` to create the target file if necessary. `is_write_stable` corresponds to NFS's WRITE `DATA_SYNC4` flag, causing the server to save the data to stable storage, avoiding a separate NFS COMMIT. Thus, a single `vwrite` can achieve the effect of multiple writes and a following `fsync`, which is a common I/O pattern (e.g., in logging or journaling).

■ **State management** NFSv4 is stateful, and OPEN is a state-mutating operation. The NFSv4 protocol requires a client to open a file before reading or writing it. Moreover, READ and WRITE must provide the *stateid* (an ID uniquely identifying a server's state [34]) returned by the preceding OPEN. Thus, state management is a key challenge when `vread` or `vwrite` adds OPEN and READ/WRITE calls into a single compound. vNFS solves this by using the NFS *current stateid*, which is a server-side state similar to the current filehandle. To ensure that the NFS server always uses the correct state, `vread` and `vwrite` take advantage of NFSv4's special support for using the current stateid [34, Section 8.2.3].

```
struct vfile {
  enum VFILETYPE type;    // PATH or DESCRIPTOR
  union {
    const char *path;  // When "type" is PATH,
    int fd;       // or (vNFS file) DESCRIPTOR.
  };
};
// The "vio" I/O structure contains a vfile.
struct vio ios[3] = {
  { .vfile = { .type = PATH,
               .path = "/home/Bob/.bashrc" },
    .offset = 0,
    .length = 64 * 1024,
    .data = buf1, // pre-allocated 64KB buffer
    .flags = 0,  // contains an output EOF bit
  }, ...   // two other I/O structures omitted
};
struct vres r = vread(ios, 3); // read 3 files
```

Figure 5: A simplified C code sample of reading three files at once using the vectorized API.

■ **Appending** `vwrite` also adds an optional small extension to the NFSv4.1 protocol to better support appends. As noted in the Linux manual page for `open(2)` [28], "O_APPEND may lead to corrupted files on NFS filesystems if more than one process appends data to a file at once." The base NFSv4 protocol does not support appending, so the kernel NFS client appends by writing to an offset equal to the current known file size. This behavior is inefficient as the file size must first be read separately, and it is vulnerable to TOCTTOU (time-of-check-to-time-of-use) attacks. Our extension uses a special offset value (`UINT64_MAX`) in the I/O structure to indicate appending, making appending reliable with a tiny (5 LoC) change to the NFS server.

vopen/vclose. Using `vread` and `vwrite`, applications can access files without explicit opens and closes. Our API still supports `vopen` and `vclose` operations, which add efficiency for large files that are read or written many times. `vopen` and `vclose` are also important for maintaining NFS's close-to-open cache consistency [25]. `vopen` opens multiple files (specified by paths) in one RPC, including LOOKUPs needed to locate their parent directories, as shown in Figure 3. Each file has its own open flags (read, write, create, etc.), which is useful when reading and writing are intermixed, such as external merge sorting. We also offer `vopen_simple`, which uses a common set of flags and mode (in case of creation) for all files. Once opened, a file is represented by a file descriptor, which is an integer index into an internal table that keeps states (file cursor, NFSv4 stateid and sequenceid [34], etc.) of open files. `vclose` closes multiple opened files and releases their resources.

vgetattrs/vsetattrs. These two functions manipulate several attributes of many files at once, combining multiple system calls (`chmod`, `chown`, `utimes`, and `truncate`, etc.) into a single compound, which is especially useful for tools like `tar` and `rsync`. The aging POSIX API is the only restriction on setting many attributes at once: the Linux kernel VFS already supports multi-attribute operations using the `setattr` method of `inode_operations`, and the NFSv4 protocol has similar SETATTRs support. `vgetattrs` and `vsetattrs` use an array of attribute structures as both inputs and outputs. Each structure contains a `vfile` structure, all attributes (mode, size, etc.), and a bitmap showing which attributes are in use.

vsscopy/vcopy. File copying is so common that Linux has added the `sendfile` and `splice` system calls to support it. Unfortunately, NFS does not yet support copying and clients must use READs and WRITEs instead, wasting time and bandwidth because data has to be read over the network and immediately written back. It is more efficient to ask the NFS server to copy the files directly on its side. This *Server Side Copy* (SSC) has already been proposed for the upcoming NFSv4.2 [17]. Being forward-looking, we included `vsscopy` in vNFS to copy many files (in whole or in part) using SSC; however, SSC requires server enhancements.

`vsscopy` accepts an array of copy structures, each containing the source file and offset, the destination file and offset, and the length. The destination files are created by `vsscopy` if necessary. The length can be `UINT64_MAX`, in which case the effective length is the distance between the source offset and the end of the source file. `vsscopy` can use a single RPC to copy many files in their entirety. The copy structures return the number of copied bytes in the length fields.

`vcopy` has the same effect but does not use SSC. `vcopy` is useful when the NFS server does not support SSC; `vcopy` can copy N small files using three RPCs (a compound for each of `vgetattrs`, `vread`, and `vwrite`) instead of $7 \times N$ RPCs (2 OPENs, 2 CLOSEs, 1 GETATTR, 1 READ, and 1 WRITE for each file). A future API could provide only `vcopy` and silently switch to `vsscopy` when SSC is available; we include `vsscopy` separately in this paper for comparison with `vcopy`.

vmkdir. vNFS provides `vmkdir` to create multiple directories at once (such as directory trees), which is common in tools such as `untar`, `cmake`, and recursive `cp`. vNFS's utility function `ensure_directory` uses `vmkdir` to ensure a deep directory and all its ancestors exist. Consider `"/a/b/c/d"` for example: the utility function first uses `vgetattrs` with arguments `["/a"; "/a/b"; ...]` to find out which ancestors exist and then creates the missing directories using `vmkdir`. Note that simply calling `vmkdir` with vector arguments `["/a"; "/a/b"; ...]` does not work: the NFS server will fail (with `EEXIST`) when trying to recreate the first existing ancestor and stop processing all remaining operations.

vlistdir. This function speeds up directory listing with four improvements to `readdir`: (1) `vlistdir` lists multiple directories at once; (2) a prior `opendir` is not necessary for listing; (3) `vlistdir` retrieves attributes along with directory entries, saving subsequent `stats`; (4) `vlistdir` can work recursively. It can be viewed as a fast vectorized `ftw(3)` that reads NFS directory contents using as few RPCs as possible.

`vlistdir` takes five arguments: an array of directories to list, a bitmap indicating desired attributes, a flag to select recursive listing, a user-defined callback function (similar to `ftw`'s second argument [27]), and a user-provided opaque pointer that is passed to the callback. `vlistdir` processes directories in the order given; recursion is breadth-first. However, directories at the same level in the tree are listed in an arbitrary order.

vsymlink/vreadlink/vhardlink. These three vNFS operations allow many links to be created or read at once. Together with `vlistdir`, `vsymlink` can optimize operations like `"cp -sr"` and `"lndir"`. All three functions accept a vector of paths and a vector of buffers containing the target paths.

vremove. `vremove` removes multiple files and directories at once. Although `vremove` does not support recursive removal, a program can achieve this effect with a recursive `vlistdir` followed by properly ordered `vremoves`; vNFS provides a utility function `rm_recursive` for this purpose.

vrename. Renaming many files and directories is common, for example when organizing media collections. Many tools [1, 22, 24, 45] have been developed just for this purpose. vNFS provides `vrename` to facilitate and speed up bulk renaming. `vrename` renames a vector of source paths to a vector of destination paths.

4 Implementation

We have implemented a prototype of vNFS in C/C++ on Linux. As shown in Figure 4, vNFS has a library and a client, both running in user space. The vNFS library implements the vNFS API. Applications use the library by including the API header file and linking to it. For NFS files, the library redirects API function calls to the vNFS client, which builds large compound requests and sends them to the server via the TI-RPC library. For non-NFS files, the library translates the API functions into POSIX calls, and therefore can also be used as a utility library. (Our current prototype considers a file to be on NFS if it is under any exported directory specified in vNFS's configuration file.) The vNFS client builds on NFS-Ganesha [12, 30], an open-source user-space NFS

server. NFS-Ganesha can export files stored in many backends, such as XFS and GPFS. Our vNFS prototype uses an NFS-Ganesha backend called PROXY, which exports files from *another* NFS server and can be repurposed as a user-space NFS client. The original PROXY backend used NFSv4.0; we added NFSv4.1 support including session management [34]. Our prototype implementation added 10,632 lines of C/C++ code and deleted 1,407. vNFS is thread-safe; we have tested it thoroughly.

RPC size limit. The vNFS API functions (shown in Table 1) do not impose a limit on the number of operations per call. However, each RPC has a configurable memory size limit, defaulting to 1MB. We ensure that vNFS does not generate RPC requests larger than that limit no matter how many operations an API call contains. Therefore, we split long arguments into chunks and send one compound request for each chunk. We also merge RPC replies upon return, to hide any splitting.

Our splitting avoids generating small compounds. For data operations (`vread` and `vwrite`), we can easily estimate the sizes of requests and replies based on buffer lengths, so we split a compound only when its size becomes close to 1MB. (The in-kernel NFS client similarly splits large READs and WRITEs according to the `rsize` and `wsize` mount options, which also default to 1MB.) For metadata operations, it is more difficult to estimate the reply sizes, especially for READDIR and GETATTR. We chose to be conservative and simply split a compound of metadata operations whenever it contains more than k NFS operations. We chose a default of 256 for k, which enables efficient concurrent processing by the NFS server, and yet is unlikely to exceed the size limit. For example, when listing the Linux source tree, the average reply size of READDIR—the largest metadata operation—is around 3,800 bytes. If k is still too large (e.g., when listing large directories), the server will return partial results and use cookies to indicate where to resume the call for follow-up requests.

Protocol extensions. vNFS contains two extensions to the NFSv4.1 protocol to support file appending (see Section 3 [¶`vread`/`vwrite`]) and Server Side Copy (see Section 3 [¶`vsscopy`/`vcopy`]). Both extensions require changes to the protocol and the NFS server. We have implemented these changes in our server, which is based on NFS-Ganesha [11, 12, 30]. The file-appending extension was easy to implement, adding only an `if` statement with 5 lines of C code. In the NFS server, we only need to use the file size as the effective offset whenever the write offset is `UINT64_MAX`.

Our implementation of Server Side Copy follows the design proposed in the NFSv4.2 draft [17]. We added the new COPY operation to our vNFS client and the NFS-Ganesha server. On the server side, we copy data using `splice(2)`, which avoids unnecessarily moving data across the kernel/user boundary. This extension added 944 lines of C code to the NFS-Ganesha server.

Path compression. We created an optimization that reduces the number of LOOKUPs when a compound's file paths have locality. The idea is to shorten paths that have redundancy by making them relative to preceding ones in the same compound. For example, when listing the directories `"/1/2/3/4/5/6/7/a"` and `"/1/2/3/4/5/6/7/b"`, a naïve implementation would generate eight LOOKUPs per directory (one per component). In such cases, we replace the path of the second directory with `"../b"` and use only one LOOKUPP and one LOOKUP; LOOKUPP sets the current filehandle to its parent directory. This simple technique saves as many as six NFS operations for this example.

Note that LOOKUPP produces an error if the current filehandle is not a directory, because most file systems have metadata recording parents of directories, but not parents of files. In that case, we use SAVEFH to remember the deepest common ancestor in the file-system tree (i.e., `"/1/2/3/4/5/6/7"` in the above example) of two adjacent files, and then generate a RESTOREFH and LOOKUPs. (However, this approach cannot be used for LINK, RENAME, and COPY, which already use the saved filehandle for other purposes.) We use this optimization only when it saves NFS operations: for example, using `"../../c/d"` does not save anything for paths `"/1/a/b"` and `"/1/c/d"`.

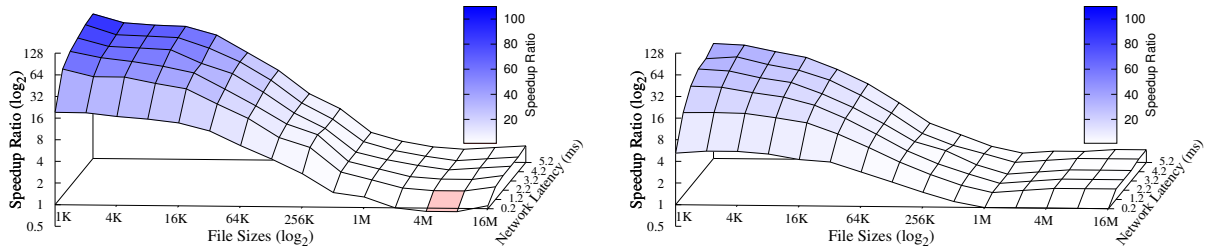
Client-side caching. Our vNFS prototype does not yet have a client-side cache, which would be useful for rereading recent data and metadata, streaming reads, and asynchronous writes. We plan to add it in the future. Compared to traditional NFS clients, vNFS does not complicate failure handling in the presence of a dirty client-side cache: cached dirty pages (not backed by persistent storage) are simply dropped upon a client crash; dirty data in a persistent cache (e.g., FS-Cache [19]), which may be used by a client holding write delegations, can be written to the server even faster during client crash recovery. Note that a client-side cache does not hold dirty metadata because all metadata changes are performed synchronously in NFS (except with directory delegations, which Linux has not yet implemented).

5 Evaluation

To evaluate vNFS, we ran micro-benchmarks and also ported applications to use it. We now discuss our porting experience and evaluate the resulting performance.

5.1 Experimental Testbed Setup

Our testbed consists of two identical Dell PowerEdge R710 machines running CentOS 7.0 with a 3.14 Linux kernel. Each machine has a six-core Intel Xeon X5650 CPU, 64GB of RAM, and an Intel 10GbE NIC. One ma-



(a) Reading whole files

(b) Writing whole files

Figure 6: vNFS's speedup ratio (the vertical Z-axis, in logarithmic scale) relative to the baseline when reading and writing 1,000 equally-sized files, whose sizes (the X-axis) varied from 1KB to 16MB. vNFS is faster than (blue), equal to (white), or slower than (red) the baseline when the speedup ratio is larger than, equal to, or smaller than 1.0, respectively. The network latency (Y-axis) starts from 0.2ms (instead of zero) because that is the measured base latency of our testbed (see Section 5.1).

chine acts as the NFS server and runs NFS-Ganesha with our file-appending and Server Side Copy extensions; the other machine acts as a client and runs vNFS. The NFS server exports to the client an Ext4 file system, stored on an Intel DC S3700 200GB SSD. The two machines are directly connected to a Dell PowerConnect 8024F 10GbE switch, and we measured an average RTT of 0.2ms between them. To emulate different LAN and WAN conditions, we injected delays of 1–30ms into the outbound link of the server using `netem`.

To evaluate vNFS's performance, we compared it with the in-kernel NFSv4.1 client (called *baseline*), which mounts the exported directory using the default options: the attribute cache (`ac` option) is enabled and the maximum read/write size (`rsize`/`wsize` options) is 1MB. Our vNFS prototype does not use `mount`, but instead reads the exported directory from a configuration file. We ran each experiment at least three times and plotted the average value. We show the standard deviation as error bars, which are invisible in most figures because of their tiny values. Before each run, we flushed the page and dentry caches of the in-kernel client by unmounting and re-mounting the NFS directory. vNFS has no cache. The NFS-Ganesha server uses an internal cache, plus the OS's page and dentry caches. To quantify the effort of porting applications, we report the LoC change for each application including the error-handling code.

5.2 Micro-workloads

Small vs. big files. vNFS's goal is to improve performance for workloads with many small NFS operations, while staying competitive for data-intensive workloads. To test this, we compared the time used by vNFS and the baseline to read and write 1,000 equally-sized files in their entirety while varying the file size from 1KB to 16MB. We repeated the experiment in networks with 0.2ms to 5.2ms latencies, and packed as many operations as possible into each vNFS compound. The results are shown (in logarithmic scale) in Figure 6, where *speedup ratio* is the ratio of the baseline's completion time to vNFS's completion time. Speedup ratios greater than one mean that vNFS performed better than the baseline; ratios less than one mean vNFS performed worse.

Because vNFS combined many small read and write operations into large compounds, it performed much better than the baseline when the file size was small. With a 1KB file size and 0.2ms network latency, vNFS is 19× faster than the baseline when reading (Figure 6(a)), and 5× faster when writing (Figure 6(b)). As the network latency increased to 5.2ms, vNFS's speedup ratio improved further to 103× for reading and 40× for writing. vNFS's speedup ratio was higher for reading than for writing because once vNFS was able to eliminate most network round trips, the NFS server's own storage became the next dominant bottleneck.

As the file size (the X-axis in Figure 6) was increased to 1MB and beyond, vNFS's compounding effect faded, and the performance of vNFS and the baseline became closer. However, in networks with 1.2–5.2ms latency, vNFS was still 1.1–1.7× faster than the baseline: although data operations were too large to be combined together, vNFS could still combine them with small metadata operations such as OPEN, CLOSE, and GETATTR. Combining metadata and data operations requires vNFS to split I/Os below 1MB due to the 1MB RPC size limit (see Section 4). When a large I/O is split into pieces, the last one may be small; this phenomenon made vNFS around 10% slower when reading 4MB and 8MB files in the 0.2ms-latency network. However, this is not a problem in most cases because that last small piece is likely to be combined into later compounds. This is why vNFS performed the same as the baseline with even larger file sizes (e.g., 16MB) in the 0.2ms-latency network. This negative effect of vNFS's splitting was unnoticeable for writing because writing was bottlenecked by the NFS server's storage. Note that the baseline (the in-kernel NFS client) splits I/Os strictly at the 1MB size, although it also adds a few trivial NFS operations such as PUTFH (see Figure 1) in its compounds, meaning that the baseline's RPC size is actually larger than 1MB.

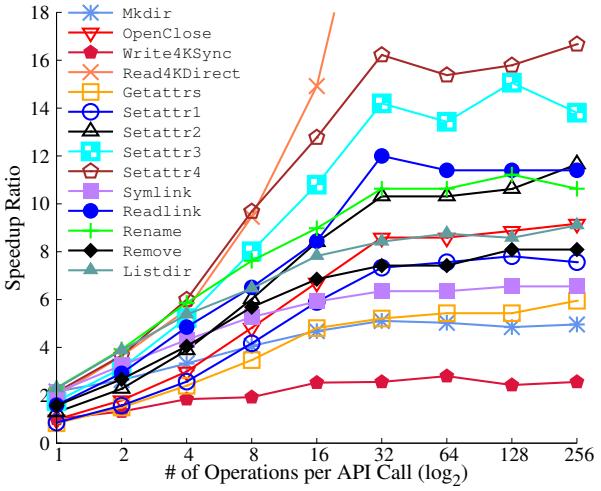


Figure 7: vNFS's speedup ratio relative to the baseline under different degrees of compounding. The X-axis is $\log_2$. The network latency is 0.2ms. Write4KSync writes 4KB data to files opened with `O_SYNC`; Read4KDirect reads 4KB data from files opened with `O_DIRECT`; `SetattrN` sets N files' attributes (mixes of mode, owner, timestamp, and size). The vector size of the baseline is actually the number of individual POSIX calls issued iteratively. The speedup ratio of Read4KDirect goes up to 46 at 256 operations per call; its curve is cut off here.

Compounding degree. The degree of compounding (i.e., the number of non-trivial NFS operations per compound) is a key factor determining how much vNFS can boost performance. The ideal is to perform a large number of file system operations at once, which is not always possible because applications may have critical paths that depend on only a single file. To study how the degree of compounding affects vNFS's performance, we compared vNFS with the baseline when calling the vNFS API functions with different numbers of operations in their vector arguments.

Figure 7 shows the speedup ratio of vNFS relative to the baseline as the number of operations per API call was increased from 1 to 256 in the 0.2ms-latency network. Even with one operation per call, vNFS outperformed the baseline for all API functions except two, because vNFS could still save round trips for single-file calls. For example, the baseline used three RPCs to rename a file: one LOOKUP for the source directory, another LOOKUP for the destination directory, and one RENAME; vNFS, however, used only one compound RPC combining all three operations. `Getattrs` and `Setattr1` are the two exceptions where vNFS performed slightly worse (17% and 14% respectively) than the baseline. This is because each of these two calls needs only a single NFS operation; so vNFS could not combine anything yet incurred the overhead of performing RPCs in user space.

When there was more than one operation per API call, compounding became effective and vNFS significantly outperformed the baseline for all calls; note that the Y axis of Figure 7 is in logarithmic scale. All calls except `Write4KSync` (bottlenecked by the server's storage stack) were more than 4× faster than the baseline when multiple operations were compounded. Note that `vsetattrs` can set multiple attributes at once, whereas the baseline sets one attribute at a time. We observe in Figure 7 that the speedup ratio of setting more attributes (e.g., `Setattr4`) at once was always higher than that of setting fewer (e.g., `Setattr3`).

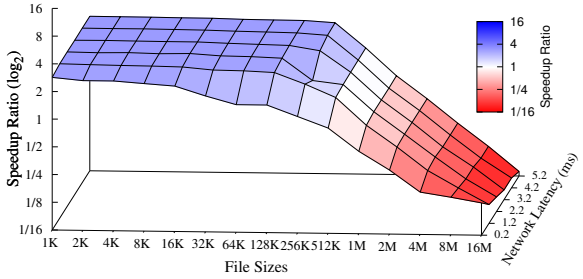


Figure 8: The speedup ratio of vNFS over the baseline (in logarithmic scale) when repeatedly opening, reading, and closing a single file, whose size is shown on the X-axis. vNFS is faster than (blue), equal to (white), and slower than (red) the baseline when the speedup ratio is larger than, equal to, and smaller than 1, respectively. Our vNFS prototype does not have a cache yet, whereas the baseline does. The Z-axis is in logarithmic scale; the higher the better.

In our experiments with slower networks (omitted for brevity), vNFS's speedups relative to the baseline were even higher than in the 0.2ms-latency network: up to two orders of magnitude faster.

Caching. Our vNFS prototype does not yet support caching. In contrast, the baseline (in-kernel NFS client) caches both metadata and data. To study the cache's performance impact, we compared vNFS and the baseline when repeatedly opening, reading, and closing a single file whose size varied from 1KB to 16MB. Figure 8 shows the results, where a speedup ratio larger than one means vNFS outperformed the baseline; and a speedup ratio less than one means vNFS performed worse.

The baseline served all reads except the first from its cache, but it was slower than vNFS (which did not cache) when the file size was 256KB or smaller. This is because three RPCs per read are still required to maintain close-to-open semantics: an OPEN, a GETATTR (for cache revalidation), and a CLOSE. In comparison, vNFS used only one compound RPC, combining the OPEN, READ (uncached), and CLOSE. The savings from compounding more than compensated for vNFS's lack of a cache. For a 512KB file size, vNFS was still faster than the baseline except in the 0.2ms-latency network. For 1MB and larger files, vNFS was worse than the baseline because read operations dominated: the baseline served all reads from its client-side cache whereas vNFS sent all reads to the server without the benefit of caching.

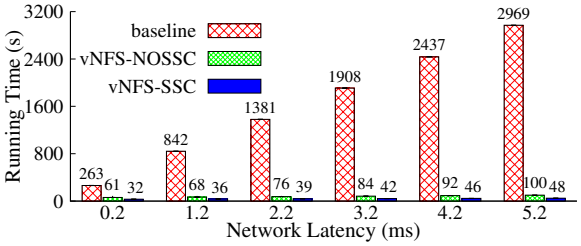


Figure 9: Running time to copy (`cp -r`) the entire Linux source tree. The lower the better. vNFS runs much faster than the baseline both with and without Server Side Copy (SSC).

5.3 Macro-workloads

To evaluate vNFS using realistic applications, we modified `cp`, `ls`, and `rm` from GNU Coreutils, Filebench [15, 40], and nghttp2 [31] to use the vNFS API; we also implemented an equivalent of GNU `tar` using vNFS.

GNU Coreutils. Porting `cp` and `rm` to vNFS was easy. For `cp`, we added 170 lines of code and deleted 16; for `rm`, we added 21 and deleted 1. Copying files can be trivially achieved using `vsscopy`, `vgetattrs`, and `vsetattrs`. Recursively copying directories requires calling `vlistdir` on the directories and then invoking `vsscopy` for plain files, `vmkdir` for directories, and `vsymlink` for symbolic links—all of which is done in `vlistdir`'s callback function. We tested our modified `cp` with `diff -qr` to ensure that the copied files and directories were exactly the same as the source. Removing files and directories recursively in `rm` was similar, except that we used `vremove` instead of `vsscopy`.

Porting `ls` was more complex because batching is difficult when listing directories recursively in a particular order (e.g., by file size). We could not use the recursive mode of `vlistdir` because the NFS READDIR operation does not follow any specific order when reading directory entries, and the whole directory tree may be too large to fit in memory. Instead, vNFS maintains a list of all directories to read in the proper order as specified by the `ls` options, and repeatedly calls `vlistdir` (not recursively) on directories at the head of the list until it is empty. Note that (1) a directory is removed from the list only after all its children have been read; and (2) sub-directories should be sorted and then inserted immediately after their parent to maintain the proper order in the list. We added 392 lines of code and deleted 203 to port `ls` to vNFS. We verified that our port is correct by comparing the outputs of our `ls` with the vanilla version.

We used the ported Coreutils programs to copy, list, and remove an entire Linux-4.6.3 source tree: it contains 53,640 files with an average size of 11.6KB, 3,604 directories with average 17 children per directory, and 23 symbolic links. The large number of files and directories thoroughly exercises vNFS and demonstrates the performance impact of compounding.

Figure 9 shows the results of copying the entire Linux source tree; vNFS outperformed the baseline in all cases. vNFS uses either `vsscopy` or `vcopy` depending on whether Server Side Copy (SSC) is enabled. However, the baseline cannot use SSC because it is not yet supported by the in-kernel NFS client. For the same workload of copying the Linux source tree, vNFS used merely 4,447 compounding RPCs whereas the baseline used as many as 506,697: two OPENs, two CLOSEs, one READ, one WRITE, and one SETATTR for each of the 53,640 files; 60,873 ACCESSes; 62,327 GETATTRs; and 8,017 other operations such as READDIR and CREATE. vNFS-NOSSC saved more than 99% of RPCs compared to the baseline, with each vNFS compounding RPC containing an average of 250 operations. Therefore, even with only a 0.2ms network latency, vNFS-NOSSC is still more than 4× faster than the baseline. The speedup ratio increases to 30× with a 5.2ms network latency.

When Server Side Copy (SSC) was enabled, vNFS ran even faster, and vNFS-SSC reduced the running time of vNFS-NOSSC by half. The further speedup of SSC is only moderate because the files are small and our network bandwidth (10GbE) is large. The speedup ratio of vNFS-SSC to the baseline is 8–60× in networks with 0.2–5.2ms latency. Even when the baseline adds SSC support in the future, vNFS would still outperform it because this workload's bottleneck is the large number of small metadata operations, not data-copying operations.

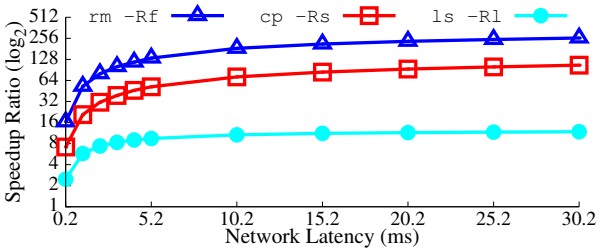


Figure 10: vNFS's speedup relative to the baseline when symbolically copying (`cp -Rs`), listing (`ls -Rl`), and removing (`rm -Rf`) the Linux source tree. The Y-axis is logarithmic.

With the `-Rs` options, `cp` copies an entire directory tree by creating symbolic links to the source directory. Figure 10 shows speedups for symlinking, recursively listing (`ls -Rl`), and removing (`rm -Rf`) the Linux source tree. When recursively listing the Linux tree, `ls`-baseline used 10,849 RPCs including 3,678 READDIRs, 3,570 ACCESSes, and 3,570 GETATTRs. Note that the in-kernel NFS client did not issue a separate GETATTR for each directory entry although the vanilla `ls` program called `stat` for each entry listed. This is because the in-kernel NFS client pre-fetches the attributes using `readdirs` and serves the `stat` calls from the local client's dentry metadata cache. This optimization enables `ls`-baseline to finish the benchmark in just 5 sec-

onds in the 0.2ms-latency network. However, with our vectorized API, `ls`-vNFS did even better and finished in 2 seconds, using only 856 RPCs. Moreover, vNFS scales much better than the baseline. When the latency increased from 0.2 to 30.2ms, vNFS's running time rose to only 28 seconds whereas the baseline increased to 336 seconds. `ls`-vNFS is 10× faster than `ls`-baseline in high-latency (>5.2ms) networks.

For symbolic copying and removing (Figure 10), vNFS was 7× and 18× faster than the baseline in the 0.2ms-latency network, respectively. This is because the baseline always operated on one file at a time, whereas vNFS could copy or remove more than 200 files at once. Compared to the baseline, vNFS improved `cp` by 52× and `rm` by 133× in the 5.2ms-latency network; with 30.2ms latency the speedup ratios became 106× for `cp`, and 263× for `rm`. For both removing and symbolic copying, vNFS ran faster in the 30.2ms-latency network (25 and 15 seconds, respectively) than the baseline did with 0.2ms latency (38s and 55s, respectively), showing that compounds can indeed help NFSv4 realize its design goal of being WAN-friendly [29].

`tar`. Because the I/O code in GNU `tar` is closely coupled to other code, we implemented a vNFS equivalent using `libarchive`, in which the I/O code is clearly separated. The `libarchive` library supports many archiving and compression algorithms; it is also used by FreeBSD `bsdtar`. Our implementation needed only 248 lines of C++ code for `tar` and 210 for `untar`, both including error-handling code.

When archiving a directory, we use the `vlistdir` API to traverse the tree and add sub-directories into the archive. We gather the listed files and symlinks into arrays, then read their contents using `vread` and `vreadlink`, and finally compress and write the contents into the archive. During extraction, we read the archive in 1MB (RPC size limit) chunks and then use `libarchive` to extract and decompress objects and their contents, which are then passed in batches to `vmkdir`, `vwrite`, or `vsymlink`. We always create parent directories before their children. We ensured that our implementation is correct by feeding our `tar`'s output into our `untar` and comparing the extracted files with the original input files. We also tested for cross-compatibility with other tar implementations including `bsdtar` and GNU `tar`.

We used our `tar` to archive and `untar` to extract a Linux 4.6.3 source tree. Archiving read 53,640 small files and wrote a large archive: 636MB uncompressed, and 86MB with the `xz` option (default compression used by `kernel.org`). Extracting reversed the process. There were also metadata operations on 23 symbolic links and 3,604 directories. Figure 11 shows the `tar`/`untar` results, compared to

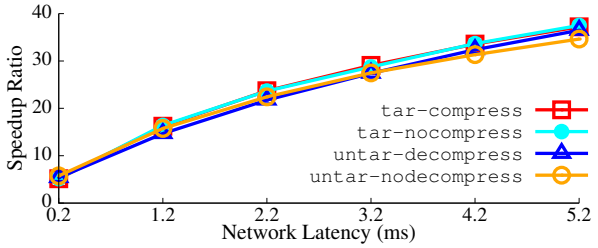


Figure 11: Speedup ratios of vNFS relative to the baseline (`bsdtar`) when archiving and extracting the Linux-4.6.3 source tree, with and without xz compression.

`bsdtar` (running on the in-kernel client) as the baseline. For `tar-nocompress` in the 0.2ms-latency network, vNFS was more than 5× faster than the baseline because the baseline used 446,965 RPCs whereas vNFS used only 2,144 due to compounding. This large reduction made vNFS 37× faster when the network latency increased to 5.2ms. In terms of running time, vNFS used 69 seconds to archive the entire Linux source tree in the 5.2ms-latency network, whereas the baseline, even in the faster 0.2ms-latency network, still used as much as 192 seconds. For `untar-nodecompress`, vNFS is also 5–36× faster, depending on the network latency.

Figure 11 also includes the results when `xz` compression was enabled. Although compression reduced the size of the archive file by 86% (from 636MB to 86MB) and thus saved 86% of the I/Os to the archive file, it had a negligible performance impact (less than 0.5%) because the most time-consuming operations were for small I/Os, not large ones. This test shows that workloads with mixed I/O sizes are slow if there are many small I/Os, each incurring a network round trip; vNFS can significantly improve such workloads by compounding those small I/Os.

Filebench. We have ported Filebench to vNFS and added vectorized *flowops* to the Filebench workload modeling language (WML) [46]. We added 759 lines of C code to Filebench, and removed 37. We converted Filebench's File-Server, NFS-Server, and Varmail workloads to equivalent versions using the new flowops: for example, we replaced N adjacent sets of `openfile`, `readwholefile`, and `closefile` (i.e., $3 \times N$ old flowops) with a single `vreadfile` (one new flowop), which internally uses our `vread` API that can open, read, and close N files in one call.

The Filebench NFS-Server workload emulates the SPEC SFS benchmark [36]. It contains one thread performing four sets of operations: (1) open, entirely read, and close three files; (2) read a file, create a file, and delete a file; (3) append to an existing file; and (4) read a file's attributes. The File-Server workload emulates 50 users accessing their home directories and spawns one thread per user to perform operations similar to the NFS-Server workload. The Varmail workload mimics

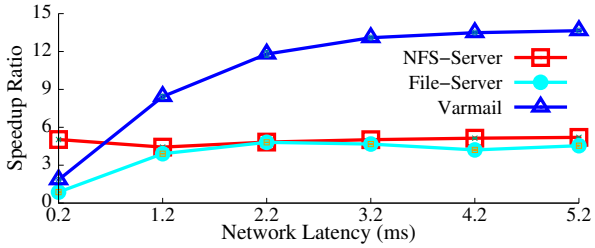


Figure 12: vNFS's speedup ratios for Filebench workloads.

a UNIX-style email server operating on a `/var/mail` directory, saving each message as a file; it has 16 threads, each performing create-append-sync, read-append-sync, read, and delete operations on 10,000 16KB files.

Figure 12 shows the results of the Filebench workloads, comparing vNFS to the baseline. For the NFS-Server workload, vNFS was 5× faster than the baseline in the 0.2ms-latency network because vNFS combined multiple small reads and their enclosing opens and closes into a single compound. vNFS was also more efficient (and more reliable) when appending files since it does not need a separate GETATTR to read the file size (see Section 3 [¶`vread/vwrite`]). This single-threaded NFS-Server workload is light, and its only bottleneck is the delay of network round trips. With compounding, vNFS can save network round trips; the amount of savings depends on the compounding degree (the number of non-trivial NFS operations per compound). This workload has a compounding degree of around 5, and thus we observed a consistent 5× speedup regardless of the network latency.

As shown in Figure 12, vNFS's speedup ratio in the File-Server workload is about the same as the NFS-Server one, except in the 0.2ms-latency network. This is because these two workloads have similar file-system operations and thus similar compounding degrees. However, in the 0.2ms-latency network, vNFS was 13% slower (i.e., a speedup ratio of 0.87) than the baseline. This is caused by two factors: (1) the File-Server workload has as many as 50 threads and generates a heavy I/O load to the NFS server's storage stack, which became the bottleneck; (2) without a cache, vNFS sent all read requests to the overloaded server whereas the in-kernel client's cache absorbed more than 99% of reads. As the network latency increased, the load on the NFS server became lighter and vNFS became faster thanks to saving round trips, which more than compensated for the lack of caching in our current prototype.

Because the Varmail workload is also multi-threaded, its speedup ratio curve in Figure 12 has a trend similar to that of the File-Server workload. However, vNFS's speedup ratio in the Varmail workload plateaued at the higher value of 14× because its compounding degree is higher than the File-Server workload.

Network Latency (ms)	0.2	1.2	2.2	3.2	4.2	5.2
Speedup Ratio	3.5	6.5	7.1	8.7	9.8	9.9

Table 2: vNFS speedup ratio relative to the baseline when requesting a set of objects with PUSH enabled in nghttp2.

HTTP/2 server. Similar to the concept of NFSv4 compounds, HTTP/2 improves on HTTP/1.*x* by transferring multiple objects in one TCP connection. HTTP/2 also added a PUSH feature that allows an HTTP/2 server to proactively push related Web objects to clients [5, Section 8.2]. For example, upon receiving an HTTP/2 request for `index.html`, the server can proactively send the client other Web objects (such as Javascript, CSS, and image files) embedded inside that `index.html` file, instead of waiting for it to request them later. PUSH can reduce a Web site's loading time for end users. It also allows Web servers to read many related files together, enabling efficient processing by vNFS.

We ported *nghttp2* [31], an HTTP/2 library and toolset containing an HTTP/2 server and client, to vNFS. Our port added 543 lines of C++ code and deleted 108.

The HTTP Archive [20] shows that, on average, an HTTP URL is 2,480KB and contains ten 5.5KB HTML files, 23 20KB Javascript files, seven 7.5KB CSS files, and 56 28KB image files. We created a set of files with those characteristics, hosted them with our modified nghttp2 server, and measured the time needed to process a PUSH-enabled request to read the file set. Table 2 shows the speedup ratio of vNFS relative to the baseline, which runs vanilla nghttp2 and the in-kernel NFS client. vNFS needed only four NFS compounds for all 96 files: one `vgetattrs` call and three `vreads`. In contrast, the baseline used 309 RPCs including one OPEN, READ, and CLOSE for each file. The reduced network round trips made vNFS 3.5× faster in the 0.2ms-latency network and 9.9× faster with the 5.2ms latency.

6 Related Work

Improving NFS performance. NFS is more than 30 years old, and has continuously evolved to improve performance. Following the initial NFSv2 [38], NFSv3 added asynchronous COMMITs to improve write performance, and READDIRPLUS to speed up directory listing [7]. NFSv4.0 [35] added more performance features including compounding procedures that batch multiple operations in one RPC, and delegations that enable the client cache to be used without lengthy revalidation. To improve performance further with more parallelism, NFSv4.1 [34] added pNFS [18] to separate data and meta-data servers so that the different request types can be served in parallel. The upcoming NFSv4.2 has yet more performance improvements such as I/O hints, *Application Data Blocks*, and Server Side Copy [17].

In addition to improvements in the protocols, other researchers also improved NFS's performance: Duchamp

found it inefficient to look up NFSv2 paths one component at a time, and reduced client latency and server load by optimistically looking up whole paths in a single RPC [13]. Juszczak improved the write performance of an NFS server by gathering many small writes into fewer larger ones [23]. Ellard and Seltzer improved read performance with read-ahead and stride-read algorithms [14]. Batsakis et al. [4] developed a holistic framework that adaptively schedules asynchronous operations to improve NFS's performance as perceived by applications. Our vNFS uses a different approach, improving performance by making NFSv4's compounding procedures easily accessible to programmers.

I/O compounding. Compounding, also called batching and coalescing, is a popular technique to improve throughput and amortize cost by combining many small I/Os into fewer larger ones. Disk I/O schedulers coalesce adjacent I/Os to reduce disk seeks [3] and boost throughput. Purohit et al. [32] proposed Compound System Calls (Cosy) to amortize the cost of context switches and to reduce data movement across the user-kernel boundary. These compounding techniques are all hidden behind the POSIX file-system API, which cannot convey the required high-level semantics [8]. The Batch-Aware Distributed File System (BAD-FS) [6] demonstrated the benefits of using high-level semantics to explicitly control the batching of I/O-intensive scientific workloads. *Dynamic sets* [37] took advantage of the fact that files can be processed in any order in many bulk file-system operations (e.g., `grep foo *.c`). Using a set-based API, distributed file system clients can pre-fetch a set of files in the optimal order and pace so that computation and I/O are overlapped and the overall latency is minimized. However, dynamic sets did not reduce the number of network round trips. SeMiNAS [9] uses NFSv4 compounds (only) in its security middleware to reduce the security overhead. To the best of our knowledge, vNFS is the first attempt to use an overt-compounding API to leverage NFSv4's compounding procedures.

Vectorized APIs. To achieve high throughput, Vilayanur et al. [43] proposed `readx` and `writex` to operate at a vector of offsets so that the I/Os could be processed in parallel. However, these operations were limited to a single file, helping only large files, whereas our `vread`/`vwrite` can access many files at once, helping with both large and small files.

Vasudevan et al. [41] envisioned the Vector OS (VOS), which offered several vectorized system calls, such as `vec_open()`, `vec_read()`, etc. While VOS is promising, it has not yet been fully implemented. In their prototype, they succeeded in delivering millions of IOPS in a distributed key-value (KV) store backed by fast NVM [42]. However, they implemented a key-value API, not a file-system API, and their vectorized KV store focuses on serving parallel I/Os on NVM, whereas vNFS focuses on saving network round trips by using NFSv4 compound procedures. The vectorized key-value store and vNFS are different but complementary.

Our vNFS API is also different from other vectorized APIs [41,43] in three aspects: (1) `vread`/`vwrite` supports automatic file opening and closing; (2) `vsscopy` takes advantage of the NFS-specific Server Side Copy feature; and (3) to remain NFSv4-compliant, vNFS's vectorized operations are executed in order, in contrast to the out-of-order execution of `lio_listio(3)` [26], `vec_read()` [41], and `readx` [43].

7 Conclusions

We designed and implemented vNFS, a file-system library that maximizes NFS performance. vNFS uses a vectorized high-level API to leverage standard NFSv4 compounds, which have the potential to reduce network round trips but were underused due to the low-level and serialized nature of the POSIX API. vNFS makes maximal use of compounds by enabling applications to operate on many file-system objects in a single RPC. vNFS complies with the NFSv4.1 protocol and has standard failure semantics. To help port applications to the vectorized API, vNFS provides a superset of POSIX file-system operations, and its library can be used for non-NFS file systems as well. We found it generally easy to port applications including `cp`, `ls` and `rm` from GNU Coreutils; `bsdtar`; Filebench; and nghttp2.

Micro-benchmarks demonstrated that—compared to the in-kernel NFS client—vNFS significantly helps workloads with many small I/Os and metadata operations even in fast networks, and performs comparably for large I/Os or with low compounding degrees. Macro-benchmarks show that vNFS sped up the ported applications by up to two orders of magnitude. Our source code is available at *https://github.com/sbu-fsl/txn-compound*.

Limitations and future work. Currently vNFS does not include a cache; an implementation is underway. To simplify error handling, we plan to support optionally executing a compound as an atomic transaction. Finally, compounded operations are processed sequentially by current NFS servers; we plan to execute them in parallel with careful interoperation with transactional semantics.

Acknowledgments

We thank the anonymous FAST reviewers and our shepherd Keith Smith for their valuable comments. We also thank Geoff Kuenning for his meticulous and insightful review comments. This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; NSF awards CNS-1251137, CNS-1302246, CNS-1305360, and CNS-1622832; and ONR award 12055763.

References

[1] Antoine Potten. Ant renamer, 2016. *http://www.antp.be/software/renamer*.

[2] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 19. ACM, 2016.

[3] J. Axboe. CFQ IO scheduler, 2007. *http://mirror.linux.org.au/pub/linux.conf.au/2007/video/talks/123.ogg*.

[4] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey. CA-NFS: A congestion-aware network file system. *ACM Transaction on Storage*, 5(4), 2009.

[5] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, Internet Engineering Task Force, May 2015.

[6] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in the batch-aware distributed file system. In *NSDI*, pages 365–378, 2004.

[7] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813, Network Working Group, June 1995.

[8] M. Chen, D. Hildebrand, G. Kuenning, S. Shankaranarayana, B. Singh, and E. Zadok. Newer is sometimes better: An evaluation of NFSv4.1. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2015)*, Portland, OR, June 2015. ACM.

[9] M. Chen, A. Vasudevan, K. Wang, and E. Zadok. SeMiNAS: A secure middleware for wide-area network-attached storage. In *Proceedings of the 9th ACM International Systems and Storage Conference (ACM SYSTOR '16)*, Haifa, Israel, June 2016. ACM.

[10] Stuart Cheshire. TCP performance problems caused by interaction between Nagle's algorithm and delayed ACK, May 2005.

[11] Philippe Deniel. GANESHA, a multi-usage with large cache NFSv4 server. *www.usenix.org/events/fast07/wips/deniel.pdf*, 2007.

[12] Philippe Deniel, Thomas Leibovici, and Jacques-Charles Lafoucrière. GANESHA, a multi-usage with large cache NFSv4 server. In *Linux Symposium*, page 113, 2007.

[13] D. Duchamp. Optimistic lookup of whole NFS paths in a single operation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 143–170, Boston, MA, June 1994.

[14] D. Ellard and M. Seltzer. NFS tricks and benchmarking traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, San Antonio, TX, June 2003. USENIX Association.

[15] Filebench, 2016. *https://github.com/filebench/filebench/wiki*.

[16] S. Han, S. Marshall, B. Chun, and S. Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[17] T. Haynes. NFS version 4 minor version 2 protocol. RFC draft, Network Working Group, September 2015. *https://tools.ietf.org/html/draft-ietf-nfsv4-minorversion2-39*.

[18] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. In *Proceedings of MSST*, Monterey, CA, 2005. IEEE.

[19] D. Howells. FS-Cache: A Network Filesystem Caching Facility. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 427–440, Ottawa, Canada, July 2006. Linux Symposium.

[20] HTTP Archive. URL statistics, September 2016. *http://httparchive.org/trends.php*.

[21] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, Network Working Group, January 1984.

[22] Jason Fitzpatrick. Bulk rename utility, 2016. *http://www.bulkrenameutility.co.uk/Main_Intro.php*.

[23] Chet Juszczak. Improving the write performance of an NFS server. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC'94, San Francisco, California, 1994. USENIX Association.

[24] Kim Jensen. AdvancedRenamer, 2016. *https://www.advancedrenamer.com/*.

[25] Chuck Lever. Close-to-open cache consistency in the Linux NFS client. *http://goo.gl/o9i0MM*.

[26] Linux Programmer's Manual. *lio_listio*, September 2016. *http://man7.org/linux/man-pages/man3/lio_listio.3.html*.

[27] Linux man pages. ftw(3) - file tree walk. *http://linux.die.net/man/3/ftw*.

[28] Linux man pages. open(2) - open and possibly create a file or device. *http://linux.die.net/man/2/open*.

[29] Alex McDonald. The background to NFSv4.1. *;login: The USENIX Magazine*, 37(1):28–35, February 2012.

[30] NFS-Ganesha, 2016. *http://nfs-ganesha.github.io/*.

[31] nghttp2. nghttp2: HTTP/2 C library, September 2016. *http://nghttp2.org*.

[32] A. Purohit, C. Wright, J. Spadavecchia, and E. Zadok. Cosy: Develop in user-land, run in kernel mode. In *Proceedings of the 2003 ACM Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 109–114, Lihue, Hawaii, May 2003. USENIX Association.

[33] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's time for low latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[34] S. Shepler and M. Eisler and D. Noveck. NFS Version 4 Minor Version 1 Protocol. RFC 5661, Network Working Group, January 2010.

[35] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3530, Network Working Group, April 2003.

[36] SPEC. SPEC SFS97_R1 V3.0. *www.spec.org/sfs97r1*, September 2001.

[37] David C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '97, 1997.

[38] Sun Microsystems. NFS: Network file system protocol specification. RFC 1094, Network Working Group, March 1989.

[39] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok. Terra incognita: On the practicality of user-space file systems. In *HotStorage '15: Proceedings of the 7th USENIX Workshop on Hot Topics in Storage*, Santa Clara, CA, July 2015.

[40] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[41] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. The case for VOS: The vector operating system. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 31–31, Berkeley, CA, USA, 2011. USENIX Association.

[42] 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 3rd ACM Symposium on Cloud Computing*, SoCC '12, 2012.

[43] M. Vilayannur, S. Lang, R. Ross, R. Klundt, and L. Ward. Extending the POSIX I/O interface: A parallel file system perspective. Technical Report ANL/MCS-TM-302, Argonne National Laboratory, October 2008.

[44] M. Mitchell Waldrop. The chips are down for Moore's law. *Nature*, 530(7589):144–147, 2016.

[45] Werner Beroux. Rename-It!, 2016. *https://github.com/wernight/renameit*.

[46] Filebench Workload Model Language (WML), 2016. *https://github.com/filebench/filebench/wiki/Workload-Model-Language*.

On the Accuracy and Scalability of Intensive I/O Workload Replay

Alireza Haghdoost*, Weiping He*, Jerry Fredin†, and David H.C. Du*

*University of Minnesota Twin-Cities, †NetApp Inc.

Abstract

We introduce a replay tool that can be used to replay captured I/O workloads for performance evaluation of high-performance storage systems. We study several sources in the stock operating system that introduce the uncertainty of replaying a workload. Based on the remedies of these findings, we design and develop a new replay tool called *hfplayer* that can more accurately replay intensive block I/O workloads in a similar unscaled environment. However, to replay a given workload trace in a scaled environment, the dependency between I/O requests becomes crucial. Therefore, we propose a heuristic way of speculating I/O dependencies in a block I/O trace. Using the generated dependency graph, *hfplayer* is capable of replaying the I/O workload in a scaled environment. We evaluate *hfplayer* with a wide range of workloads using several accuracy metrics and find that it produces better accuracy when compared with two exiting available replay tools.

1. Introduction

Performance evaluation of a storage system with realistic workloads has always been a desire of storage systems developers. Trace-driven evaluation is a well-known practice to accomplish this goal. It does not require installation of a real system to run applications and does not expose a production systems to potential downtime risk caused by performance evaluation experiments. However, the lack of an accurate trace replay tool makes it less appealing and draws some skepticism of using trace-driven methods for performance evaluation of block storage devices and systems [19, 23].

An I/O trace typically includes the timestamps of when each I/O request is issued and completed. These timestamps can be used to replay the workload on a similar unscaled environment. However, the block I/O trace does not usually include any information about the dependencies between I/O requests. Therefore, these timestamps cannot be directly used in a workload replay on a scaled environment where the issue time of a latter I/O request may be determined by the completion time of a former request in a dependency chain. However, a latter I/O request may be issued earlier before the completion time of a former I/O request if there is no dependency between the two. Some examples of a scaled environment can mean a server speed is increased, the number of disk drives is doubled, or a faster type of drives is considered.

The I/O dependency information is available only in the file system or application layers. However, the intrinsic software overhead and high degree of parallelism that embedded in these layers reduce the capability of the workload replay to stress-test a modern block storage system when replaying I/O traces at file system or application layer. For example, Weiss et al. [24] demonstrates Can a scalable file system replay tool that can barely hit 140 IOPS, while modern storage systems are capable of driving intensive I/O workloads with hundreds of thousands of IOPS and a few milliseconds response-time [4].

Typical block I/O replay tools ignore I/O dependencies between I/O requests and replay them as fast as possible (AFAP) [15]. The AFAP approach cannot accurately replay a workload since it overlooks the intrinsic computation and wait time in the original application. Therefore, the characteristics of replayed workload may be different from that of an original application in terms of throughput, response-time and request ordering.

We believe a more accurate replay tool for scaled environments should try to mimic the behavior of the real application on the scaled environments and respect the existing dependencies in the I/O trace. This is possible by speculating the dependencies of I/O requests and trying to propagate I/O-related performance gains along the dependency chains. However, it is challenging to discover I/O dependencies simply based on a given block I/O workload trace without accessing the application source code.

In this work, we propose *hfplayer*, a replay tool that tries to infer potential I/O dependencies from the block I/O traces and replay I/O intensive workloads with more accuracy in both scaled and unscaled replay modes. In the scaled mode, the arrival of new requests depends on the completion of the previous requests, while in the unscaled mode each request is issued independently at a scheduled time [14].

The goal of this replay tool is to ensure that replayed I/O requests arrive the storage device (e.g., SAN controller) at the right time and order. We develop methods to ensure that the right number of I/O requests being issued from the user level and these requests traverse all the OS layers (the entire path from user space to device controller) with minimal interference on the workload throughput, I/O response time and ordering.

The intended use of *hfplayer* is for performance evaluation, debugging and validating different block storage devices and systems with realistic workloads. For the set of our experiments, it replays realistic I/O workload on both scaled and unscaled storage systems with less than 10% average error.

Our main contributions in the paper are summarized as follows: 1) Uncover and propose remedies to Linux I/O stack limitations and its non-deterministic I/O behaviors which are sources of measurement uncertainty. 2) Design and implement a workload replay engine which can more accurately replay a block I/O workload that has been captured in a similar unscaled environment. 3) Design and implement a scalable workload replay engine which can speculate I/O request dependencies and replay a realistic workload in a scaled environment. Our workload replay tool is open-source and available for download at: https://github.com/umn-cris/hfplayer.

The rest of this paper is structured as follows. In the next section, we describe several sources of non-deterministic behaviors for block I/O workload replay in the stock operating system. We also describe our approaches to remedy or work around these limitations. In Section 3 we discuss the I/O request dependencies and how we construct a dependency graph for a given workload by speculating I/O request dependencies. We also describe our approach of replaying the I/O workload considering the influences of the target storage in a scaled environment using the scalable replay engine of *hfplayer*. Then we evaluate the replay accuracy of *hfplayer* on both unscaled and scaled storages and compare the results with existing tools in Section 5. We refer and describe the relevant literature of this paper in Section 6. Finally, we summarize and offer some conclusions in Section 7.

2. Sources of Workload Replay Uncertainty

In this section, we introduce several key limitations to faithfully replaying a block I/O workload in the Linux I/O stack. One solution to potentially limiting OS impacts the workload replay accuracy is to embed the I/O engine of the workload replay tool in the kernel space [5]. This would reduce forced preemption of replay engine threads and eliminates the cost of user to kernel space context switch in replaying an I/O request. However, this approach limits the portability of the developed engine to a single OS platform and even to a specific kernel version. Therefore, we focus on developing a replay tool with a user space engine which is capable of working with the standard system calls on most OS platforms. In this work, we focus on Linux and propose a method to significantly reduce the context switch overhead. We will introduce the integration with IBM's AIX OS in the future work. Moreover, we believe user space I/O engine development is aligned with the emerging efforts in the industry to develop user space APIs like Intel SPDK for ultra high-speed storage devices [2].

User-space replay threads can submit I/O requests using synchronous or asynchronous system calls. Replay tools that are implemented with the synchronous method like *Buttress* [6] and *blkreplay* [15] need a large number of worker threads running in parallel to achieve a high replay throughput. Therefore, they may expose more inter-thread coordination overheads compared with the asynchronous method. These overheads are known as a major source of uncertainty in the workload replay [13]. As we show in Section 5, these limitations severely impact the accuracy of replaying I/O intensive workloads. Therefore, *hfplayer* replay engine is based on the asynchronous method and exclusively uses *libaio* which is Linux kernel support for asynchronous I/O operations as well as *NO-OP* scheduler without its *merge* functionality. We have identified that Linux kernel is not able to fully disable the I/O merge functionality in certain situations. We have proposed a patch to fix this bug and the patch has been merged into the mainstream branch since the kernel v3.13 [3].

Accurate I/O replay timing is another challenging issue which heavily depends on the timing predictability of the I/O stack. In an ideal situation, it should cost each I/O request a fixed amount of time to travel from the replay engine to the storage device. Therefore, if there is an inter-arrival gap of n milliseconds between two consecutive I/O requests in the given workload trace, the replay thread just needs to wait n milliseconds after issuing the first I/O request to issue the second request and expects both requests to arrive at the device with an accurate time interval of n milliseconds. However, our experiments show that the I/O stack travel time is quite unpredictable and thus such a timing accuracy is impossible if we do not carefully tune and watch the I/O queuing mechanism in the kernel. In order to work around these limitations, we have practiced the following four techniques to improve the workload replay accuracy.

2.1. I/O Stack Queue Sizing

In general there are two queuing layers in the asynchronous I/O submission path of Linux kernel. The *io_submit* system call first pushes a block I/O request (or *bio*) into the tail of the block layer queue, then it pulls another *bio* from the head of the queue, transforms it into a SCSI command and pushes it into the SCSI layer queue. Finally, it returns the success code to the user space, while the SCSI Initiator processes the commands in the SCSI layer queue using a kernel worker thread. These two queues usually do not have enough room for all new requests when the I/O stack is overloaded. In this case, the execution time of the system call becomes unpredictable.

While the block layer request queue size is tunable, the SCSI layer queue size is limited to the maximum supported TCQ tags by the SCSI layer. Usually a system call

puts the *bio* request into a waiting list if it cannot find an empty queue tag and schedules a kernel worker thread to push it later. This schedule is expensive and significantly increases I/O submission unpredictability. Moreover, it changes the delivery order of I/O requests in the kernel space even if the replay tool thread submits these requests in order. Therefore, internal I/O queues in the kernel should be sized according to the HBA capability to reduce queue insertion failures. In our test environment, while our HBA can handle 2048 in-flight requests, the default block layer queue size is set to 128 and the default SCSI layer queue size is 32.

Obviously, adjusting the I/O stack queue sizes below the HBA capability reduces the potential I/O throughput and impacts the replay accuracy to reproduce high-throughput workload. On the other hand, enlarging the queue sizes more than HBA capability without any control on the number of in-flight requests overloads the HBA and results in multiple retries to sink I/O requests from the block layer queue into the SCSI command queue. Therefore, we set the internal queue size to the maximum size supported by HBA hardware and dynamically monitor in-flight requests in the replay engine to make sure it does not overload the internal queues.

2.2. System-Call Isolation

We have identified that forced preemption is another factor making the execution time of I/O submission system call unpredictable. The system call execution thread can be preempted by other kernel processes with higher priorities like interrupt service routines. Consequently, it will impact the timing and ordering accuracy of a typical multi-threaded I/O workload replay tool. While it is possible to isolate regular interrupt handlers from system call threads using IRQ and CPU affinities, it is hard to avoid collisions of the system call threads with non-maskable timer interrupts. Moreover, scheduling replay tool threads with real-time scheduling priority is not a viable option since it has been discovered as another source of uncertainty for the workload replay [13]. Therefore, our best effort is to pin the replay tool threads to a CPU set and exclude all maskable interrupts from execution on that CPU set.

2.3. In-Flight I/O Control

After making an I/O submission system call more predictable with the aforementioned techniques, we need to work around potential scenarios that the I/O path unexpectedly takes more time to submit asynchronous I/O requests. When the submissions of a few I/O requests get delayed in a worker thread, it prevents the following I/O requests from being submitted on time. For example, assuming three consecutive I/O requests are scheduled to issue at time t, $t+10\mu s$ and $t+20\mu s$ by a worker thread. If the submission of the first request unexpectedly takes $30\mu s$, there is no need to wait any longer to issue the following two I/O requests since the scheduled issue times of these two requests are passed. In such a scenario, the replay tool issues several I/O requests in a very short amount of time and this burst of I/O requests forces the I/O stack to operate in a state that we call *Response-Time Trashing* state. In this state, an artificially large number of I/O requests are pushed into the I/O stack queues which severely impacts I/O response-time accuracy.

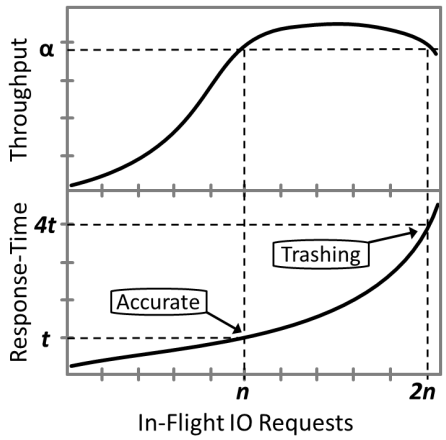


Figure 1: Response-Time Trashing

Figure 1 illustrates the *Response-Time Trashing* which occurs when the number of in-flight I/O requests is artificially high. For example, assuming we are planning to replay an I/O workload with a throughput of α and response-time of t. An accurate replay should reproduce the workload with the same throughput and response-time by keeping n I/O requests in-flight. However, unpredictable delays in the path cause a burst submission of an unexpectedly large number of I/O requests since their scheduled issue times have already passed. Therefore, in this example $2n$ requests are being queued in the stack instead of n requests. The throughput of I/O becomes saturated after a threshold number of queued requests is reached. Pushing more I/O requests to the internal queues only exponentially increases the response-time without any throughput improvement. Without an in-flight I/O control mechanism, the workload is being replayed with response-time of $4t$ instead of t since an artificially high number of requests are queued in the stack. Moreover, there is no reason to slow down or flush the queues since the workload is being replayed with the expected throughput of α. Note that some I/O workloads do not experience the dropping tail of I/O throughput by increasing the number of in-flight requests. These workloads usually saturate physical hardware resources of the SAN controller.

Limiting internal queue sizes is a conservative default approach to prevent I/O stack operating in the trashing state. This conservative approach bounds the maximum throughput intentionally to prevent typical applications

without in-flight I/O control from creating a burst submissions of I/O request and forcing the I/O stack to operate in the trashing state. Therefore, we need to keep the internal kernel queues open and meanwhile dynamically monitor and control the number of in-flight I/Os in the replay engine.

Depending on the information included in the workload trace, it is possible to estimate the number of in-flight I/O requests required to replay the workload. The number of in-flight I/O requests at time t can be calculated by counting the number of I/O requests with issue times smaller than t and completion times larger than t. Assume that n requests are counted as the number of in-flight requests at time t for a given trace file. During the replay process, we can throttle the I/O injection rate if more than n requests have been issued but not yet completed. This throttling creates a small pause and helps the I/O stack to reduce the artificially high number of in-flight requests. This approach dynamically controls the number of the in-flight I/O requests and protects them from going beyond a predetermined limit before replaying each I/O request.

Note that we round up the dynamic limit of in-flight requests to the next power of two number. If we limit the worker thread to keep at most n requests in-flight, it can replay I/O requests at the same pace as they were captured, but cannot speed up even when some I/O requests fall behind the replay schedule due to unexpected latency of system calls. Therefore, it will negatively impact the timing accuracy of the replay process. On the other hand, if we do not put any limitation on the in-flight I/O count, the replay thread tries to catch up as fast as possible and issues too many I/O requests in a short period of time which resulting in *Response-Time Trashing*. Therefore, we round up the predetermined limit of in-flight requests to the next power of two number and let the replay threads speed up slowly to catch up with the schedule if needed.

2.4. I/O Request Bundling

If the inter-arrival time between two consecutive I/O requests is shorter than the average latency of an I/O submission system call, it is impossible for a single worker thread to submit both requests one after another on time with separate I/O submission system calls. On the other hand, if we deploy two worker threads, it is possible to submit both I/O requests on time, but they may arrive out-of-order to the SAN controller due to the variations of system call latency and device driver queue processing. Therefore, the high-fidelity replay engine of *hfplayer* bundles these consecutive requests into one group and lets the *libaio* unroll it on a single worker thread context within the kernel space. As a result, not only the I/O requests arrive at the SAN controller in order but there is less overhead of kernel to user-space context switch involved in the I/O submissions of multiple I/O requests. We recommend bundling multiple I/O requests before the submission if the inter-arrival time between consecutive I/O requests is less than a certain threshold. This inter-arrival time threshold is proportional to the system performance to execute I/O submission system calls. We will discuss this threshold value in Section 5.1.5.

3. Dependency and Scalable Replay

In most applications, there are dependencies that exist between I/O operations. A simple case would be an application that reads an input file, manipulates the data in some manner and writes the result to an output file. In this case, there is a dependency that exists between the read and the write since the write cannot occur until the read and the data manipulation have completed. Therefore, the actual issue time of an I/O request in the application is relative to the latency of its dependent I/Os as well as the data manipulation time.

The I/O latency is determined by the storage device performance and I/O queue depth. On the other hand, the data manipulation time is determined by the computing power of the server. If we replay a captured I/O trace in an environment similar to the one where the application was run initially (i.e., an unscaled environment), both I/O latency and data manipulation time would be the same as the original environment (with a reasonable pre-conditioning step). Therefore, I/O dependencies are maintained by simply replaying the I/O requests in the sequence that they were originally captured.

However, if we want to scale the workload replay to emulate running the application on a faster storage system or host (i.e., a scaled environment), the I/O latency or data manipulation time would be different from those in the original environment. Therefore, simply replaying the I/O requests in the captured order and time does not necessarily maintain real application I/O behavior. The application might issue I/O requests with a different order and perhaps shorter inter-arrival time in a scaled environment. The sequence between dependent I/O requests is actually maintained by the application and should be respected during replay. Therefore, an accurate scalable workload replay tool should be able to estimate these I/O dependencies from a captured trace and issue an I/O request after the completion time of its dependent I/O requests plus a data manipulation time.

Unlike traditional data dependency analysis in the compiler theory, it is not possible to precisely declare dependencies based on block I/O trace information without accessing application source code. Moreover, the file system semantics are not available in a block I/O request and it is not practical to derive dependencies similar to ARTC and TBBT replay approaches [24, 25] based on the file system level operations. Moreover, a set of I/O requests considered as independent from the point of view of either application or file system might become dependent with each other in the block layer since they are

pushed into the same I/O queues and processed in order by the same I/O logic. Therefore, our best effort is to take a heuristic method and speculate potential I/O dependencies from a given trace and propagating I/O-related performance gains along the dependency chains during a workload replay on a faster (scaled) storage device. Finally in Section 5.2 we demonstrate the accuracy of our heuristic method by comparing the performance characteristics of the replayed workloads to real workloads generated by the applications on the scaled environment.

3.1. Inferring Dependency from Block I/O Trace

A block I/O trace usually includes information about the read and write request types, logical block address, transfer size and timestamps of issuing and completing I/O requests. Unfortunately, this information of the I/O operations itself does not help inferring the dependencies. We try to infer request dependencies using the timing relationships between these operations in the workload trace.

Inferring potential I/O dependency might create both false positives and false negatives. A false positive is to mistakenly identify two requests as dependent with each other, whereas they are independent. Therefore, they are replayed from a single dependency chain while they originated from separate chains in the application. A false negative is the prediction that mistakenly considers two dependent requests as independent. Therefore, they are replayed from separate dependency chains, while their relative order and timing are ignored. Consequently, the performance gains originated from the scaled system are not properly distributed among these requests.

False negative dependencies are more destructive for the workload replay accuracy on a scaled system because falsely identified independent requests can take advantage of any performance gains and replay as fast as possible with an arbitrary order that is not reproducible by the original application. On the other hand, false positive dependencies are more conservative since they are only allowed to take advantage of performance gain in a single dependency chain, while they could issue independently and perhaps even faster by the original application.

Moreover, some of these false positive dependencies may actually help improve the workload replay accuracy. For example, consecutive dependent I/O requests with a very long inter-arrival time are probably independent from the application point of view. They may be generated by independent threads at different times or the same thread with a long idle time. However, *hfplayer* still considers these consecutive requests as dependent to preserve the long idle time during the workload replay task. Otherwise, such a long think time will be dismissed if these requests are being considered as independent and will be replayed around the same time from separate dependency chains.

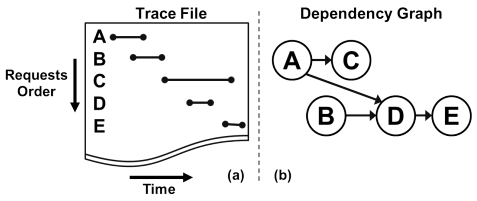


Figure 2: Example of Block I/O Dependency Graph

Therefore, it can be beneficial for a workload replay to maintain some of these false positive dependencies.

hfplayer takes the following conservative approach to infer I/O dependencies with a minimum false negative prediction rate. First it assumes all consecutive pairs of requests are dependent, meaning that the second request depends on the outcome of the first request. Since no independent requests are detected initially, the false negative prediction rate would be zero. However, the false positive rate is at a maximum value. Second, it scans the I/O trace and tries to reduce the false positive rate by excluding those requests that are impossible to be dependent with each other. Finally, it searches the remaining list of dependent I/O requests and removes redundant dependencies to simplify the dependency graph.

Requests that are impossible to be dependent on each other are identified solely by their timing information in the trace file. Each block I/O operation in the request trace has a start time and a completion time. Start time is the time-stamp indicating when a request arrives at the trace capture point and the completion time is the time-stamp indicating when a request completion signal is returned back to the trace capture point in the I/O stack. Figure 2-a illustrates a sample request trace where the start and completion times of a request create a time interval represented as a line segment capped by two dots. Figure 2-b illustrates the dependency graph that is constructed based on these time intervals. We use traditional directed acyclic graph to visualize I/O dependencies in the trace file. During workload replay, the graph dependency edges implies that child request will be issued after the completion of the parent request plus a manipulation time associated with the parent. *hfplayer* distinguishes independent requests without introducing false negative in the following scenarios:

Overlapping Requests: A pair of I/O operations are identified as independent if they have overlapping time intervals since the first request finishes after the start time of the second request. Therefore, it is impossible for the second request to depend on the outcome of the first request. For example, Requests A and B in Figure 2 are independent since they are identified as overlapping requests.

Short Think Time: Think time is defined as the time duration starting from the completion of the first oper-

ation and ending at the start of the second operation. A pair of I/O operations with a very short think time are also identified as an independent pair. The threshold value of this short think time varies depending on the processing power of the application server and represents the time it takes to push up the completion signal of the first request from the capture point to the file system layer plus the time to issue the second operation from the file system. Therefore, it is impossible for the second request to depend on the outcome of the first request with a very short think time. Although an artificially large threshold value for the think time helps reduce the false positive rate, it may also introduce a false negative since a pair of dependent I/O requests is falsely marked as independent. Therefore, we take a conservative approach and just assign the smallest possible threshold of think time for a consecutive pair of I/O requests. This would help to keep the minimum false negative rate. This threshold time is around $10\mu s$ in our test environment where the trace capture point is located in the SAN controller. For example, the think time between Requests B and C in Figure 2 is too short such that they are considered as independent requests.

After excluding independent requests based on their timing positions, *hfplayer* scans the remaining list of dependent I/O requests, constructs a dependency graph and removes redundant dependencies. A pair of I/O operations has a redundant dependency in the graph if there is an indirect path from the first to the second operation in the graph. Therefore, removing redundant dependencies does not affect semantic information of the graph. For example, the dependency between Requests A and E in Figure 2 is maintained implicitly since Request E waits for the completion of Request D and Request D starts after completion of Request A. Therefore, there is no need to add another dependency edge from Request A to Request E.

3.2. Scalable Replay According to I/O Dependency

A scaled server may be capable of running the application at a faster pace and pushing more I/O requests into the I/O stack with less think time. On the other hand, a scaled storage device may pull more I/O requests out of the stack and process the requests faster with shorter response-time (I/O latency). An accurate I/O replay on a scaled system not only requires I/O dependency prediction, but also requires differentiation between server and storage scale factors.

Figure 3 illustrates the way that block I/O replay tools issue Requests A and B on a scaled storage device where Request B is dependent to the outcome of Request A. We assume that the workload is replayed from a server with the performance characteristics similar to that of the original application server. Therefore, it is expected to see the same think time and reduced I/O latency for both

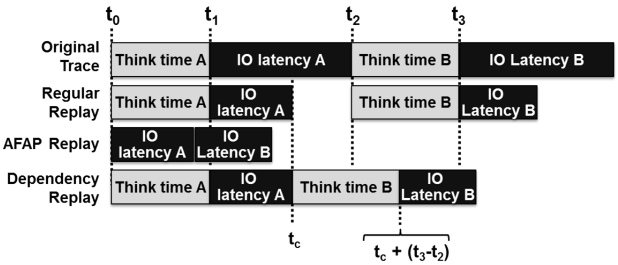


Figure 3: Replay methods on the Scaled Storage

Requests A and B. Typical workload replay tools either ignore the I/O latency reduction (i.e., issue I/O requests based on their original scheduled time: regular replay), or ignore the scheduled time (i.e., issue I/O requests As Fast As Possible: AFAP replay). We believe neither one of these approaches is accurate. The regular replay approach issues the requests at the same speed as they were captured on the unscaled storage device. The AFAP replay approach issues I/O requests too fast which can violate the I/O request dependencies and accumulates a large number of in-flight I/O requests that severely impacts the response-time and throughput accuracy of the reproduced workload.

hfplayer emulates the execution of the original application on a scaled storage device by preserving the original think times and dynamically adjusting the issue times of I/O operations according to their dependent parent's completion time. If the completion signal of Request A (parent) arrives at t_c, the issue time of Request B (child) will be set to $t_c + (t_3 - T_2)$ during the replay process.

Note that only a few I/O requests in the beginning of the workload trace that are in-flight at the same time do not have parents. Therefore, almost all I/O requests in the workload trace have at least one dependent parent that form multiple dependency chains. The total number of parallel dependency chains at any point of the time obviously cannot exceed the maximum number of in-flight I/O requests recorded in the workload trace file.

4. hfplayer Architecture

The main components of *hfplayer* are illustrated in Figure 4. The dependency analyzer module is used only during the scaled replay mode. It takes the trace file with a short inter-arrival time threshold from the inputs, identifies I/O request dependencies and creates a dependency graph based on the method as we have described in Section 3. Finally, this module annotates the original trace file and creates a new intermediate trace file with a list of dependent parents for each I/O request.

We have deployed four types of threads for a replay session, the worker threads, a harvest thread, a timer thread and the main application thread. The main application thread is responsible for initializing and preparing the required data structures for other threads. These data structures include individual I/O request data structure with

optional request bundling and dependency information as well as internal I/O queues dedicated to each worker thread. We have carefully aligned these data structures to the processor cache line size to avoid multiple cache misses when *hfplayer* touches these data structures.

In the unscaled replay mode, each worker thread picks an I/O request from its request queue and issues the request according to the scheduled issue time recorded in the I/O data structure. It also makes sure that the I/O stack is not operating in the response time trashing mode by keeping track of the total number of I/O requests issued and completed as we discussed in Section 2.3.

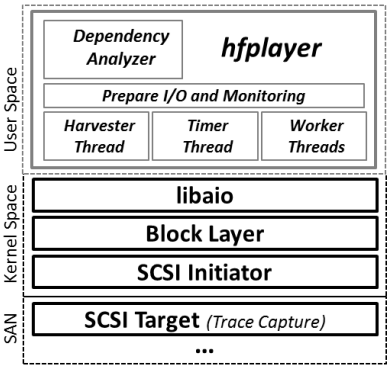


Figure 4: hfplayer *Components on Linux I/O Stack*

In the scaled replay mode, a worker thread issue an individual I/O request after it receives the completion signal of all of its parents plus their corresponding think times. After a worker thread successfully issues an I/O request to the storage controller, its I/O completion signal will be picked up by the harvester thread. Then, the harvester thread decrements the number of in-flight parents counter in its children's data structure. This process eventually reduces individual children's counter to zero and then let other worker threads to issue its children requests in a dependency chain.

Finally, a timer thread is used to share a read-only timer variable to all worker threads with nanoseconds accuracy. Worker threads in the unscaled replay mode compare this shared timer value with the scheduled issue time of the current request to make sure I/O requests are submitted on time. We add a fixed offset to the timer value to make sure all worker threads have enough time to start and wait for the submission of the very first I/O request.

5. Evaluation

We evaluate the replay accuracy of the *hfplayer* in the following two scenarios. First, we evaluate its **high-fidelity unscaled replay engine** in a unscaled environment and compare it with existing replay tools in Section 5.1. Then, we evaluate its **scalable replay engine** on scaled storage devices in Section 5.2. These experiments are done with Linux 3.13 kernel running on a 2x6-Core Xeon X5675 server with 64GB memory and 8Gbps Fiber Channel connection to a NetApp E-Series SAN controller.

We have created three SAN volumes for these experiments with RAID-0 configuration. The first volume is composed of four enterprise SAS solid-state drives (4xSSD volume). The second volume is composed of nine enterprise SSDs (9xSSD volume) and the last SAN volume is composed of two 15K RPM SAS hard disk drives (2xHDD volume). We have disabled the read and write cache since warming up the SAN controller caches is out of the scope of this work. Note that the performance numbers presented in this section do not represent the actual capability of the products used in these experiments.

5.1. Workload Replay on the Unscaled Storage

In this experiment, we collect 12 I/O workload traces, then replay them with multiple replay tools and quantify the unscaled workload replay accuracy. At first, we compare the *hfplayer* replay accuracy with other existing block I/O replay tools. Next, we evaluate its multi-threading and bundling features.

5.1.1. Workload Types: An accurate replay tool should be able to faithfully replay any workloads in a wide intensity spectrum from low to high throughputs. However, as we mentioned earlier it is a challenge to maintain the replay accuracy for intensive and high throughput workloads. Therefore, we need a tunable synthetic I/O workload to demonstrate when the accuracy of existing replay tools drops as we increase the I/O throughput in this experiment. Although it is more appealing to replay the workload trace of a realistic application, we use synthetic workloads in this experiment since most of the open-source enterprise and HPC applications does not use a raw volume without a file system. Therefore it require a large cluster of servers and huge parallelism to overcome the application and file system layer overheads and stress the storage controller with an intensive I/O workload. For example, a recent study reported that 168 Lustre nodes (OSS and OST) are required to create a storage space with 100K IOPS [7]. Obviously, producing such a workload is a challenge not only for us with limited hardware/software resources but also for the research community to reproduce and validate our research results. Therefore, we have selected synthetic workloads that can stress the SAN controller up to 200K IOPS and then demonstrate how *hfplayer* can replay these workloads with high-fidelity on a similar storage device.

We have used *FIO* and its *libaio* engine to generate 11 synthetic I/O workloads that cover low to high throughput intensity spectrum on the raw 4xSSD volume. We have also selected *Sequential Write* and *Random Read/Write* with a default 50/50 split for the I/O patterns. These two patterns then are used to generate 10 workloads with the incremental throughput tuned by *iodepth* values of 16, 32,

64, 128 and 512. Finally, we generated another workload with the I/O pattern of *Sequential Read* and *iodepth* of 512 to create a very high-throughput workload for evaluation of multi-threading and request bundling features of *hfplayer*. All of these workloads are generated for the duration of 60 seconds. The traces of block I/O requests that have been generated by these workloads are collected from the SAN controller SCSI Target layer. These trace files are then used in our replay experiments. *hfplayer* input trace file format is compatible with SNIA draft of block-trace common semantics [17] and does not include proprietary fields.

5.1.2. Replay Tools: We compare the workload replay accuracy of *hfplayer* with two publicly available block I/O replay tools. First, we use *blkreplay* [15] which is developed by a large web hosting company for the performance validation of block storage devices used in its data centers. *blkreplay* is built with a synchronous multi-threaded I/O engine as we mentioned in Section 2 and suffers from inter-thread coordination overhead. We configure it with *no-overhead* and *with-conflicts* to disable its write-verify and write-protection mode in order to make an apples to apples comparison with other tools. Moreover, we set the maximum thread count to 512 since some of our workloads carry-on such a high number of in-flight I/O requests.

Second, we use *btreplay* [8] which is developed originally to replay workloads that are captured with the *blktrace* tool on Linux. It uses *libaio* with an asynchronous I/O engine like *hfplayer* and has a similar feature to combine the I/O requests into a bundle if they fit in a fixed time window with a fixed request count per bundle. The bundle time window and the maximum request count are statically set in the command line and are not dynamically set according to the workload intensity. However, as we mentioned in Section 2.4, *hfplayer* can dynamically adjust the bundle size since it bundles requests together if their issue time is very close to each other. Moreover, we just use a single worker thread in the *btreplay* since it cannot replay a single trace file using multiple worker threads. We configure *hfplayer* to work with a single worker thread for a fair comparison as well and evaluate its multi-threading and bundling features separately in Section 5.1.5. We do not use *FIO* replay engine since it is similar to *btreplay* and uses asynchronous I/O engine. Moreover, it has been reported in the open-source community that *FIO* is not as accurate as *btreplay* [1].

5.1.3. Accuracy Metrics: An accurate unscaled workload replay should maintain both temporal and spatial characteristics of the original trace workload. Temporal characteristics of an I/O workload are usually quantified by I/O throughput and response-time. We use relative error percent of average response-time to demonstrate the response-time accuracy of the workload replay. Moreover,

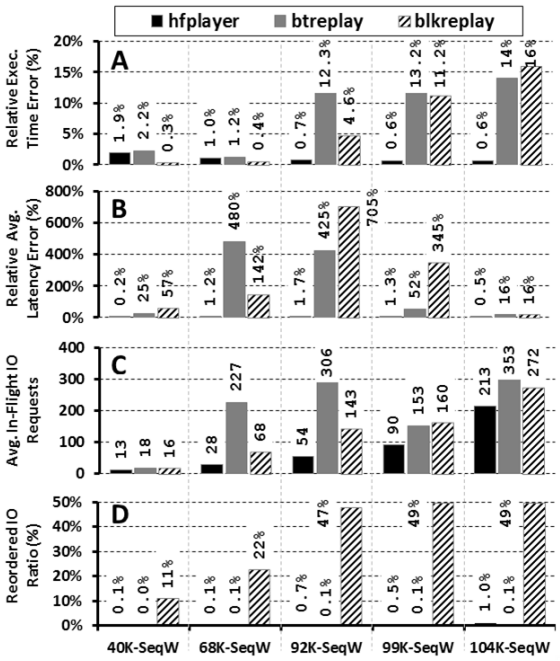


Figure 5: Replay Accuracy for the Sequential Write Workloads

determining factor of I/O throughput is the total execution time since the number of replayed requests is the same in each replay experiment. Therefore, we use relative error percentage of execution time to demonstrate the I/O throughput accuracy of the workload replay. The spatial characteristics of the workload replay remain the same if all I/O requests arrive in the storage device(s) with the same order that is presented in the original trace. Therefore, we have used the Type-P-Reordered metric [12] to measure the number of requests that are reordered in the replayed workload.

5.1.4. Workload Replay Accuracy Comparisons: Figure 5 demonstrates the accuracy of the three replay tools that we have tested using five sequential write workloads. The shared horizontal axis represents the demanding throughput of the captured workload in IOPS. The vertical axes A, B and D represent three accuracy metrics discussed earlier. First, axis A shows the relative execution error of the replayed workload. While all three replay tools can reproduce low throughput workloads with negligible timing errors, *btreplay* and *blkreplay* cannot keep up when throughput demand increases from 92K to 104K IOPS and cannot finish the replay task on time.

Second, axis B shows the relative error of average I/O response-time during the workload replay compared with the original workload. Once again all three tools have a negligible response-time error to replay the low throughput workload (40K IOPS). However, both *btreplay* and *blkreplay* fail to replay with accurate response-time for higher throughput demanding I/O workloads. Starting with 68K IOPS workload, these tools build an artificially

high number of in-flight I/Os in the I/O path as we show in axis C which results in *Response-Time Trashing*. The I/O response-time error of the replay task for the most demanding I/O workload (104K) is negligible because the I/O path is already overloaded and achieving its maximum response-time at this throughput rate. Therefore, the internal overhead of *btreplay* and *blkreplay* prevents them from achieving the target throughput and causes them to finish the replay task with 14% and 16% relative execution time error respectively.

Finally, axis D shows the ordering accuracy of the replayed workload. *blkreplay* is based on a multi-threaded synchronous I/O engine. Therefore, it is expected to see it reorders most of I/O requests in the workload. We believe reordering can significantly change the workload characteristics. In this case, it converts a sequential write workload into a half sequential, half random workload. *btreplay* and *hfplayer* both use the single threaded async I/O engine and can submit all I/O requests in order but a negligible number of these requests are reordered in the I/O stack by the SCSI Initiator since it operates in the *Simple* TCQ mode which is inevitable.

Figure 6 shows the replay accuracy for mixed random read/write workloads. First, axis A shows all three replay tools are capable of replaying mixed workloads with a negligible error. That is because the overhead of the I/O stack is lower for read/write workload compared with previous write-only workload. Typically write requests are more resource consuming since writes carry a payload that requires extra protection for data consistency and durability both in the Kernel and SAN controller I/O stacks. Second, axis D shows both *blkreplay* and *btreplay* fail to replay the workload with accurate response-time which is directly impacted by the number of in-flight I/O requests shown by axis C. Note that the only workload that *hfplayer* does not provide the best response-time accuracy is on the 99K IOPS workload, where its relative error rate is 52%, whereas *blkreplay* error rate is 16%. That is because *blkreplay* managed to replay the workload with a lower number of in-flight I/O requests. However, *blkreplay* cannot maintain a low request ordering accuracy at the same time. As we show in axis D, about 34% of requests that are issued with this tool at 99K IOPS workload arrived in the controller out of order.

Note that the number of reordered requests in the mixed read/write workload is significantly more than sequential write workload for *btreplay* and *hfplayer* (comparing axis D in Figures 6 and 5). These requests are submitted in order with a single worker thread in both replay tools. However, they are reordered in the SCSI initiator layer since read and write requests have different service time. Usually read requests are serviced faster since the read I/O path is faster on a SAN controller without a caching layer. Therefore, the queue tags that are allocated for the

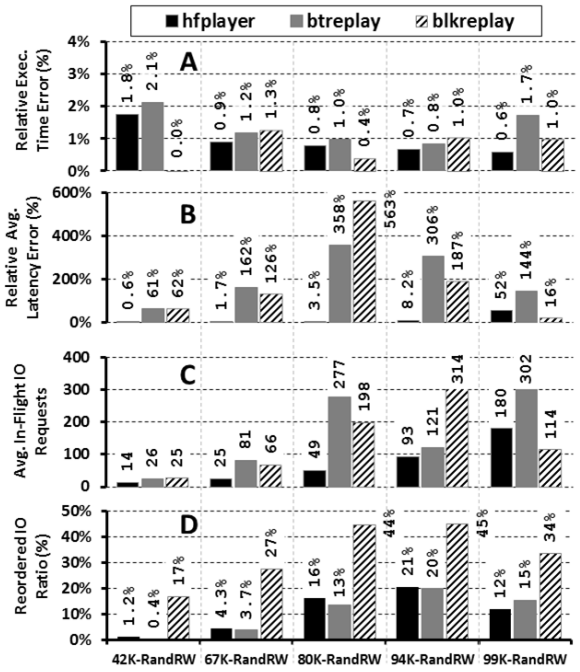


Figure 6: Replay Accuracy for the Random R/W Workload

read requests in the SCSI initiator layer are deallocated faster than the tags allocated for writes requests. However, in the write-only workload, all the tags are allocated and deallocated with the same rate and the SCSI initiator queue will operate similar to a FIFO queue.

5.1.5. Multi-threading and Request Bundling: We did not evaluate these two features of the *hfplayer* in the previous experiments to make an apples to apples comparison with other replay tools. However, these two features are useful during the replay of very high throughput workloads where the other replay tools are incapable of maintaining replay accuracy as we elaborated in the previous subsection. In this experiment, we go extreme and replay a very high throughput sequential read workload with 232K IOPS. This workload is captured and replayed on a 9xSSD volume, while the previous experiments were done on a 4xSSD volume.

Figure 7 shows the replay accuracy of *hfplayer* where multi-threaded and bundling features are used. The horizontal axis shows the maximum inter-arrival time between two requests to fit into a bundle. Obviously, *hfplayer* forms larger bundles with a larger maximum inter-arrival time value. We have replayed the workload with one to eight threads for each time to evaluate the multi-threading feature as well. On the vertical axes, we just show the relative execution time error (or throughput accuracy) and reordered I/O ratio. We do not show the average response-time accuracy since the I/O path in all of these experiments is fully loaded. Therefore, I/O response-times are all saturated and close to expectation. We have described a similar behavior in Figure 5-B that happens

for all replay tools during replay of a high-throughput 104K IOPS workload. Figure 7 shows that *hfplayer* cannot finish replaying the 232K IOPS workload on time when the bundling feature is disabled. Enabling the multi-threaded feature helps to reduce the relative execution time error from 28% to 27% and 21% with two and four worker threads respectively. The improvement is minimal and the resulting error rate is still high. Moreover, using eight worker threads does not improve the replay accuracy further due to extra threading overhead to acquire block layer queue lock. On the other hand, adding more worker threads significantly increases the reordered I/O rate.

In contrast, when request bundling is enabled (for example, set bundling threshold to $20\mu s$), the ordering errors are reduced to around 30% in multiple worker threads mode. Moreover, multiple worker threads help *hfplayer* submit all I/O requests on time and reduce the execution time error rate. As we increase the bundling threshold, we see a negligible change in the accuracy metrics. That is because *hfplayer* can fit up to 256 requests in a bundle and increasing the bundle threshold cannot create a larger bundle when almost all bundles are full. The maximum bundle size limit is an implementation trade-off that forces the bundle data structure to fit in a CPU cache line. Therefore, the replay task does not suffer from multiple cache line misses when it sends a bundle to the kernel.

Finally, Figure 7 shows a significant amount of I/O requests are reordered in the single threaded mode when bundling is enabled. For example, 29.3% of requests arriving in the SAN controller are out of order when we set the bundling threshold to $30\mu s$. All of these requests are inserted into the SCSI initiator queue in order using a single thread that mostly lives in the kernel space. In other words, the *io_submit* system call unrolls a bundle and issues individual I/O requests from the kernel space and does not context switch to the user space frequently. Therefore, the tag allocation in the SCSI initiator queue takes place at a faster pace compared with deallocation. As a result, the SCSI driver cannot allocate in order tags when a burst of tag allocation requests arrive during the unrolling of a bundle in the kernel space.

5.2. Evaluate Scalable Workload Replay

In this subsection, we evaluate the scalability feature of the *hfplayer* and its scaled replay engine. The ultimate goal of this experiment is to evaluate how accurate *hfplayer* can replay a workload on a faster storage device, given a workload trace file that is captured from a slower storage device. First, we will describe the workloads that we have used for this experiment. Then we describe the evaluation methodology and finally we present the evaluation results.

5.2.1. Workload Types: As we described in Section 3, the dependency replay engine of *hfplayer* tries to emulate the application behavior by speculating the I/O dependencies from a given trace file and replay I/O requests according to their dependencies. Therefore, we could not use synthetic workloads for this experiment since they do not contain realistic dependency information that can be emulated. Therefore, we have tried to use realistic applications on top of a file system for this purpose. Instead of using a commercial I/O intensive application (like an Oracle Database) or making our own I/O intensive application (like copying large ISO files), we have used *Filebench* [21] and *mkfs* to generate the workloads for this experiment. These applications are available in the public domain and can be used by others to reproduce the results.

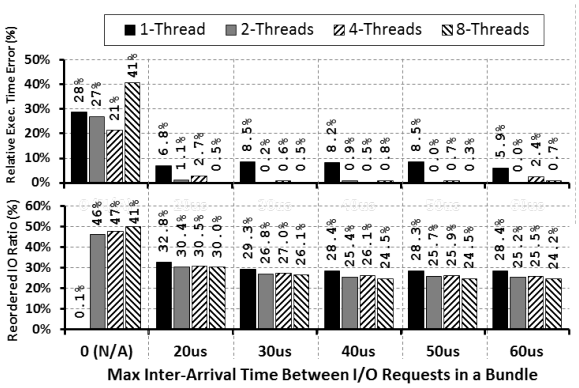


Figure 7: Multi-threading and Request Bundling Evaluation

The key features of the selected I/O intensive applications are a) they perform direct I/O instead of buffered I/O which eliminates the ext4 buffer cache impact on the I/O performance, b) they perform a meaningful task from the application layer and their durations depend only on the performance of the storage device, and c) the task execution time is fixed and reproducible on a single storage device.

According to these criteria, we have selected *Copyfiles* and *Createfiles* benchmarks from the *Filebench* suite. The first benchmark is configured to copy 100K files with 256 threads. The second benchmark is configured to create 50K files with mean directory width of 100 and mean file size of 16K using 16 threads and a 1MB request size. Both benchmarks are set to do direct I/O and the other configuration parameters are set to default. These two benchmarks run a meaningful I/O intensive task on the SAN volumes mounted with the *ext4* file system.

The *mkfs* application creates another I/O intensive workload during the file system initialization phase. We have created fixed size partitions on all SAN volumes and used *mkfs.ext4* utility without the lazy inode table and journal initialization to create a file system image. This forces *mkfs* to create all inode tree and journal data structures on the disk in the foreground.

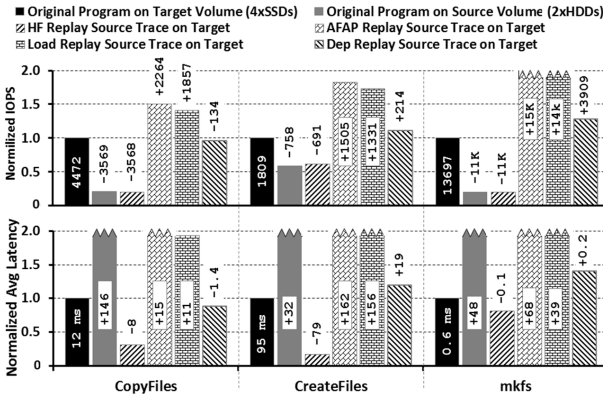


Figure 8: Replay Workloads Captured from 2xHDD on 4xSSD

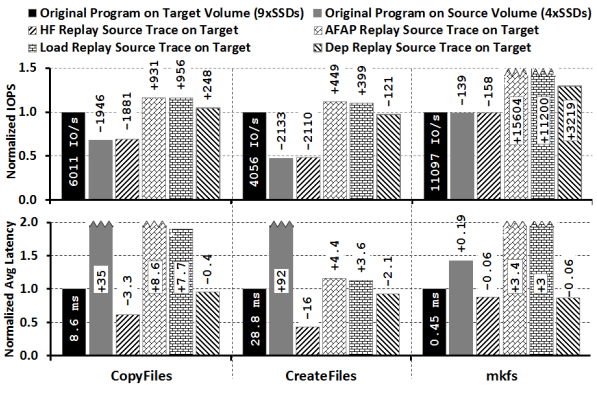


Figure 9: Replay Workloads Captured from 4xSSD on 9xSSD

5.2.2. Evaluation Methodology: We will describe our evaluation methodology with a simple example. Let us assume an I/O intensive application that executes on storage device A for 10 minutes. However, it takes just 5 minutes to run the same application on a faster storage device B. A scalable replay tool should be able to take the trace file of the I/O workload from storage device A (which lasts for 10 minutes) and replay it on device B in 5 minutes. In other words, storage device B takes almost the same workload during the replay as it received when the real application was running. In this example, storage device A is the *Source* and B is the *Target* storage device.

In practice, the source is the slower storage device that is installed in the production site and the target is a faster storage device that is under development or validation test. The workload is captured from the source (on the production site) and replayed on the target storage (in the lab). The expectation is that the replay tool will generate the same workload that the production application would have generated on the target storage and quantifies how much it can improve application performance.

Our methodology to quantify how *hfplayer* can meet such an expectation is as follows. First, we execute the I/O intensive applications on 2xHDD, 4xSSD and 9xSSD volumes for multiple times, capture their I/O workloads from the SAN controller and make sure the execution time of captured workloads are repeatable. We have validated that these applications are repeatable with less than 2.7% relative standard deviation. Then we consider the 2xHDD volume as a source and 4xSSD volume as the target for the replay. This means that the trace files captured on 2xHDD volume are replayed on 4xSSD volume. Finally, we compare the execution time and I/O response-time with the original application trace files that were captured from 4xSSD in the first step. We do the same steps to take 4xSSD volume as a source and 9xSSD as a target.

We use *hfplayer* to replay the captured workload from the source on the target storage in the following four replay modes to make our evaluation more comprehensive. First, we replay with the **HF** or high-fidelity replay engine. Note that this is the same replay mode that we have used in Section 5.1. Second, we replay with the **AFAP** replay mode which ignores the scheduled timing and tries to replay I/O requests on the target storage device as fast as possible. Note that other block I/O replay tools like blkreplay recommend this replay mode to replay a workload on a scaled storage [15]. Third, we replay with the **Load** replay mode which is similar to AFAP, but it only tries to dynamically match the number of in-flight I/O requests as described in a previous work [16]. Finally, we replay with the **Dep** replay mode which is based on the dependency replay engine of the *hfplayer*. It estimates I/O dependencies and replays requests according to those dependencies as we described in Section 3.

5.2.3. Result Discussion: Figure 8 shows the results of taking the captured workload from the 2xHDD volume (source) and replaying it on the 4xSSD volume (target). In this figure, the horizontal axis is the workload type. The top vertical axis is the workload IOPS normalized to original application IOPS running on the target storage device. The bottom vertical axis shows the average response-time again normalized with what is expected to see on the target storage device. An accurate replay should have a patterned bar that is very close to the solid black bar. The numbers on the patterned and gray bars show the error values or how far each bar is from the solid black bar.

This figure shows that the dependency replay mode can accurately replay captured workload from the 2xHDD volume on the 4xSSD volume in terms of both IOPS and average response-time. The IOPS generated with high-fidelity replay mode matches with the low throughput of the source volume (gray bar) and its its response-time is significantly lower than the original application response-time on the target storage (black bar). AFAP and Load replay mode both replay the workload on the target volume with a significantly higher throughput. As a result, they queue more requests than expected in the I/O path and inflate the response-time. Even the in-flight I/O rate control mechanism that is embedded in the Load replay

mode does not help to slow down the workload replay and match with the original program I/O performance running on the target volume.

Finally, Figure 9 shows the results of taking a captured workload from the 4xSSD volume and replaying it on the 9xSSD volume. This figure shows that dependency replay mode of *hfplayer* can accurately replay a workload on a scaled storage even if the performance variation between the source and target is not significant. For example, *mkfs* is a single threaded application that typically does synchronous I/O with one in-flight request at a time. Therefore, as we see in this figure, both the source and target storage devices have the same IOPS value (solid black and solid gray bars). That is because only one SSD produces the throughput at any point in time with one in-flight I/O request. This is the worst case for *hfplayer* with dependency replay with an IOPS error rate less than 30%, compared to less than 10% IOPS error rates in all other cases.

6. Related Work

Various trace replay tools have been developed at file system level. For example, Joukov et al. developed Replayfs [9], a tool that can replay file system I/O traces that were captured at the VFS level. Zhu et al. proposed TBBT [25], an NFS trace replay tool that automatically detects and recovers missing file system I/O operations in the trace. Mesnier et al. proposed //TRACE [11] for replaying traces of parallel applications. It achieves a high accuracy in terms of inter-node data dependencies and inter-I/O compute times for individual nodes by utilizing a throttling technique. More recently, Weiss et al. designed ARTC [24], a new method of replaying system call traces of multi-threaded applications which can explore some non-deterministic properties of the target application. However, as we mentioned in Section 1 none of these replay tools are capable to reproduce high-throughput workload due to intrinsic file system overhead. More recently, Pereria et al. compares the replay accuracy of ARTC with TBBT [13].

There are several other block I/O replay tools. Liu et al. designed TRACER [10], a replay tool used for evaluating storage system energy efficiency. It can selectively replay a certain percentage of a real world block I/O trace to reach different levels of workload intensity by filtering trace entries uniformly. Anderson et al. proposed Buttress [6] as a toolkit to replay block traces with a loose timing accuracy of $100\mu s$ more than 10 years ago. This tool uses synchronous I/O and thus requires instantiating a great number of threads in order to achieve a high number of outstanding I/O requests on the target storage system. Therefore, its replay performance and scalability are limited by the threading overhead and cannot keep up with the capabilities of modern SAN storage systems. Sivathanu et al. proposed a load-aware trace replay [16] that aims to preserve the same I/O load pattern of the original application traces irrespective of the performance of the target storage system. However, we have evaluated this technique in Section 5.2 and demonstrated that it cannot replay a workload on the scaled target storage accurately.

More recently, Tarihi et al. proposed DiskAccel [22], a sampling methodology to accelerate trace replay on conventional disk drives. DiskAccel uses a weighted variant of the K-Means clustering algorithm to select representative intervals of the I/O trace file. These I/O intervals instead of the whole trace are then replayed on the target disk drive. Therefore, a week long captured I/O trace file can be replayed in about an hour, while maintaining the same average I/O response-time. Moreover, Tarasov et al. proposed a flexible workload modeling technique that extract a mathematical model from the block trace [20]. This model is then used as an input for a benchmark tool to reproduce the workload. These trace reduction and modeling methodologies are complementary to our work and can be used to shrink the trace size and I/O workload duration with *hfplayer* as well. DiskAccel also implements a method to enforce I/O requests dependency during the replay job. However, due to the lack of block I/O dependency information, it assumes all reads requests are dependent and all write requests are independent. In contrast, *hfplayer* infers dependency information without such an unrealistic assumption.

Finally, Tarasov mentioned a few limitations of the workload replay on a scaled storage [18]. He described the dependency replay as a viable approach but claimed that approximation of I/O dependencies from the block layer can add extra dependencies that does not exist in the original workload. Therefore, the workload replay effort might not be as accurate as workload modeling effort. In this work, we have demonstrated a method to make an approximation of the I/O dependencies and found that the dependency workload replay can reproduce original application workload on a scaled storage device.

7. Conclusions and Future Work

In this paper, we have introduced new methods to replay intensive block I/O trace in a scaled or unscaled environments with more accuracy. First, we have proposed a detailed analysis of various points preventing an accurate replay in Linux I/O stack. Second, we have considered the notion of dependency between block I/O requests and then described how the *hfplayer* infers and replays events in a dependency aware fashion on a scaled system, efficiently propagating I/O-related performance gains along dependency chains. Finally, we have provided a careful evaluation of the *hfplayer* in both scaled and unscaled environments. In the future, we seek to port our replay tool to IBM's AIX operating system.

8. Acknowledgments

We thank our shepherd, Remzi Arpaci-Dusseau and Matias Bjørling, and the anonymous reviewers for their comments and suggestions. This work has been supported by NSF I/UCRC Center for Research in Intelligent Storage (CRIS) and the National Science Foundation (NSF) under awards 130523, 1439622, and 1525617 as well as the support from NetApp.

References

[1] fio mailing list, re: fio replay.

[2] Intel storage performance development kit (spdk) official web site.

[3] Linux bug fix patch: Enable sysfs nomerge control for i/o requests in the plug list.

[4] Spc-1 benchmak results fot top 10 block storage by price-performance.

[5] ANDERSON, E. Buttress, a cautionary tale. In *File and Storage Systems Benchmarking Workshop* (2008).

[6] ANDERSON, E., KALLAHALLA, M., UYSAL, M., AND SWAMINATHAN, R. Buttress: A toolkit for flexible and high fidelity i/o benchmarking. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004).

[7] BOURILKOV, D., AVERY, P. R., FU, Y., PRESCOTT, C., AND KIM, B. The lustre filesystem for petabyte storage at the florida hpc center. In *HEPiX Fall 2014 Workshop*.

[8] BRUNELLE, A. D. btrecord and btreplay user guide.

[9] JOUKOV, N., WONG, T., AND ZADOK, E. Accurate and efficient replaying of file system traces. In *FAST* (2005), vol. 5, pp. 25–25.

[10] LIU, Z., WU, F., QIN, X., XIE, C., ZHOU, J., AND WANG, J. Tracer: A trace replay tool to evaluate energy-efficiency of mass storage systems. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on* (2010), pp. 68–77.

[11] MESNIER, M. P., WACHS, M., SIMBASIVAN, R. R., LOPEZ, J., HENDRICKS, J., GANGER, G. R., AND O'HALLARON, D. R. //trace: parallel trace replay with approximate causal events. USENIX.

[12] MORTON, A., CIAVATTONE, L., RAMACHANDRAN, G., SHALUNOV, S., AND PERSER, J. Packet reordering metrics. *IETF internet-standard: RFC4737* (2006).

[13] PEREIRA, T. E., BRASILEIRO, F., AND SAMPAIO, L. File system trace replay methods through the lens of metrology. *32nd International Conference on Massive Storage Systems and Technology (MSST 2016)*.

[14] SCHROEDER, B., WIERMAN, A., AND HARCHOL-BALTER, M. Open versus closed: A cautionary tale. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation, NSDI* (2006), vol. 6, p. 18.

[15] SCHBEL-THEUER, T. blkreplay and sonar diagrams.

[16] SIVATHANU, S., KIM, J., KULKARNI, D., AND LIU, L. Load-aware replay of i/o traces. In *The 9th USENIX Conference on File and Storage Technologies (FAST), Work in progress (WiP) session* (2011).

[17] (SNIA), S. N. I. A. Block i/o trace common semantics (working draft).

[18] TARASOV, V. *Multi-dimensional workload analysis and synthesis for modern storage systems*. PhD thesis, Stony Brook University, 2013.

[19] TARASOV, V., KUMAR, S., MA, J., HILDEBRAND, D., POVZNER, A., KUENNING, G., AND ZADOK, E. Extracting flexible, replayable models from large block traces. In *FAST* (2012), p. 22.

[20] TARASOV, V., KUMAR, S., MA, J., HILDEBRAND, D., POVZNER, A., KUENNING, G., AND ZADOK, E. Extracting flexible, replayable models from large block traces. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), vol. 12, p. 22.

[21] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A flexible framework for file system benchmarking.

[22] TARIHI, M., ASADI, H., AND SARBAZI-AZAD, H. Diskaccel: Accelerating disk-based experiments by representative sampling. *SIGMETRICS Perform. Eval. Rev. 43*, 1 (June 2015), 297–308.

[23] TRAEGER, A., ZADOK, E., JOUKOV, N., AND WRIGHT, C. P. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS) 4*, 2 (2008), 5.

[24] WEISS, Z., HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Root: Replaying multithreaded traces with resource-oriented ordering. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 373–387.

[25] ZHU, N., CHEN, J., CHIUEH, T.-C., AND ELLARD, D. Tbbt: scalable and accurate trace replay for file server evaluation. In *ACM SIGMETRICS Performance Evaluation Review* (2005), vol. 33, ACM, pp. 392–393.

On the Performance Variation in Modern Storage Stacks

Zhen Cao[1], Vasily Tarasov[2], Hari Prasath Raman[1], Dean Hildebrand[2], and Erez Zadok[1]
[1]*Stony Brook University* *and* [2]*IBM Research—Almaden*

Abstract

Ensuring stable performance for storage stacks is important, especially with the growth in popularity of hosted services where customers expect QoS guarantees. The same requirement arises from benchmarking settings as well. One would expect that repeated, carefully controlled experiments might yield nearly identical performance results—but we found otherwise. We therefore undertook a study to characterize the amount of variability in benchmarking modern storage stacks. In this paper we report on the techniques used and the results of this study. We conducted many experiments using several popular workloads, file systems, and storage devices—and varied many parameters across the entire storage stack. In over 25% of the sampled configurations, we uncovered variations higher than 10% in storage performance between runs. We analyzed these variations and found that there was no single root cause: it often changed with the workload, hardware, or software configuration in the storage stack. In several of those cases we were able to fix the cause of variation and reduce it to acceptable levels. We believe our observations in benchmarking will also shed some light on addressing stability issues in production systems.

1 Introduction

Predictable performance is critical in many modern computer environments. For instance, to achieve good user experience, interactive Web services require stable response time [9, 20, 22]. In cloud environments users pay for computational resources. Therefore, achieving predictable system performance, or at least establishing the limits of performance variation, is of utmost importance for the clients' satisfaction [37, 48]. In a broader sense, humans generally expect repetitive actions to yield the same results and take the same amount of time to complete; conversely, the lack of performance stability, is fairly unsatisfactory to humans.

Performance variation is a complex issue and can arise from nearly every layer in a computer system. At the hardware level, CPU, main memory, buses, and secondary storage can all contribute to overall performance variation [9,22]. At the OS and middleware level, when background daemons and maintenance activities are scheduled, they impact the performance of deployed applications. More performance disruptions come into play when considering distributed systems, as applications on different machines have to compete for heavily shared resources, such as network switches [9].

In this paper we focus on characterizing and analyzing performance variations arising from benchmarking a typical modern storage stack that consists of a file system, a block layer, and storage hardware. Storage stacks have been proven to be a critical contributor to performance variation [18, 33, 40]. Furthermore, among all system components, the storage stack is the cornerstone of data-intensive applications, which become increasingly more important in the big data era [8, 21]. Although our main focus here is reporting and analyzing the variations in benchmarking processes, we believe that our observations pave the way for understanding stability issues in production systems.

Historically, many experienced researchers noticed how workloads, software, hardware, and the environment—even if reportedly "identical"—exhibit different degrees of performance variations in repeated, controlled experiments [7, 9, 11, 22, 23]. We first encountered such variations in experiments using Ext4: multiple runs of the same workload in a carefully controlled environment produced widely different performance results. Over a period of two years of collecting performance data, we later found that such high performance variations were not confined to Ext4. Over 18% of 24,888 different storage stack configurations that we tried exhibited a standard deviation of performance larger than 5% of the mean, and a range value (maximum minus minimum performance, divided by the average) exceeding 9%. In a few extreme cases, standard deviation exceeded 40% even with numerous repeated experiments. The observation that some configurations are more stable than others motivated us to conduct a more detailed study of storage stack performance variation and seek its root causes.

To the best of our knowledge there are no systematic studies of performance variation in storage stacks. Thus, our first goal was to characterize performance variation in different storage stack configurations. However, measuring this for even a single storage stack configuration is time consuming; and measuring all possible stack configurations is time-prohibitive. Even with a small fraction of selected parameters, it could take more than 1.5 years of evaluation time (see Table 1). Therefore, in this study we combined two approaches to reduce the configuration space and therefore the amount of time to run the experiments: (1) we used domain expertise to select the most relevant parameters, and (2) we applied a Latin Hypercube Sampling (LHS) to the configuration space. Even for the reduced space, it took us over 33 clock days

to complete these experiments alone.

We focused on three local file systems (Ext4, XFS, and Btrfs) which are used in many modern local and distributed environments. Using our expertise, we picked several widely used parameters for these file systems (e.g., block size, inode size, journal options). We also varied the Linux I/O scheduler and storage devices, as they can have significant impact on performance. We benchmarked over 100 configurations using different workloads and repeated each experiment 10 times to balance the accuracy of variation measurement with the total time taken to complete these experiments. We then characterized performance variation from several angles: throughput, latency, temporally, spatially, and more. We found that performance variation depends heavily on the specific configuration of the stack. We then further dove into the details, analyzed and explained certain performance variations. For example: we found that unpredictable layouts in Ext4 could cause over 16–19% of performance variation in some cases. We discovered that the magnitude of variation also depends on the observation window size: in one workload, 40% of XFS configurations exhibited higher than 20% variation with a window size of 60s, but almost all of them stabilized when the window size grew to 400s. Finally, we analyzed latency variations from various aspects, and proposed a novel approach for quantifying the impacts of each operation type on overall performance variation.

Our paper has three key contributions: ■ **(1)** To the best of our knowledge, we are the first to provide a detailed characterization of performance variation occurring in benchmarking a typical modern storage stack. We believe our study paves the way towards the better understanding of complex storage stack performance variations, in both experimental and production settings. ■ **(2)** We conducted a comprehensive study of storage stack performance variation. Our analysis includes throughput and latency, and both spatial and temporal variations. ■ **(3)** We offer insights into the root causes of some performance variations, which could help anyone who seeks stable results from benchmarking storage systems, and encourage more follow-up work in understanding variations in production systems.

The rest of the paper is organized as follows. §2 explains background knowledge. §3 describes our experimental methodology. We list our experimental settings in §4. §5 evaluates performance variations from multiple dimensions. §6 covers related work. We conclude and discuss future directions in §7.

2 Background

The storage stack is an essential part of modern computer systems, and critical to the performance of data-intensive applications. Often, the storage stack is the slowest component in a system and thus is one of the main contributors to the overall variability in a system's performance. Characterizing this variation in storage-stack performance is therefore essential for understanding overall system-performance variation.

We first define common performance metrics and notations used in this paper. *Throughput* is defined as the average number of I/O operations completed per second. Here we use a "*Throughput-N*" notation to represent the throughput within the last N seconds of an observation. There are two types of throughput that are used most frequently in our analysis. One is *cumulative* throughput, defined as the throughout from the beginning to the end of the experiment. In this paper, cumulative throughput is the same as *Throughput-800* or *Throughput-2000*, because the complete runtime of a single experiment was either 800 or 2,000 seconds, depending on the workload. The other type is called *instantaneous* throughput, which we denote as *Throughput-10*. Ten seconds is the smallest time unit we collected performance for, in order to avoid too much overhead (explained further in § 4).

Since our goal is to characterize and analyze collected experimental data, we mainly use concepts from *descriptive statistics*. *Statistical variation* is closely related to *central tendency*, which is an estimate of the *center* of a set of values. *Variation* (also called *dispersion* or *variability*), refers to the spread of the values around the central tendency. We considered the most commonly used measure for central tendency—the *mean*: $\bar{x} = \sum_{i=1}^{N} x_i$.

In descriptive statistics, a measure of variation is usually a non-negative real number that is zero if all readings are the same and increases as the measurements become more dispersed. To reasonably compare variations across datasets with different mean values, it is common to normalize the variation by dividing any absolute metric of variation by the mean value. There are several different metrics for variation. In this paper we initially considered two that are most commonly used in descriptive statistical analysis:

- *Relative Standard Deviation (RSD)*: the RSD, (or *Coefficient of Variation (CV)*) is

$$RSD = \frac{\sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}}{\bar{x}} \quad (1)$$

- *Relative Range*: this is defined as the difference between the smallest and largest values:

$$RelativeRange = \frac{max(X) - min(X)}{\bar{x}} \quad (2)$$

Because a range uses maximum and minimum values in its calculation, it is more sensitive to outliers. We did not want to exclude or otherwise diminish the significance of performance outliers. We found that even a few

long-running I/O operations can substantially worsen actual user experience due to outliers (which are reproducible). Such outliers have real-world impact, especially as more services are offloaded to the cloud, and customers demand QoS guarantees through SLAs. That is one reason why researchers recently have begun to focus on tail latencies [9, 17, 18]. In considering the two metrics above, we felt that the RSD hides some of the magnitudes of these variations—because using square root tends to "compress" the outliers' values. We therefore decided to use the *Relative Range* as our main metric of variation in this work and the rest of this paper.

3 Methodology

Although we encountered storage stack performance variations in past projects, we were especially struck by this issue in our recent experiments on automated recognition of optimal storage configurations. We found that multiple runs of the same workload in a carefully controlled environment could sometimes produce quite unstable results. We later observed that performance variations and their magnitude depend heavily on the specific configuration of the storage stack. Over 18% of 24,888 different storage stack configurations that we evaluated (repeatedly over several workloads) exhibited results with a relative range higher than 9% and relative standard deviation higher than 5%.

Workloads also impact the degree of performance variation significantly. For the same storage stack configuration, experiments with different workloads could produce different magnitudes of variation. For example, we found one Btrfs configuration produces variation with over 40% relative range value on one workload but only 6% for another. All these findings led us to study the characteristics and analyze performance variations in benchmarking various storage stack configurations under multiple workloads. Due to the high complexity of storage stacks, we have to apply certain methodologies in designing and conducting our experiments.

Reducing the parameter space. In this work we focus on evaluating *local* storage stacks (e.g., Ext4, Linux block layer, SSD). This is a useful basis for studying more complex distributed storage systems (e.g., Ceph [46], Lustre [27], GPFS [34], OpenStack Swift [30]). Even a small variation in local storage system performance can result in significant performance fluctuations in large-scale distributed system that builds on it [9, 25, 28].

Despite its simple architecture, the local storage stack has a large number of parameters at every layer, resulting in a vast number of possible configurations. For instance, common parameters for a typical local file system include block size, inode size, journal options, and many more. It is prohibitively time consuming and impractical to evaluate every possible configuration exhaustively. As shown in Table 1, Ext4 has 59 unique parameters that can have anywhere from 2 to numerous allowed values each. If one experiment runs for 15 minutes and we conduct 10 runs for each configuration, it will take us 7.8×10^{33} years of clock time to finish evaluating all Ext4 configurations.

Parameter Space	#Unique Params.	#Unique Configs.	Time (years)
Ext4	59	2.7×10^{37}	7.8×10^{33}
XFS	37	1.4×10^{19}	4.1×10^{15}
Btrfs	54	8.8×10^{26}	2.5×10^{23}
Expert Space	10	1,782	1.52
Sample Space	10	107	33.4 days

Table 1: Comparison for Parameter Spaces. Time is computed by assuming 15 minutes per experimental run, 10 runs per configuration and 3 workloads in total.

Therefore, our first task was to reduce the parameter space for our experiments by carefully selecting the most relevant storage stack parameters. This selection was done in close collaboration with several storage experts that have either contributed to storage stack designs or have spent years tuning storage systems in the field. We experimented with three popular file systems that span a range of designs and features. ■ **(1) Ext4** [12] is a popular file system that inherits a lot of internal structures from Ext3 [6] and FFS [26]) but enhances performance and scalability using extents and delayed allocation. ■ **(2) XFS** [35, 38] was initially designed for SGI's IRIX OS [38] and was later ported to Linux. It has attracted users' attention since the 90s thanks to its high performance on new storage devices and its high scalability regarding large files, large numbers of files, and large directories. XFS uses B+ trees for tracking free extents, indexing directory entries, and keeping track of dynamically allocated inodes. ■ **(3) Btrfs** [5, 31] is a complex file system that has seen extensive development since 2007 [31]. It uses copy-on-write (CoW), allowing efficient snapshots and clones. It has its own LVM and uses B-trees as its main on-disk data structure. These unique features are garnering attention and we expect Btrfs to gain even greater popularity in the future.

For the three file systems above we experimented with the following nine parameters. ■ **(1) Block size**. This is a group of contiguous sectors and is the basic unit of space allocation in a file system. Improper block size selection can reduce file system performance by orders of magnitude [18]. ■ **(2) Inode size**. This is one of the most basic on-disk structures of a file system [3]. It stores the metadata of a given file, such as its size, permissions, and the location of its data blocks. The inode is involved in nearly every I/O operation and thus plays a crucial role for performance, especially for metadata-intensive workloads. ■ **(3) Journal mode**. Journaling is

the write-ahead logging implemented by file systems for recovery purposes in case of power losses and crashes. In Ext4, three types of journaling modes are supported: *writeback*, *ordered*, and *journal* [13]. The *writeback* mode journals only metadata whereas the *journal* mode provides full data and metadata journaling. In *ordered* mode, Ext4 journals metadata only, but all data is forced directly out to the disk prior to its metadata being committed to the journal. There is a trade-off between file system consistency and performance, as journaling generally adds I/O overhead. In comparison, XFS implements metadata journaling, which is similar to Ext4's *writeback* mode, and there is no need for journaling in Btrfs because of its CoW nature. ■ **(4) Allocation Group (AG) count**. This parameter is specific to XFS which partitions its space into regions called Allocation Groups [38]. Each AG has its own data structures for managing free space and inodes within its boundaries. ■ **(5) Nodatacow** is a Btrfs mount-time option that turns the CoW feature on or off for data blocks. When data CoW is enabled, Btrfs creates a new version of an extent or a page at a newly allocated space [31]. This allows Btrfs to avoid any partial updates in case of a power failure. When data CoW is disabled, partially written blocks are possible on system failures. In Btrfs, *nodatacow* implies *nodatasum* and compression disabled. ■ **(6) Nodatasum** is a Btrfs mount-time option and when specified, it disables checksums for newly created files. Checksums are the primary mechanism used by modern storage systems to preserve data integrity [3], computed using hash functions such as SHA-1 or MD5. ■ **(7) atime Options**. These refer to mount options that control the inode access time. We experimented with *noatime* and *relatime* values. The *noatime* option tells the file system not to update the inode access time when a file data read is made. When *relatime* is set, atime will only be updated when the file's modification time is newer than the access time or atime is older than a defined interval (one day by default). ■ **(8) I/O scheduler**. The I/O Scheduler manages the submission of block I/O operations to storage devices. The choice of I/O scheduler can have a significant impact on the I/O stack performance [4]. We used the *noop*, *deadline*, and *Completely Fair Queuing (CFQ)* I/O schedulers. Briefly explained, the *noop* scheduler inserts all incoming I/O requests into a simple FIFO queue in order of arrival; the *deadline* scheduler associates a deadline with all I/O operations to prevent starvation of requests; and the *CFQ* scheduler try to provide a fair allocation of disk I/O bandwidth for all processes that requests I/O operations. ■ **(9) Storage device**. The underlying storage device plays an important role in nearly every I/O operation. We ran our experiments on three types of devices: two HDDs (SATA vs. SAS) and one (SATA) SSD.

File System	Parameter	Value Range
Ext4	Block Size	1024, 2048, 4096
	Inode Size	128, 512, 2048, 8192
	Journal Mode	data=journal, ordered, writeback
XFS	Block Size	1024, 2048, 4096
	Inode Size	256, 512, 1024, 2048
	AG Count	8, 32, 128, 512
Btrfs	Node Size	4096, 16384, 65536
	Special Options	nodatacow, nodatasum, default
All	atime Options	relatime, noatime
	I/O Scheduler	noop, deadline, cfq
	Storage Devices	HDD (SAS, SATA), SSD (SATA)

Table 2: List of parameters and value ranges.

Table 2 summarizes all parameters and the values used in our experiments.

Latin Hypercube Sampling. Reducing the parameter space to the most relevant parameters based on expert knowledge resulted in 1,782 unique configurations ("Expert Space" in Table 1). However, it would still take more than 1.5 years to complete the evaluation of every configuration in that space. To reduce the space further, we intelligently sampled it using *Latin Hypercube Sampling (LHS)*, a method often used to construct computer experiments in multi-dimensional parameter spaces [19,24]. LHS can help explore a search space and discover unexpected behavior among combinations of parameter values; this suited our needs here. In statistics, a *Latin Square* is defined as a two-dimensional square grid where each row and column have only one sample; *Latin Hypercube* generalizes this to multiple dimensions and ensures that each sample is the only one in the axis-aligned hyper-plane containing it [24]. Using LHS, we were able to sample 107 representative configurations from the Expert Space and complete the evaluation within 34 days of clock time (excluding lengthy analysis time). We believe this approach is a good starting point for a detailed characterization and understanding of performance variation in storage stacks.

4 Experimental Setup and Workloads

This section details our experimental setup, which used a variety of storage devices and workloads.

Hardware. Our experiments were conducted on four identical Dell PowerEdge R710 servers equipped with Intel Xeon quad-core 2.4GHz CPUs. To maintain realistically high ratio of the dataset size to the RAM size and ensure that our experiments produce enough I/O, we limited the RAM size on all machines to 4GB. Each server has three types of storage devices installed: (1) 250GB Fujitsu SATA HDD with 5,400 RPM, (2) 147GB Seagate SAS HDD with 15,000 RPM, and

Workload	Average File Size	# Files		Time(s)		I/O Size			Num. of Threads	R/W Ratio
		Default	Actual	Default	Actual	Read	Write	Append		
Webserver	16KB	1,000	640,000	60	800	WF	-	16KB	100	10:1
Fileserver	128KB	10,000	80,000	60	800	WF	WF	16KB	50	1:2
Mailserver	16KB	1,000	640,000	60	2,000	WF	-	16KB	16	1:1

Table 3: Filebench workload characteristics used in our experiments. WF stands for Whole-File read or write.

(3) 200GB Intel SATA SSD (MLC). This allowed us to evaluate the impact of devices on storage stack performance variation. On each device, we created a full-disk partition. The machines ran an Ubuntu 14.04.1 LTS system, with the kernel upgraded to version 4.4.12. We also updated *e2fsprogs*, *xfsprogs*, and *btrfs-progs* to latest version as of May, 2016.

Workloads. We used *Filebench* [14, 41] v1.5 to generate various workloads in our experiments. In each experiment, if not stated otherwise, we formatted and mounted the storage devices with a file system and then ran Filebench. We used the following three pre-configured Filebench macro-workloads: ■ **(1) Mailserver** emulates the I/O workload of a multi-threaded email server. It generates sequences of I/O operations that mimic the behavior of reading emails (open, read the whole file, and close), composing emails (open/create, append, close, and fsync) and deleting emails. It uses a flat directory structure with all the files in one single directory, and thus exercises the ability of file systems to support large directories and fast lookups. ■ **(2) Fileserver** emulates the I/O workload of a server that hosts users' home directories. Here, each thread represents a user, which performs create, delete, append, read, write, and stat operations on a unique set of files. It exercises both the metadata and data paths of the targeted file system. ■ **(3) Webserver** emulates the I/O workload of a typical static Web server with a high percentage of reads. Files (Web pages) are read sequentially by multiple threads (users); each thread appends to a common log file (Web log). This workload exercises fast lookups, sequential reads of small files and concurrent data and metadata management.

Table 3 shows the detailed settings of these workloads. All are set to Filebench default values, except for the number of files and the running time. As the average file size is an inherent property of a workload and should not be changed [41], the dataset size is determined by the number of files. We increased the number of files such that the dataset size is 10GB—2.5× the machine RAM size. By fixing the dataset size, we normalized the experiments' set-size and run-time, and ensured that the experiments run long enough to produce enough I/O. With these settings, our experiments exercise both in-memory cache and persistent storage devices [42].

We did not perform a separate cache warm-up phase in our experiments because in this study we were interested in performance variation that occurred *both* with cold and warm caches [42]. The default running time for Filebench is set to 60 seconds, which is too short to warm the cache up. We therefore conducted a "calibration" phase to pick a running time that was long enough for the cumulative throughput to stabilize. We ran each workload for up to 2 hours for testing purposes, and finally picked the running time as shown in Table 3. We also let Filebench output the throughput (and other performance metrics) every 10 seconds, to capture and analyze performance variation from a short-term view.

5 Evaluation

In this work we are characterizing and analyzing storage performance variation from a variety of angles. These experiments represent a large amount of data, and therefore, we first present the information with brief explanations, and in subsequent subsections we dive into detailed explanations. §5.1 gives an overview of performance variations found in various storage stack configurations and workloads. §5.2 describes a case study by using Ext4-HDD configurations with the Fileserver workload. §5.3 presents temporal variation results. Here, temporal variations consist of two parts: changes of throughput over time and latency variation.

5.1 Variation at a Glance

We first overview storage stack performance variation and how configurations and workloads impact its magnitude. We designed our experiments by applying the methodology described in §3. We benchmarked configurations from the Sample Space (see Table 1) under three representative workloads from Filebench. The workload characteristics are shown in Table 3. We repeated each experiment 10 times in a carefully-controlled environment in order to get unperturbed measurements.

Figure 1 shows the results as scatter plots broken into the three workloads: *Mailserver* (Figure 1(a)), *Fileserver* (Figure 1(b)), and *Webserver* (1(c)). Each symbol represents one storage stack configuration. We use squares for Ext4, circles for XFS, and triangles for Btrfs. Hollow symbols are SSD configurations, while filled symbols are for HDD. We collected the cumulative throughput for each run. As described in §2, we define the cumulative throughput as the average number of I/O operations completed per second throughout each experiment run. This can also be represented as *Throughput-800* for *Fileserver* and *Webserver*, and

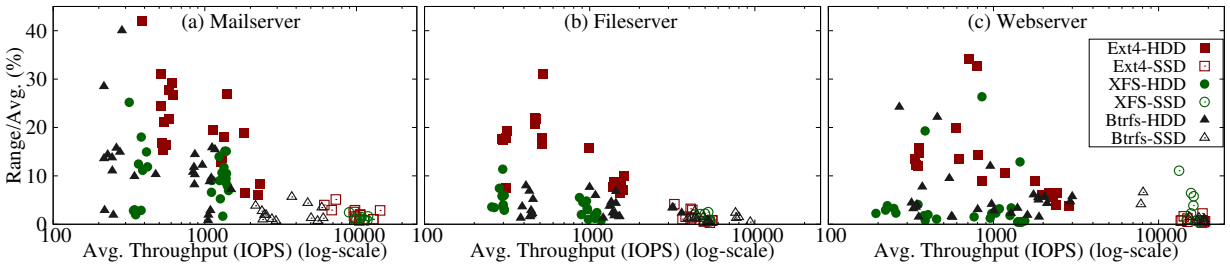


Figure 1: Overview of performance and its variation with different storage stack configurations under three workloads: (a) Maileserver, (b) Fileserver, and (c) Webserver. The X axis represents the mean of throughput over 10 runs; the Y axis shows the relative range of cumulative throughput. Ext4 configurations are represented with squares, XFS with circles, and Btrfs with triangles. HDD configurations are shown with filled symbols, and SSDs with hollow ones.

Throughput-2000 for *Mailserver*, as per our notation. In each subfigure, the Y axis represents the relative range of cumulative throughputs across the 10 runs. As explained in §2, here we use the relative range as the measure of variation. A higher relative range value indicates higher degree of variation. The X axis shows the mean cumulative throughput across the runs; higher values indicate better performance. Since performance for SSD configurations is usually much better than HDD configurations, we present the X axis in log_{10} scale.

Figure 1 shows that HDD configurations are generally slower in terms of throughput but show a higher variation, compared with SSDs. For HDDs, throughput varies from 200 to around 2,000 IOPS, and the relative range varies from less than 2% to as high as 42%. Conversely, SSD configurations usually have much higher throughput than HDDs, ranging from 2,000 to 20,000 IOPS depending on the workload. However, most of them exhibit variation less than 5%. The highest range for any SSD configurations we evaluated was 11%.

Ext4 generally exhibited the highest performance variation among the three evaluated file systems. For the *Mailserver* workload, most Ext4-HDD configurations had a relative range higher than 12%, with the highest one being 42%. The *Fileserver* workload was slightly better, with the highest relative range being 31%. Half of the Ext4-HDD configurations show variation higher than 15% and the rest between 5–10%. For *Webserver*, the Ext4-HDD configuration varies between 6–34%. All Ext4-SSD configurations are quite stable in terms of performance variation, with less than 5% relative range.

Btrfs configurations show a moderate level of variation in our evaluation results. For *Mailserver*, two Btrfs-HDD configurations exhibited 40% and 28% ranges of throughput, and all others remained under 15%. Btrfs was quite stable under the *Fileserver* workload, with the highest variation being 8%. The highest relative range value we found for Btrfs-HDD configurations under *Webserver* is 24%, but most of them were below 10%. Similar to Ext4, Btrfs-SSD configurations were also quite stable, with a maximum variation of 7%.

XFS had the least amount of variation among the three file systems, and is fairly stable in most cases, as others have reported before, albeit with respect to tail latencies [18]. For *Mailserver*, the highest variation we found for XFS-HDD configurations was 25%. In comparison, Ext4 was 42% and Btrfs was 40%. Most XFS-HDD configurations show variation smaller than 5% under *Fileserver* and *Webserver* workloads, except for one with 11% for *Fileserver* and three between 12–23% for *Webserver*. Interestingly, however, across all experiments for all three workloads conducted on SSD configurations, the highest variation was observed on one XFS configuration using the *Webserver* workload, which had a relative range value of 11%.

Next, we decided to investigate the effect of workloads on performance variation in storage stacks. Figure 2 compares the results of the same storage stack configurations under three workloads. These results were extracted from the same experiments shown in Figure 1. Although we show here only all Ext4-HDD configurations, we have similar conclusions for other file systems and for SSDs. The bars represent the relative range of 10 repeated runs, and correspond to the left Y1 axis. The average throughput of 10 runs for each configuration is shown as symbols, and corresponds to the right Y2 axis. The X axis consists of configuration details, and is formatted as the six-part tuple ⟨*block size - inode size - journal option - atime option - I/O scheduler - device*⟩. We can see that some configurations remain unstable in all workloads. For example, the configuration *2K-128-writeback-relatime-deadline-SATA* exhibited high performance variation (around 30%) under all three workloads. However, for some configurations, the actual workload played an important role in the magnitude of variation. For example, in the configuration *2K-2K-writeback-noatime-noop-SATA*, the *Mailserver* workload varies the most; but in the configuration *4K-512-ordered-relatime-noop-SATA*, the highest range of performance was seen on *Fileserver*. Finally, configurations with SAS HDD drives tended to have a much lower range variation but higher average throughput than

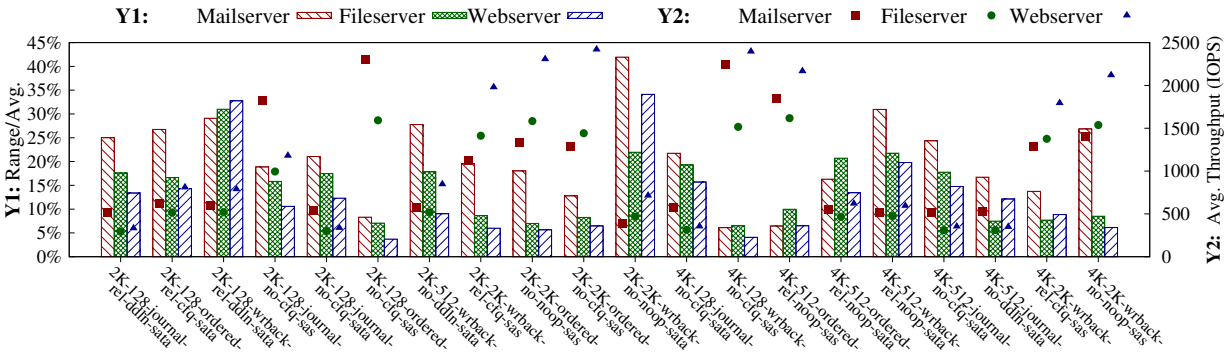


Figure 2: Storage stack performance variation with 20 sampled Ext4-HDD configurations under three workloads. The range is computed among 10 experiment runs, and is represented as bars corresponding to the Y1 (left) axis. The mean of throughput among the 10 runs is shown with symbols (squares, circles, and triangles), and corresponds to the Y2 (right) axis. The X axis represents configurations formatted by ⟨block size - inode size - journal - atime - I/O scheduler - device⟩.

SATA drives.

5.2 Case Study: Ext4

Identifying root causes for performance variation in the storage stack is a challenging task, even in experimental settings. Many components in a modern computer system are not isolated, with complex interactions among components. CPU, main memory, and secondary storage could all contribute to storage variation. Our goal was not to solve the variation problem completely, but to report and explain this problem as thoroughly as we could. We leave to future work to address these root causes from the source code level [44]. At this stage, we concentrated our efforts solely on benchmarking local storage stacks, and tried to reduce the variation to an acceptable level. In this section we describe a case study using four Ext4 configurations as examples. We focused on Ext4-HDD (SATA) here, as this combination of file systems and device types produced the highest variations in our experiments (see Figures 1 and 2).

Figure 3 shows results as two boxplots for the *Fileserver* workload, where each box plots the distribution of throughputs across the 10 runs, with the relative range shown below. The top border represents the 1^{st} quartile, the bottom border the 3^{rd} quartile, and the line in the middle is the median value. Whiskers show the maximum and minimum throughputs. We also plotted one dot for the throughput of each run, overlapping with the boxes but shifted to the right for easier viewing. The X axis represents the relative improvements that we applied based on our successive investigations and uncovering of root causes of performance variation, while the Y axis shows the cumulative throughput for each experiment run. Note that the improvement label is prefixed with a "+" sign, meaning that an additional feature was added to the previous configuration, cumulatively. For example, *+umount* actually indicates *baseline* + *no_lazy* + *umount*. We also added labels on the bottom of each subfigure showing the configuration details, formatted as

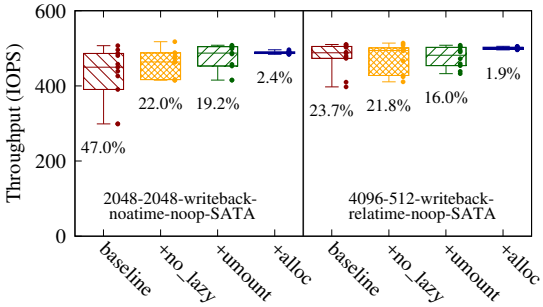


Figure 3: Performance variation for 2 Ext4-HDD configurations with several diagnoses. Each experiment is shown as one box, representing a throughput distribution for 10 identical runs. The top border line of each box marks the 1^{st} quartile; the bottom border marks the 3^{rd} quartile; the line in the middle is the median throughput; and the whiskers mark maximum and minimum values. The dots to the right of each box show the exact throughputs of all 10 runs. The percentage numbers below each box are the relative range values. The bottom label shows configuration details for each figure.

⟨*block size - inode size - journal option - atime option - I/O scheduler - device*⟩.

After addressing all causes we found, we were able to reduce the relative range of throughput in these configurations from as high as 47% to around 2%. In the rest of this section, we detail each root cause and how we addressed it.

Baseline. The first box for each subfigure in Figure 3 represents our original experiment setting, labeled *baseline*. In this setting, before each experimental run, we format and mount the file system with the targeted configuration. Filebench then creates the dataset on the mounted file system. After the dataset is created, Filebench issues the *sync* command to flush all dirty data and metadata to the underlying device (here, SATA HDD); Filebench then issues an *echo 3 > /proc/sys/vm/drop_caches* command, to evict non-dirty data and metadata from the page cache. Then, Filebench runs the Fileserver workload for a pre-defined amount of time (see

Table 3). For this baseline setting, both Ext4-HDD configurations show high variation in terms of throughput, with range values of 47% (left) and 24% (right).

Lazy initialization. The first contributor to performance variation that we identified in Ext4-HDD configurations is related to the lazy initialization mechanism in Ext4. By default, Ext4 does not immediately initialize the complete inode table. Instead, it gradually initializes it in the background when the created file system is first mounted, using a kernel thread called *ext4lazyinit*. After the initialization is done, the thread is destroyed. This feature speeds up the formatting process significantly, but also causes interference with the running workload. By disabling it during format time, we reduced the range of throughput from 47% to 22% for Configuration *2048-2048-writeback-noatime-noop-SATA*. This improvement is labelled *+no_lazy* in Figure 3.

Sync then umount. In Linux, when *sync* is called, it only guarantees to *schedule* the dirty blocks for writing: there is often a delay until all blocks are actually written to stable media [29, 39]. Therefore, instead of calling *sync*, we *umount* the file system each time after finishing creating the dataset and then *mount* it back, which is labelled as *+umount* in Figure 3. After applying this, both Ext4-HDD configurations exhibited even lower variation than the previous setting (disabling lazy initialization only).

Block allocation and layout. After applying the above improvements, both configurations still exhibited higher than 16% variations, which could be unacceptable in settings that require more predictable performance. This inspired us to try an even more strictly-controlled set of experiments. In the *baseline* experiments, by default we re-created the file system before each run and then Filebench created the dataset. We assumed that this approach would result in identical datasets among different experiment runs. However, block allocation is not a deterministic procedure in Ext4 [18]. Even given the same distribution of file sizes and directory width, and also the same number of files as defined by Filebench, multiple trials of dataset creation on a freshly formatted, clean file system did not guarantee to allocate blocks from the same or even near physical locations on the hard disk. To verify this, instead of re-creating the file system before each run, we first created the file system and the desired dataset on it. We then dumped out the entire partition image using *dd*. Then, before each run of Filebench, we used *dd* to restore the partition using the image, and mounted the file system back. This approach guaranteed an identical block layout for each run.

Figure 3 shows these results using *+alloc*. We can see that for both Ext4-HDD configurations, we were able to achieve around 2% of variation, which verified our hypothesis that block allocation and layout play an important role in the performance variation for Ext4-HDD configurations.

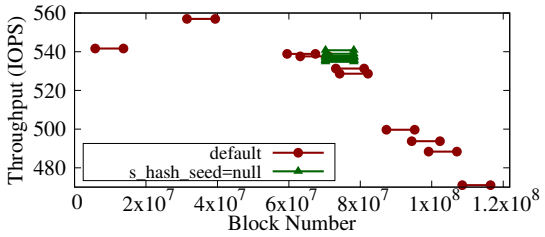


Figure 4: Physical blocks of allocated files in Ext4 under the Fileserver workload. The X axis represents the physical block number of each file in the dataset. Since the Fileserver workload consists of small files, and one extent per file, we use the starting block number for each file here. The Y axis is the final cumulative throughput for each experiment run. Note that the Y axis does not start from 0. Lines marked with solid circles are experiment runs with the default setting; lines with triangles represent experiment runs where we set the field s_hash_seed in Ext4s's superblock to null.

Storing the images of file systems using the *dd* command, however, could be too costly in practice, taking hours of clock time. We found a faster method to generate reproducible Ext4 layouts by setting the *s_hash_seed* field in Ext4's superblock to *null* before mounting. Figure 4 shows the distribution of physical blocks for allocated files in two sets of *Fileserver* experiments on Ext4. This workload consists of only small files, resulting in exactly one extent for each file in Ext4, so we used the starting block number (X axis) to represent the corresponding file. The Y axis shows the final cumulative throughput for each experiment run. Here the lines starting and ending with solid circles are 10 runs from the experiment with the full-disk partition. The lines with triangles represent the same experiments, but here we set the *s_hash_seed* field in Ext4's superblock to *null*. We can see that files in each experiment run are allocated into one cluster within a small range of physical block numbers. In most cases, experimental runs with their dataset allocated near the outer tracks of disks, which correspond to smaller block numbers, tend to produce higher throughput. As shown in Figure 4, with the default setting, datasets of 10 runs clustered in 10 different regions of the disk, causing high throughput variation across the runs. By setting the Ext4 superblock parameter *s_hash_seed* to *null*, we can eliminate the non-determinism in block allocation. This parameter determines the group number of top-level directories. By default, *s_hash_seed* is randomly generated during format time, resulting in distributing top-level directories all across the LBA space. Setting it to *null* forces Ext4 to use the hard-coded default values, and thus the top-level directory in our dataset is allocated on the same position among different experiment runs. As we can see from

Figure 4, for the second set of experiments, the ranges of allocated block numbers in all 10 experiment runs were exactly the same. When we set the *s_hash_seed* parameter to *null*, the relative range of throughput dropped from and 16.6% to 1.1%. Therefore, setting this parameter could be useful when users want stable benchmarking results from Ext4.

In addition to the case study we conducted on Ext4-HDD configurations, we also observed similar results for Ext4 on other workloads, as well as for Btrfs. For two of the Btrfs-HDD configurations, we were able to reduce the variation to around 1.2%, by using *dd* to store the partition image. We did not try to apply any improvements on XFS, since most of its configurations were already quite stable (in terms of cumulative throughput) even with the *baseline* setting, as shown in Figure 1.

5.3 Temporal Variation

In Sections 5.1 and 5.2, we mainly presented and analyzed performance variation among repeated runs of the same experiment, and only in terms of throughput. Variation can actually manifest itself in many other ways. We now focus our attention on *temporal* variations in storage stack performance—the variation related to time. §5.3.1 discusses temporal throughput variations and §5.3.2 focuses on latency variations.

5.3.1 Throughput over Time

After finding variations in cumulative throughputs, we set out to investigate whether the performance variation changes over time within single experiment run.

To characterize this, we calculated the throughput within a small time window. As defined in §2, we denote throughput with window size of N seconds as *Throughput-N*. Figure 5 shows the *Throughput-120* value (Y axis) over time (X axis) for Btrfs-HDD, XFS-HDD, and Ext4-HDD configurations using the *Fileserver* workload.

Here we use a window size of 120 seconds, meaning that each throughput value is defined as the average number of I/O operations completed per second with the latest 120 seconds. We also investigated other window sizes, which we discuss later. The three configurations shown here exhibited high variations in the experiments discussed in §5.1. Also, to show the temporal aspect of throughput better, we extended the running time of this experiment set to 2 hours, and we repeated each experiment 10 times. Two lines are plotted connecting the maximum and minimum throughput values among 10 runs. We fill in colors between two lines, this producing a color band: green for Btrfs, red for Ext4, and blue for XFS. The line in the middle of each band is plotted by connecting the average *Throughput-120* value among 10 runs. We observed in Figure 1(b) that for the *Fileserver* workload, Ext4-HDD configurations generally exhibited higher variations than XFS-HDD or Btrfs-HDD configurations in terms of final cumulative throughput. However, when it comes to *Throughput-120* values, Figure 5 leads to some different conclusions. The Ext4-HDD configuration still exhibited high variation in terms of short-term throughout across the 2 hours of experiment time, while the Btrfs-HDD configuration is much more stable. Surprisingly, the XFS-HDD configuration has higher than 30% relative range of *Throughput-120* values for most of the experiment time, while its range for cumulative throughput is around 2%. This suggests that XFS-HDD configurations might exhibit high variations with shorter time windows, but produces more stable results in longer windows. It also indicates that the choice of window sizes matters when discussing performance variations.

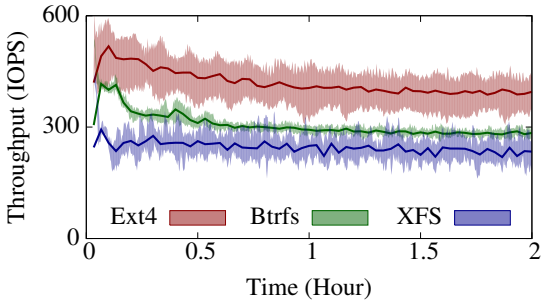
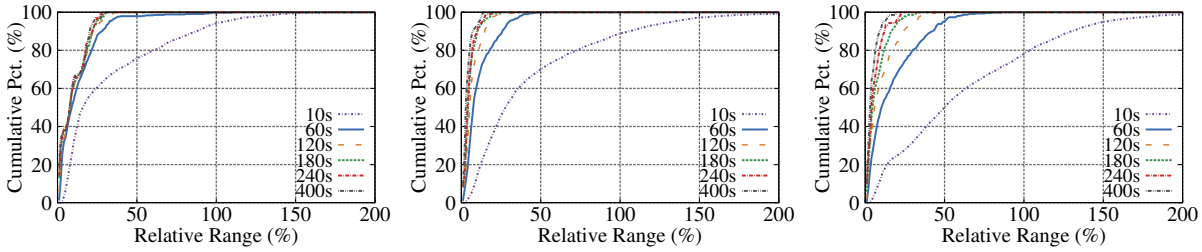


Figure 5: Throughput-120 over time for Btrfs, XFS, and Ext4 HDD configurations under the Fileserver workload. Each configuration was evaluated for 10 runs. Two lines were plotted connecting maximum and minimum throughput values among 10 runs. We fill in colors between two lines, green for Btrfs, red for Ext4, and blue for XFS. We also plotted the average Throughput-120 among 10 runs as a line running through the band. The maximum relative range values of Throughput-120 for Ext4, Btrfs, and XFS are 43%, 23%, and 65%, while the minimum values are 14%, 2%, and 7%, respectively.

We can see from the three average lines in Figure 5 that performance variation exists even within one single run—the short-term throughput varies as the experiment proceeds. For most experiments, no matter what the file system type is, performance starts slow and climbs up quickly in the beginning phase of experiments. This is because initially the application is reading cold data and metadata from physical devices into the caches; once cached, performance improves. Also, for some period of time, dirty data is kept in the cache and not yet flushed to stable media, delaying any impending slow writes. After an initial peak, performance begins to drop rapidly and then declines steadily. This is because the read performance already reached its peak and cached dirty data begins to be flushed out to slower media. Around several minutes in, performance begins to stabilize, as we see the throughput lines flatten.

The unexpected difference in variations for short-term



(a) Ext4 (b) Btrfs (c) XFS

Figure 6: CDFs for relative range of throughput under Fileserver workload with different window sizes. For window size N, we calculated the relative range values of throughput for all configurations within each file system type, and then plotted the corresponding CDF.

and cumulative throughput of XFS-HDD configurations lead us to investigate the effects of the time window size on performance variations. We calculated the relative range of throughput with different window sizes for all configurations within each file system type. We present the CDFs of these range values in Figure 6. For example, we conducted experiments on 39 Btrfs configurations. With a window size of 60 seconds and total running time of 800 seconds, the corresponding CDF for Btrfs is based on $39 \times \frac{800}{60} = 507$ relative range values. We can see that Ext4's unstable configurations are largely unaffected by the window size. Even with *Throughput-400*, around 20% of Ext4 configurations produce higher than 20% variation in terms of throughput. Conversely, the range values for Btrfs and XFS are more sensitive to the choice of window size. For XFS, around 40% of the relative range values for *Throughput-60* are higher than 20%, whereas for *Throughput-400*, nearly all XFS values fall below 20%. This aligns with our early conclusions in §5.1 that XFS configurations are relatively stable in terms of cumulative throughput, which is indeed calculated based on a window size of 800 seconds; whereas XFS showed the worst relative range for *Throughput-60*, it stabilized quickly with widening window sizes, eventually beating Ext4 and Btrfs.

All the above observations are based on the throughput within a certain window size. Another approach is to characterize the *instant throughput* within an even shorter period of time. Figure 7 shows the instantaneous throughput over time for various configurations under the *Fileserver* workload. We collected and calculated the throughput every 10 seconds. Therefore we define instantaneous throughput as the average number of I/O operations completed in the past 10 seconds. This is actually *Throughput-10* in our notation. We normalize this by dividing each value by the maximum instantaneous throughput value for each run, to compare the variation across multiple experimental runs. The X axis still shows the running time.

We picked one illustrative experiment run for each configuration (Ext4-HDD, XFS-HDD, Btrfs-HDD, and Ext4-SSD). We can see from Figure 7 that for all configurations, instantaneous performance fluctuated a lot throughout the experiment. For all three HDD configurations, the variation is even higher than 80% in the first 100 seconds. The magnitude for variation reduces later in the experiments, but stays around 50%.

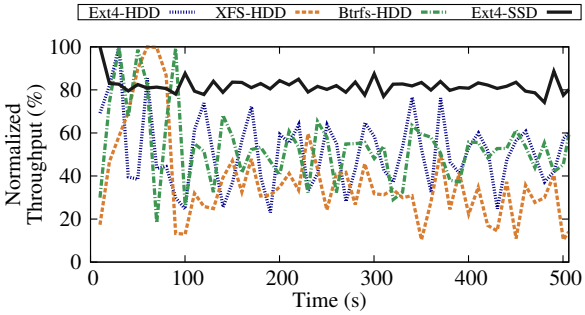
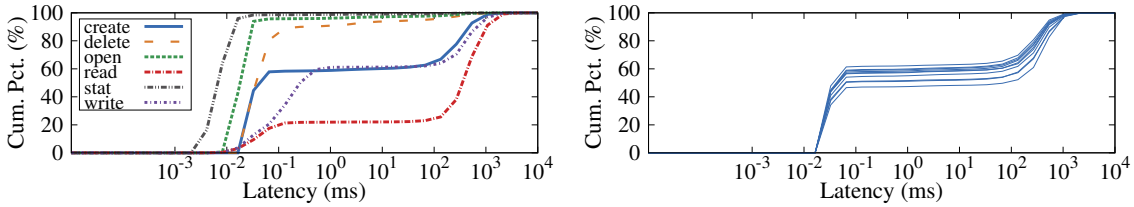


Figure 7: Normalized instantaneous throughput (Throughput-10) over time for experiments with various workloads, file systems, and devices. The Y axis shows the normalized values divided by the maximum instantaneous throughput through the experiment. Only the first 500s are presented for brevity.

The throughput spikes occur nearly every 30 seconds, which could be an indicator that the performance variation in storage stacks is affected by some cyclic activity (e.g., kernel flusher thread frequency). For SSD configurations, the same up-and-down pattern exists, although its magnitude is much smaller than for HDD configurations, at only around 10%. This also confirms our findings from §5.1 that SSDs generally exhibit more stable behavior than HDDs.

5.3.2 Latency Variation

Another aspect of performance variation is latency, defined as the time taken for each I/O request to complete. Much work has been done in analyzing and taming long-tail latency in networked systems [20, 22] (where 99.9^{th} percentile latency is orders of magnitude worse than the median), and also in local storage systems [18]. Throughout our experiments, we found out that long-tail latency is not the only form of latency variation; there are other factors that can impact the latency distribution for I/O operations.



(a) CDFs of operations within one single experiment run

(b) CDFs of *create* operation among repeated experiment runs

Figure 8: Latency CDF of one Ext4-HDD configuration under Fileserver workload.

A *Cumulative Distribution Function (CDF)* is a common approach to present latency distribution. Figure 8(a) shows the latency CDFs for 6 I/O operations of one Ext4-HDD configuration under the *Fileserver* workload. The X axis represents the latency in log_{10} scale, while the Y axis is the cumulative percentage. We can see that for any one experimental run, operations can have quite different latency distribution. The latencies for *read*, *write*, and *create* form two clusters. For example, about 20% of the *read* operation has less than 0.1ms latency while the other 80% falls between 100ms and 4s. Conversely, the majority of *stat*, *open*, and *delete* operations have latencies less than 0.1ms.

The I/O operation type is not the only factor that impacts the latency distribution. Figure 8(b) presents 10 CDFs for *create* from 10 repeated runs of the same experiment. We can see for the 60^{th} percentile, the latency can vary from less than 0.1ms to over 100ms.

Different I/O operations and their latencies impact the overall workload throughput to a different extent. With the empirical data that we collected—per-operation latency distributions and throughput—we were able to discover correlations between the speed of individual operations and the throughput. We first defined a metric to quantify the difference between two latency distributions. We chose to use the Kolmogorov-Smirnov test (K-S test), which is commonly used in statistics to determine if two datasets differ significantly [43]. For two distributions (or discrete dataset), the K-S test uses the maximum vertical deviation between them as the distance. We further define the range for a set of latency distributions as the maximum distance between any two latency CDFs. This approach allows us to use only one number to represent the latency variation, as with throughput. For each operation type, we calculated its range of latency variation for each configuration under all three workloads. We then computed the Pearson Correlation Coefficient (PCC) between the relative range of throughput and the range of latency variation.

Figure 9 shows our correlation results. The PCC value for any two datasets is always between [-1,+1], where +1 means total positive correlation, 0 indicates no correlation, and –1 means total negative correlation. Generally, any two datasets with PCC values higher than 0.7 are considered to have a strong positive correlation [32], which we show in Figure 9 with a horizontal dashed red line. The Y axis represents the PCC value while the X axis is the label for each operation. We separate workloads with vertical solid lines. As most SSD configurations are quite stable in terms of performance, we only considered HDD configurations here. For Ext4 configurations, *open* and *read* have the highest PCC values on both *Mailserver* and *Webserver* workloads; however, on *Fileserver*, *open* and *stat* have the strongest correlation. These operations could possibly be the main contributors to performance variation on Ext4-HDD configurations under each workload; such operations would represent the first ones one might tackle in the future to help stabilize Ext4's performance on HDD. In comparison, *write* has a PCC value of only around 0.2, which indicates that it may not contribute much to the performance variation. Most operations show PCC values larger than 0.4, which suggest weak correlation. This is possibly because I/O operations are not completely independent with each other in storage systems.

For the same workload, different file systems exhibit different correlations. For example, under the *Webserver* workload, Ext4 show strong correlation on both *read* and *open*; but for XFS, *read* shows a stronger correlation than *open* and *write*. For Btrfs, no operation had a strong correlation with the range of throughput, with only *read* showing a moderate level of correlation.

Although such correlations do not always imply direct causality, we still feel that this correlation analysis sheds light on how each operation type might contribute to the overall performance variation in storage stacks.

6 Related Work

To the best of our knowledge, there are no systematic studies of performance variation of storage stacks. Most previous work focuses on long-tail I/O latencies. Tarasov et al. [40] observed that file system performance could be sensitive to even small changes in running workloads. Arpaci-Dusseau [2] proposed an I/O programming environment to cope with performance variations in clustered platforms. Worn-out SSDs exhibit high latency variations [10]. Hao et al. [16] studied device-level performance stability, for HDDs and SSDs.

For long-tail latencies of file systems, He et al. [18] developed Chopper, a tool to explore a large input space

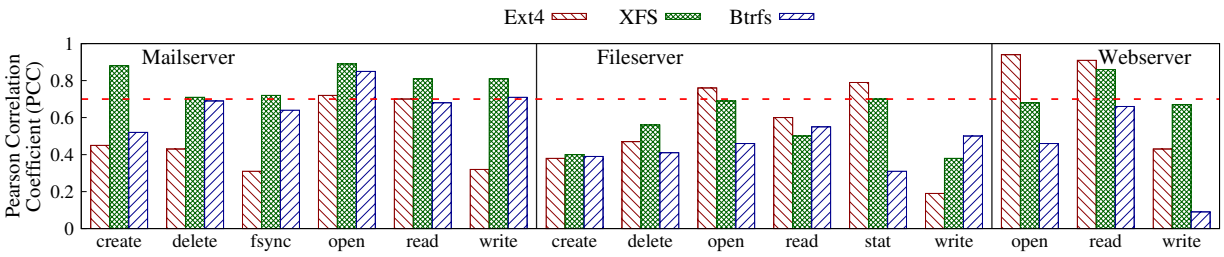


Figure 9: Pearson Correlation Coefficient (PCC) between throughput range and operation types, for three workloads and three file systems. The horizontal dashed red line at Y=0.7 marks the point above which a strong correlation is often considered to exist.

of file system parameters and find behaviors that lead to performance problems; they analyzed long-tail latencies relating to block allocation in Ext4. In comparison, our paper's goal is broader: a detailed characterization and analysis of several aspects of storage stack performance variation, including devices, block layer, and the file systems. We studied the variation in terms of both throughput and latency, and both spatially and temporally. Tail latencies are common in network or cloud services [9, 22]: several tried to characterize and mitigate their effects [17, 20, 37, 48], as well as exploit them to save data center energy [45]. Li et al. [22] characterized tail latencies for networked services from the hardware, OS, and application-level sources. Dean and Barroso [9] pointed out that small performance variations could affect a significant fraction of requests in large-scale distributed systems, and can arise from various sources; they suggested that eliminating all of them in large-scale systems is impractical. We believe there are possibly many sources of performance variation in storage systems, and we hope this work paves the way for discovering and addressing their impacts.

7 Conclusion

In this work we provided the first systematic study on performance variation in benchmarking a modern storage stack. We showed that variation is common in storage stacks, although its magnitude depends heavily on specific configurations and workloads. Our analysis revealed that block allocation is a major cause of performance variation in Ext4-HDD configurations. From the temporal view, the magnitude of throughput variation also depends on the window size and changes over time. Latency distribution for the same operation type could also vary even over repeated runs of the same experiment. We quantified the correlation between performance and latency variations using a novel approach. Although most of our observations are made in experimental settings, we believe they are a useful step towards a thorough understanding of stability issues in storage stacks of production systems. In conclusion, we list three best practices for people either benchmarking storage systems or dealing with performance variations in real systems. The goal here is not to "teach," but rather provide some guidelines to the best of our knowledge. ■ **(1)** Performance variation is a complex issue, and could be caused and affected by various factors: the file system, configurations of the storage system, the running workload, or even the time window for quantifying the performance. ■ **(2)** Non-linearity is inherent in complex storage systems. It could lead to large differences in results, even in well-controlled experiments; conclusions drawn from these could be misleading or even wrong. ■ **(3)** Disable all lazy initialization and any background activities, if any, while formatting, mounting, and experimenting on file systems.

Future Work. We believe that more work still needs to be done to more fully understand the causes of different types of variation and especially to address them. All experiments in this paper were conducted on freshly-formatted file systems, and thus we only focused on performance variations in such systems. We did not analyze aged file systems, a subject of our future work. We plan to expand our parameter search space (e.g., compression options in Btrfs [31]). Alas, Filebench currently creates files by filling them with 0s, so first we have to make Filebench output data with controlled compression ratios. We plan to use other benchmarking tools such as SPEC SFS 2014 [36] which comes with several pre-configured and realistic workloads. We plan to expand the study to new types of devices such as PCM [15, 47] and SMRs [1], which have their own complex behavior such as worse tail latencies due to internal garbage collection [9, 10]. In the meanwhile, we will tackle other storage layers (LVM, RAID) and networked/distributed file systems. Finally, We plan to make all of our datasets and sources public. This includes not only the data from this work, but also a much larger dataset we continue to collect (now over two years).

Acknowledgments

We thank the anonymous FAST reviewers and our shepherd Remzi Arpaci-Dusseau for their valuable comments; and to Ted Ts'o for his assitance in understanding Ext4's behavior. This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; NSF awards CNS-1251137, CNS-1302246, CNS-1305360, and CNS-1622832; and ONR award 12055763.

References

[1] Abutalib Aghayev, Mansour Shafaei, and Peter Desnoyers. Skylight—a window on shingled disk operation. *Trans. Storage*, 11(4):16:1–16:28, October 2015.

[2] R. H. Arpaci-Dusseau, E. Anderson, N. T., D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with river: making the fast case common. In *Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, May 1999.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[4] D. Boutcher and A. Chandra. Does virtualization make disk scheduling passé? In *Proceedings of the 1st USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '09)*, October 2009.

[5] BTRFS. *http://btrfs.wiki.kernel.org/*.

[6] M. Cao, T. Y. Ts'o, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the Art: Where we are with the Ext3 filesystem. In *Proceedings of the Linux Symposium*, Ottawa, ON, Canada, July 2005.

[7] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding latency variation in modern DRAM chips: Experimental characterization, analysis, and optimization. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS'16, pages 323–336, New York, NY, USA, 2016. ACM.

[8] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209, 2014.

[9] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[10] Peter Desnoyers. Empirical evaluation of nand flash memory performance. In *HotStorage '09: Proceedings of the 1st Workshop on Hot Topics in Storage*. ACM, 2009.

[11] Nosayba El-Sayed, Ioan A. Stefanovici, George Amvrosiadis, Andy A. Hwang, and Bianca Schroeder. Temperature management in data centers: Why some (might) like it hot. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'12, pages 163–174, New York, NY, USA, 2012. ACM.

[12] Ext4. *http://ext4.wiki.kernel.org/*.

[13] Ext4 Documentation. *https://www.kernel.org/doc/Documentation/filesystems/ext4.txt*.

[14] Filebench, 2016. *https://github.com/filebench/filebench/wiki*.

[15] H. Kim and S. Seshadri and C. L. Dickey and L. Chiu. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 33–45, Berkeley, CA, 2014. USENIX.

[16] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The tail at store: a revelation from millions of hours of disk and ssd deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 263–276, 2016.

[17] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 161–175, New York, NY, USA, 2015. ACM.

[18] 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 Conference on File and Storage Technologies*, FAST'15, pages 119–133, Berkeley, CA, USA, 2015. USENIX Association.

[19] Ronald L Iman, Jon C Helton, James E Campbell, et al. An approach to sensitivity analysis of computer models, part 1. introduction, input variable selection and preliminary variable assessment. *Journal of quality technology*, 13(3):174–183, 1981.

[20] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. Predictive parallelization: Taming tail latencies in web search. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR'14, pages 253–262, New York, NY, USA, 2014. ACM.

[21] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561–2573, 2014. Special Issue on Perspectives on Parallel and Distributed Processing.

[22] 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 ACM Symposium on Cloud Computing*, SoCC'14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[23] Chieh-Jan Mike Liang, Jie Liu, Liqian Luo, Andreas Terzis, and Feng Zhao. RACNet: A high-fidelity data center sensing network. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys'09, pages 15–28, New York, NY, USA, 2009. ACM.

[24] W. J. Conover M. D. McKay, R. J. Beckman. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[25] Pradipta De Vijay Mann and Umang Mittaly. Handling OS jitter on multicore multithreaded systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2009 IEEE International*, IPDPS'09, pages 1–12. IEEE, 2009.

[26] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[27] Sun Microsystems. Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System White Paper. *www.sun.com/servers/hpc/docs/lustrefilesystem_wp.pdun.com/servers/hpc/docs/lustrefilesystem_wp.pdf*, December 2007.

[28] Alessandro Morari, Roberto Gioiosa, Robert W Wisniewski, Francisco J Cazorla, and Mateo Valero. A quantitative analysis of OS noise. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, IPDPS'11, pages 852–863. IEEE, 2011.

[29] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 1–14, Seattle, WA, November 2006. ACM SIGOPS.

[30] OpenStack Swift. *http://docs.openstack.org/developer/swift/*.

[31] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[32] Richard P Runyon, Kay A Coleman, and David J Pittenger. *Fundamentals of behavioral statistics* . McGraw-Hill, 2000.

[33] 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.

[34] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, pages 231–244, Monterey, CA, January 2002. USENIX Association.

[35] SGI. XFS Filesystem Structure. *http://oss.sgi.com/projects/xfs/papers/xfs_filesystem_structure.pdf*.

[36] SPEC SFS® 2014. *https://www.spec.org/sfs2014/*.

[37] 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 USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 513–527, Berkeley, CA, USA, 2015. USENIX Association.

[38] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 1–14, San Diego, CA, January 1996.

[39] sync(8) - Linux man page. *https://linux.die.net/man/8/sync*.

[40] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of HotOS XIII:The 13th USENIX Workshop on Hot Topics in Operating Systems*, Napa, CA, May 2011.

[41] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[42] Vasily Tarasov, Zhen Cao, Ming Chen, and Erez Zadok. The dos and don'ts of file system benchmarking. *FreeBSD Journal*, January/February, 2016.

[43] Olivier Thas. *Comparing distributions*. Springer, 2010.

[44] A. Traeger, I. Deras, and E. Zadok. DARC: Dynamic analysis of root causes of latency distributions. In *Proceedings of the 2008 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2008)*, pages 277–288, Annapolis, MD, June 2008. ACM.

[45] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. TimeTrader: Exploiting latency tail to save datacenter energy for online search. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO'48, pages 585–597, New York, NY, USA, 2015. ACM.

[46] S. Weil, S. Brandt, E. Miller, 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 (OSDI 2006)*, pages 307–320, Seattle, WA, November 2006. ACM SIGOPS.

[47] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, Dec 2010.

[48] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 329–342, Berkeley, CA, USA, 2013. USENIX Association.

Enlightening the I/O Path: A Holistic Approach for Application Performance

Sangwook Kim[†§], Hwanju Kim[§*], Joonwon Lee[§], Jinkyu Jeong[§]
[†]*Apposha,* [§]*Sungkyunkwan University*
sangwook@apposha.io, hwandori@gmail.com, joonwon@skku.edu, jinkyu@skku.edu

Abstract

In data-intensive applications, such as databases and key-value stores, reducing the request handling latency is important for providing better data services. In such applications, I/O-intensive background tasks, such as checkpointing, are the major culprit in worsening the latency due to the contention in shared I/O stack and storage. To minimize the contention, properly prioritizing I/Os is crucial but the effectiveness of existing approaches is limited for two reasons. First, statically deciding the priority of an I/O is insufficient since high-priority tasks can wait for low-priority I/Os due to *I/O priority inversion*. Second, *multiple independent layers* in modern storage stacks are not holistically considered by existing approaches which thereby fail to effectively prioritize I/Os throughout the I/O path.

In this paper, we propose a *request-centric I/O prioritization* that dynamically detects and prioritizes I/Os delaying request handling at all layers in the I/O path. The proposed scheme is implemented on Linux and is evaluated with three applications, PostgreSQL, MongoDB, and Redis. The evaluation results show that our scheme achieves up to 53% better request throughput and 42× better 99^{th} percentile request latency (84 ms vs. 3581 ms), compared to the default configuration in Linux.

1 Introduction

In data-intensive applications, such as databases and key-value stores, the response time of a client's *request* (e.g., key-value PUT/GET) determines the level of application performance a client perceives. In this regard, many applications are structured to have two types of tasks: *foreground tasks*, which perform essential work for handling requests, and *background tasks*, which conduct I/O-intensive internal activities, such as checkpointing [31, 14, 3], backup [11, 9, 24, 18], compaction [21, 34, 38, 13], and contents filling [36]. The main reason for this form of structuring is to reduce request handling latency by taking off the internal activities from the critical path of request execution. However, background tasks are still interfering foreground tasks since they inherently share the I/O path in a storage stack. For example, background checkpointing in relational database has known to hinder delivering low and predictable transaction latency, but the database and operating system (OS) communities have no reasonable solution despite their collaborative efforts [12].

The best direction to resolve this problem in OS is to provide an interface to specify I/O priority for a differentiated storage I/O service. Based on this form of OS support, two important issues should be addressed: 1) deciding which I/O should be given high priority, and 2) effectively prioritizing high priority I/Os along the I/O path. The conventional approaches for classifying I/O priorities are I/O-centric and task-centric. These approaches statically assign high priority to a specific type of I/O (e.g., synchronous I/O [30, 2, 32]) and to I/Os issued by a specific task (e.g., interactive task [42]). This I/O priority is typically enforced at the block-level scheduler or at several layers [42, 43].

The previous approaches, however, have limitations in achieving high and consistent application performance. First, they do not holistically consider *multiple independent layers* including caching, file system, block, and device layers in modern storage stacks. Missing I/O prioritization in any of the layers can degrade application performance due to delayed I/O processing in such layers (Section 2.1). Second, they do not address the *I/O priority inversion* problem caused by runtime dependencies among concurrent tasks and I/Os. Similar to the priority inversion problem in CPU scheduling [35], low-priority I/Os (e.g., asynchronous I/Os and background I/Os) sometimes can significantly delay the progress of a high-priority foreground task, thereby inverting I/O priority (Section 2.2). More seriously, I/O priority inversions can occur across different layers in a storage stack.

[*]Currently at Dell EMC

Due to these limitations, existing approaches are limited in effectively prioritizing I/Os and result in suboptimal performance.

In this paper, we introduce a *request-centric I/O prioritization* (or RCP for short) that holistically prioritizes *critical I/Os* (i.e., performance-critical I/Os) over non-critical ones along the I/O path; we define a critical I/O as an I/O in the critical path of request handling regardless of its I/O type and submitting task. Specifically, our scheme identifies foreground tasks by exposing an API and gives critical I/O priority to the foreground tasks (Section 4.1). To handle I/O priority inversions, critical I/O priority is dynamically assigned to a task or an outstanding I/O on which a foreground task depends to make progress (Section 4.2). Then, each layer in the I/O path is adapted to prioritize the critical I/Os and to support I/O priority inheritance (Section 4.3). We also resolve an I/O priority inversion caused by a transitive dependency, which is a chain of dependencies involving multiple tasks (Section 4.4).

As a prototype implementation, we enlightened the I/O path of the Linux kernel. Specifically, in order to accurately identify I/O criticality, we implemented the I/O priority inheritance to blocking-based synchronization methods (e.g., mutex) in the Linux kernel. Based on the identified I/O criticality, we made the Linux caching layer, ext4 file system, and the block layer understand and enforce I/O criticality. Based on the prototype, we evaluated our scheme using PostgreSQL [10], MongoDB [8], and Redis [22] with TPC-C [17] and YCSB [25] benchmarks. The evaluation results have shown that our scheme effectively improves request throughput and tail latency (99.9^{th} percentile latency) by about 7–53% and 4.4–20×, respectively, without penalizing background tasks, compared to the default configuration in Linux.

2 Motivation and Related Work

Background I/O-intensive tasks, such as checkpointing and compaction, are problematic for achieving the high degree of application performance. We illustrate this problem by running the YCSB [25] benchmark against MongoDB [8] document store on a Linux platform with two HDDs each of which is allocated for data and journal, respectively; see Section 7 for the details. As shown in Figure 1, application performance represented as operations per second is highly fluctuated with the CFQ [2], the default I/O scheduler in Linux, mainly due to the contention incurred by periodic checkpointing (60 seconds by default) [1]. Assigning low priority (idle-priority [7] in CFQ) to the checkpoint task using the existing interface,

[1]The interference is not exactly periodic because the checkpointing occurs 60 seconds after the completion of the previous checkpointing.

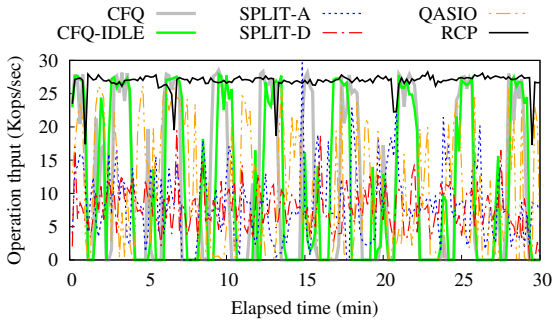


Figure 1: **Limitation of existing approaches.** *Update-heavy workload in YCSB is executed against MongoDB.*

denoted as CFQ-IDLE, is also ineffective in alleviating the performance interference. Moreover, Split-AFQ (SPLIT-A) and Split-Deadline (SPLIT-D), the state-of-the-art cross-layer I/O schedulers [43], also cannot provide consistent application performance even though the checkpoint thread is given lower priority than foreground ones; adjusting the parameters in the SPLIT-A/D (e.g., `fsync()` deadline) did not show any noticeable improvement. Likewise, QASIO [32], which tries to eliminate I/O priority inversions, also shows frequent drops in application performance.

The root causes of this undesirable result in the existing I/O prioritization schemes are twofold. First, the existing schemes do not fully consider multiple independent layers including caching, file system, and block layers in modern storage stacks. Prioritizing I/Os only in one or two layers of the I/O path cannot achieve proper I/O prioritization for foreground tasks. Second and more importantly, the existing schemes do not address the *I/O priority inversion* problem caused by runtime dependencies among concurrent tasks and I/Os. I/O priority inversions can occur across different I/O stages in multiple layers due to transitive dependencies. As shown by RCP in Figure 1, the cliffs in application throughput can be significantly mitigated if the two challenges are addressed. In the rest of this section, we detail the two challenges from the perspective of application performance and discuss existing approaches.

2.1 Multiple Independent Layers

In modern OSes, a storage I/O stack is comprised of multiple and independent layers (Figure 2). A caching layer first serves reads if it has the requested block and it buffers writes until they are issued to a lower layer. If a read miss occurs or a writeback of buffered writes is required, a file system generates block I/O requests and passes them to a block layer. Then, the block layer admits an I/O request into a block-level queue and schedules a queued I/O request to dispatch to a storage device. Finally, a storage device admits an I/O command received from a host into a device-internal queue and

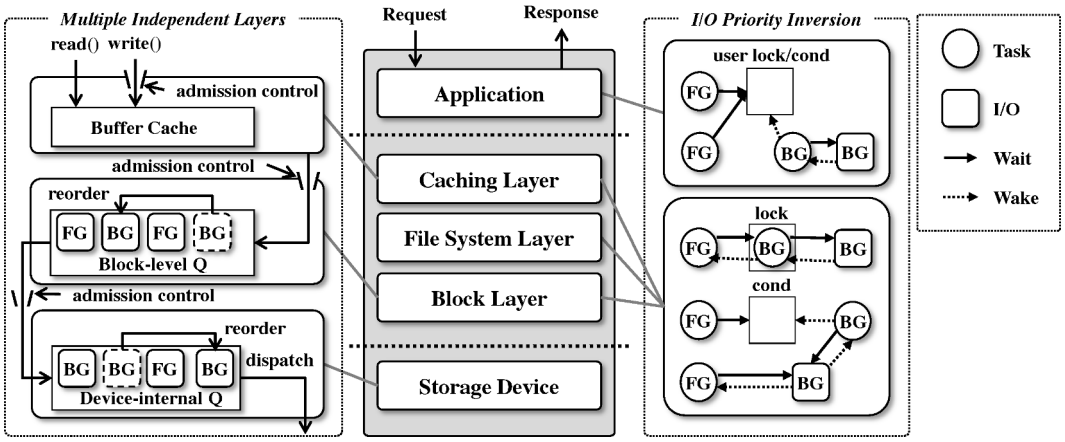


Figure 2: **Challenges in modern storage stacks.** *FG in a circle and a box means a foreground task and an I/O, respectively. Likewise, BG in a circle and a box indicates a background task and an I/O each.*

schedules a queued command to an internal device controller. Though this form of layering with abstraction is an essential part in computer systems for interoperability and independent innovations across the layers, it makes effective I/O prioritization strenuous because each layer has independent policy with limited information.

In the caching layer, priority-agnostic admission control can harm the application performance. Modern OSes, such as Linux [26] and Windows [40], control the admission of buffered writes based on the dirty ratio of available memory for ensuring system stability. However, since a single system-wide dirty ratio is applied to all tasks, a foreground task can be blocked even if most dirty pages are made by background tasks. Giving higher I/O priority to the foreground task is ineffective unless I/O priority is applied to the dirty throttling.

A file system forces specific ordering of writes for consistent updates [39, 41, 19], thereby complicating effective I/O prioritization. For example, ext4 file system, which is the default for most Linux distributions, entangles buffered writes into a single, global, compound transaction that needs to be durable atomically. Since a file system transaction contains dirty pages from any file written by any task, a foreground task calling `fsync()` should wait for the completion of I/Os issued not only by itself, but also by a background task.

In the block layer, many modern block I/O schedulers already reflect the priority of I/Os in their scheduling. However, priority-agnostic admission control can also degrade the application performance. Typically, the size of a block-level queue is limited to restrict memory usage and to control disk congestion [42]. In this case, a burst of background I/Os can significantly delay the processing of a foreground I/O by quickly filling the available slots in a block-level queue. The existing priority schedulers cannot help to mitigate this problem because they have no control of submitted I/Os that are not yet entered the block-level queues.

Even after a foreground I/O becomes ready to dispatch to a storage device, the processing of the foreground I/O can be further prolonged. This is because the size of a device-internal queue (e.g., NCQ [16]) is also limited and a device firmware reorders I/O commands based on the internal geometry of storage media for improving device throughput [44, 20]. Hence, a foreground I/O can be staged because the queue slots are busy handling background I/Os. Furthermore, even if a foreground I/O can be dispatched to the device, the device-internal scheduling can delay the processing of the I/O because of its internal scheduling policy.

2.2 I/O Priority Inversion

The most straightforward way of improving application performance in the existence of background tasks would be to prioritize foreground I/Os over background ones and all I/O layers respect their priorities. However, this simple prioritization is insufficient since I/O priority inversions caused by runtime dependencies can delays the execution of a foreground task (Figure 2). Similar to the priority inversion problem in CPU scheduling [35], I/O priority inversions are problematic because the processing of a background I/O on which a foreground task depend can be arbitrarily delayed by other background I/Os.

Two types of dependencies cause I/O priority inversions: a *task dependency* and an *I/O dependency*. The task dependency occurs when two tasks interact with each other via synchronization primitives, such as a lock and a condition variable. The dependency caused by a lock complicates effective I/O prioritization because a background task can be blocked waiting for an I/O within a critical section that a foreground task needs to enter. For instance, a foreground task attempting to write a file can be blocked on an inode mutex if the mutex is already held by a background task concurrently writing to the different part of that file. Likewise, the depen-

dency caused by a condition variable also cause a similar problem. A foreground task should indirectly wait for the I/Os awaited by a background task that is going to signal the foreground task. For example in Linux ext4, when a foreground task calls `fsync()`, it waits on a specific condition variable which is signaled by jbd2 kernel thread, which could be busy completing journal transactions for background tasks.

Meanwhile, the I/O dependency occurs between a task and an outstanding I/O. Basically, the I/O dependency is generated when a task needs to directly wait for the completion of an ongoing I/O in order to ensure correctness and/or durability. For example, when a foreground task calls `fsync()`, it blocks on the completion of a write I/O that is asynchronously issued by a kernel thread (e.g., pdflush in Linux) for cleaning buffer cache. Once the task and the I/O dependency-induced priority inversions occur, the foreground task should wait for a long time because each layer in the I/O path can arbitrarily prolong the processing of low-priority background I/Os.

Unfortunately, resolving I/O priority inversions is challenging for the following reasons. Firstly, dependency relationships cannot be statically determined since they depend on various runtime conditions, such as execution timing, resource constraint, and client requirement. For example, a foreground task does not always depend on the progress of a kernel thread handling file system transaction since the kernel thread periodically writes out transactions in background [4]. Secondly, dependency occurs in a transitive manner involving multiple concurrent tasks blocked at either synchronization primitives or various I/O stages in multiple layers. We empirically found that a dependency sometimes cascaded in four steps due to the complex interaction between delayed allocation and crash-consistency mechanism in a file system (Section 4.4). Finally, a dependency relationship might not be visible at the kernel-level because of the extensive use of user-level synchronizations (e.g., shared memory mutex) based on kernel-level supports (e.g., Futex [6]) in modern applications.

2.3 Related Work

Table 1 summarizes how the illustrated challenges are addressed (or not) by the existing prioritization schemes. CFQ [2] is a block-level I/O scheduler that supports multiple priority classes (real-time, best-effort, and idle) and priority levels (0 to 7) [7]. However, CFQ prioritizes I/Os only at the block-level queue. It does not consider the I/O priority inversion problem as well as the prioritization at the block queue admission stage.

Redline [42] adapts all I/O layers to limit the interference from background tasks (e.g., virus scanner) for improving responsiveness of interactive applications (e.g., web browser). Redline, however, lacks resolving I/O priority inversions that occur between foreground and background tasks in typical data-intensive applications.

Scheme	Multiple Independent Layers			I/O Priority Inversion	
	Cache	Filesystem	Block	Kernel	User
CFQ [2]	No	No	Yes	No	No
Redline [42]	Yes	Yes	Yes	No	No
Split [43]	Yes	Yes	Yes	No	No
QASIO [32]	No	No	Yes	Yes	No
sCache [33]	No	No	No	Yes	No
RCP	Yes	Yes	Yes	Yes	Yes

Table 1: **I/O prioritization challenges.** *This table shows whether a specific challenge for effective I/O prioritization is addressed or not in each previous work.*

Recently, Split [43], a cross-layer I/O scheduling framework, is introduced to address the limitation of a single-level I/O schedulers. Basically, Split provides additional hooks to several layers for supporting correct cause mapping, cost estimation, and reordering, in the existence of the file system challenges like delayed allocation and journaling [43]. Based on the proposed framework, Split-AFQ and Split-Deadline have been implemented to prove its effectiveness. Split-AFQ, a priority-based scheduler using the Split framework, schedules write I/Os including `write()` and `fsync()` at the system-call layer to avoid the runtime dependencies caused by file system journaling. Different from conventional deadline schedulers, Split-Deadline provide deadline scheduling of `fsync()` calls. In addition, it aggressively writes-back dirty data in background to make the latency of `fsync()` more deterministic by minimizing the file system transaction entanglement. Though Split itself is a generic I/O scheduling framework, its representative schedulers do not specifically consider the I/O priority inversion problem despite its significance.

On the other side, QASIO [32] considers I/O priority inversions for improving system responsiveness. However, QASIO solely focuses on the kernel-level dependencies to asynchronous writes based on the analysis of the several mobile app scenarios. Furthermore, sCache [33] fully considers I/O priority inversions at the kernel-level in order to effectively utilize non-volatile write caches. Both QASIO and sCache, however, do not consider I/O priority inversions at the user-level. Moreover, they do not address the challenges in the I/O path for effective I/O prioritization.

Though several challenges have been separately addressed in the previous work, we argue that only a holistic approach can deliver consistently high application performance as in Figure 1. This is because the I/O priority inversion problem can be worsened when combined with multiple layers as a dependency transitively occurs across layers. Our scheme (RCP) addresses all the challenges in Table 1 by enlightening the I/O path and resolving the kernel- and user-level I/O priority inversions.

3 Our Approach

In this work, we classify I/Os into two priority levels: (performance) *critical* and *non-critical* I/Os. In particular, we define a critical I/O as an I/O in the critical path of request handling since the response time of a request determines the level of application performance.

The proposed classification is distinguished from conventional I/O classification schemes: *I/O-centric* and *task-centric* classifications (Figure 3). The I/O-centric classification differentiates the priority of each I/O based on its operation type (e.g., synchronous I/Os over asynchronous ones [30, 2, 32]). On the other side, the task-centric classification distinguishes the I/Os based on issuing tasks (e.g., foreground I/Os over background ones [42]). These static classification schemes, however, are inadequate for identifying the (performance) criticality of I/Os. In our request-centric viewpoint, synchronous I/Os (e.g., checkpoint writes) and foreground I/Os (e.g., buffered writes) can be non-critical whereas asynchronous I/Os and background I/Os can sometimes be critical due to the runtime dependencies.

Based on the I/O criticality classification, we introduce a *request-centric I/O prioritization* (or RCP) that identifies critical I/Os and prioritizes them over non-critical ones along the I/O path. This form of two-level I/O prioritization is effective for many cases since background tasks are ubiquitous in practice. For example, according to a Facebook developer: "*... There are always cleaners and compaction threads that need to do I/O, but shouldn't hold off the higher-priority "foreground" I/O. ... Facebook really only needs two (or few) priority levels: low and high.*" [5].

Our goals for realizing RCP are twofold: 1) minimizing application modification for detecting critical I/Os, and 2) processing background tasks in a best-effort manner while minimizing the interference to foreground tasks. The following section describes our design for effectively identifying and enforcing I/O criticality throughout the I/O path.

4 I/O Path Enlightenment

4.1 Enlightenment API

The first step to identifying critical I/Os is to track a set of tasks (i.e., foreground tasks) involved in request handling and this can be done in two ways: system-level and application-guided approaches. A system-level approach infers foreground tasks by using information available in the kernel. Though this approach has the benefit of avoiding application modification, it may induce runtime overhead for the inference and the possibility of misidentification. In contrast, an application-guided approach can accurately identify foreground tasks without runtime overheads at the expense of application modification.

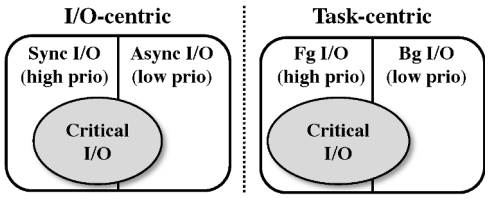


Figure 3: **Comparison with conventional approaches.**

We chose the application-guided approach for accurate detection of foreground tasks without burdening the OS kernel. In particular, we provide an enlightenment interface to user-level so that an application developer (or an administrator if possible) can dynamically set/clear a foreground task based on application-level semantics. The foreground task can be from a short code section to an entire life of a thread depending on where set/clear APIs are called. The simplicity of using APIs makes developers easily prioritize critical I/Os in their applications. We believe the modification cost is also low in practice because typical data-intensive applications already distinguish foreground tasks from background tasks; see Section 6 for the details.

Since the API is solely used for deciding I/O criticality in the OS kernel, a wrong API call does not affect the correct execution of an application. However, API abuse by a malicious or a thoughtless application/tenant may compromise performance isolation among multiple applications/tenants sharing a storage stack. This problem can be solved by integrating RCP to a group-based resource management (e.g., cgroup in Linux [37]). Addressing this issue is out of scope of this paper.

4.2 I/O Priority Inheritance

Based on the enlightenment API, we basically regard a synchronous I/O issued by a foreground task as a critical I/O. This obvious identification, however, is insufficient for detecting all critical I/Os because runtime dependencies cause background I/Os to be awaited by foreground tasks indirectly (Section 2.2). Hence, the next step for critical I/O detection is to handle I/O priority inversions caused by runtime dependencies. To this end, we introduce *I/O priority inheritance* that temporarily gives critical I/O priority to a background task (Section 4.2.1) or a background I/O (Section 4.2.2) on which a foreground task depends to make progress.

4.2.1 Handling Task Dependency

Kernel-level dependency. Resolving the lock-induced dependency has been well-studied in the context of CPU scheduling [35]. Inspired by the previous work, we resolve the lock-induced dependency by inheriting critical I/O priority to a background task when it blocks a foreground task until it leaves a critical section. Specifically, we record an owner task into each lock object (e.g., mutex). When a task is blocked to acquire a lock, the lock

owner inherits I/O priority of the waiting task. The inherited priority is revoked when the owner task exits the critical section. This inheritance procedure is repeated until the foreground task acquires the lock. Note that we consider only blocking-based locks since spinning-based locks are not involved with I/O waiting.

Unlike the case of locks, there is no distinct owner in a condition variable at the time of dependency occurs. To deal with the condition varible-induced dependency, we borrow a solution in previous work [27]. In particular, a task is registered as a *helper task* [27] into the corresponding object of a condition variable when the task is going to signal the condition variable. Later, the helper task inherits I/O priority of a task blocked to wait for the condition become true. The inherited I/O priority is revoked when the helper task finally signals the waiting task. In the kernel-level, this approach is viable because only a few condition variables and threads cause the runtime dependencies.

User-level dependency. The key challenge in handling the user-level dependency is that the OS kernel cannot clearly identify a dependency relationship resulted from user-level synchronizations. For user locks, the kernel cannot determine the owner because a user lock variable is located in a shared memory region and modified through atomic operations (e.g., `cmpxchg` in x86) to indicate the lock status. This is an intrinsic optimization to eliminate unnecessary kernel interventions in the uncontended cases. The kernel is involved only when to block or to wake up lock waiters via a system call (e.g., `sys_futex()`). As a consequence, the OS kernel can see only the waiters failed to acquire a user-level lock.

To detect the owner of a user lock, we adjust a user lock primitive (e.g., `pthread_mutex_lock()`) to additionally pass down the information of the owner when a task should block in the kernel due to lock contention. In practice, this can be readily done without modifying existing interfaces; see Section 5 for our implementation. Based on the delivered information, the kernel can properly inherit the I/O priority of a waiting task to a lock-holder task. Note that this modification does not entail the kernel intervention in uncontended cases.

Unlike the condition variable-induced dependency in kernel-level, handling such dependency in user-level is difficult because it is hard to pinpoint helper tasks for condition variables. Modern applications extensively use user-level condition variables for various purposes. For instance, we found over a hundred of user-level condition variables in the source code of MongoDB. Therefore, properly identifying all helper tasks is not trivial even for an application developer.

We adopt an inference technique that identifies a helper task based on usage history of each user-level condition variable. Typically, a background task is dedicated to a specific activity like logging, checkpointing, compaction, and buffer cleaning. Hence, a task signaling a condition is highly likely signal the condition again. Based on this observation, a background task is registered as a helper task when it signals a user-level condition variable. Then, the helper task inherits critical I/O priority of a foreground task when the foreground task needs to block on the user-level condition variable. The helper task is unregistered when it does not signal again for a specific time window.

4.2.2 Handling I/O Dependency

Properly resolving a dependency to an outstanding I/O is complicated because the dependent I/O can be in any stage in the block layer at the time of the dependency occurs. For example, an outstanding I/O can be in admission control stage waiting for the available slots of a block-level queue. Hence, we need to track the status of an ongoing non-critical I/O and appropriately reprioritize it according to the current location when required.

For tracking outstanding non-critical I/Os, we add an `ncio` data structure that stores an I/O descriptor, current location, and the descriptor of a requesting task. An `ncio` object is allocated when an incoming I/O is classified as non-critical at the entrance of the block layer, and inserted to a per-device list indexed by starting sector number. The fields including I/O descriptor and current location in the `ncio` object are properly updated as the corresponding I/O flows along the I/O path. The allocated `ncio` object is freed when the corresponding I/O is reclassified or dispatched to a device.

When a dependency to an ongoing non-critical I/O occurs, the per-device `ncio` list is searched to find the corresponding `ncio` object. Then, the non-critical I/O is reclassified as critical I/O based on the information in the I/O descriptor stored in an `ncio` object if the corresponding `ncio` object is found. In this case, we may need to conduct additional chores according to the current location of the dependent I/O; we present the details in the following subsection.

The runtime overhead for maintaining the `ncio` structure is fairly small. In our implementation, we used a red-black tree for fast lookup. The memory cost is also limited because the number of outstanding non-critical I/Os is limited (128 by default) by the admission control mechanism at the block layer.

4.3 Criticality-Aware I/O Prioritization

As we discussed in Section 2.1, prioritizing critical I/Os only within a single layer (e.g., scheduling in a block-level queue) is ineffective for improving application performance. Hence, we adapt each layer in the I/O path to make it understand and enforce the I/O criticality.

In the caching layer, similar to the approach in [42], we apply separate dirty ratios to tasks issuing critical and non-critical writes, respectively. For tasks issuing non-critical writes, the applied dirty ratio is low (1% by default) to mitigate the interference to foreground tasks. With the lowered limit, a background task writing a large amount of data to buffer cache cannot fill all the available space since it should block until the current dirty ratio drops below the configured ratio. As a consequence, a foreground task is not blocked by a burst of background writes. Moreover, a foreground task calling `fsync()` does not need to depend on a large amount of dirty data generated by background tasks resulting from the file system ordering requirement [42, 43].

In the admission control stage at the block layer, we separately allocate the block queue slots for critical and non-critical I/Os, respectively, so that the admission control is isolated between critical and non-critical I/Os. To resolve the dependency to the I/O blocked at this stage, we transiently give critical I/O priority to the requesting task recorded in the corresponding `ncio` object and wake up the task to make it retry allocation of available slot with critical I/O priority. By doing so, the criticality-inherited I/O can avoid the unnecessary congestion with other non-critical I/Os during the admission.

We also designed a simple priority-based I/O scheduler at the block layer. In particular, our scheduler maintains two FIFO queues that are dedicated for critical and non-critical I/Os each. Then, all I/Os in the critical queue is dispatched first before any I/O in the non-critical queue. To prevent starvation, we use a timer to monitor the non-critical queue and guarantee that at least one non-critical I/O is processed per unit of time (10 ms by default). Furthermore, we added queue promotion support into our scheduler for properly handling a dependency to a non-critical I/O staged at the block-level queue. In order to minimize the interference at the device-level, we conservatively limit the number of non-critical I/Os dispatched to a storage device; this number is configurable and we use one by default. This form of limiting is common in practice for improving responsiveness [2, 36]. Our scheme can be integrated with an existing feature-rich scheduler like Linux CFQ at the expense of additional work to support the I/O priority inheritance.

4.4 Handling Transitive Dependency

Transitive dependencies make effective I/O prioritization more challenging. Consider a dependency chain of tasks $(\tau_1, \tau_2, ..., \tau_n)$ where each task $\tau_i (1 \leq i \leq n-1)$ is blocked due to a task dependency to τ_{i+1}. The last task τ_n can be in one of the three states: runnable, blocked due to I/O dependency, and blocked at the admission control stages. If τ_n is runnable or blocked due to the I/O dependency, the transitive dependency can be resolved by inheriting the critical I/O priority through the dependency chain.

However, if τ_n is blocked at one of the admission stages, inheriting the critical I/O priority is insufficient because the cause of the blocking should also be resolved. To handle this case, the applied dirty ratio at the caching layer is transiently changed to that of critical I/Os and the blocked task is woken up. At the block layer, similar to the I/O dependency resolution, the critical I/O priority is transiently inherited by the blocked task and the task is woken up. Then, the awakened task retries the admission with the changed I/O priority.

In order to resolve the transitive dependencies, we record a blocking status into the descriptor of a task when the task is about to be blocked. A blocking status consists of blocking cause and an object to resolve the cause. Blocking cause can be one of task dependency, I/O dependency, and blocking at admission control stage. For the task dependency cause, a corresponding object of lock or condition vatiable is recorded. For the I/O dependency cause, an I/O descriptor (i.e., block device identifier and sector number) is recorded. No additional object is recorded for the blocking at admission control stage. Based on the recorded blocking status, a foreground task can effectively track and resolve the blocking causes in the transitive dependencies.

In our experiments, at most four steps of transitive dependency has occurred. In particular, a foreground task is blocked on an inode mutex for file writes. The mutex is held by a background task and the task is also blocked waiting for the signal by a journaling daemon since the task tries to open a new file system transaction. The journaling daemon is also waiting for the completion of updating journal handle by another background task. The last background task is blocked on the admission control stage of the block layer because the task is issuing a burst of writeback for carrying out delayed allocation.

5 Implementation on Linux

We implemented our schemes in Linux 3.13 including around 3100 lines of additional code. A task descriptor has a field indicating whether this task is a foreground task or not. The field is set/cleared by using an existing `setpriority()` system call interface; unused values are used. To denote I/O crticality, a `bi_rw` field in `bio` and `cmd_flags` field in `request` data structures are given an extra flag. These flags are used for the admission control and I/O scheduling at the block layer.

We implemented the I/O priority inheritance to `mutex`, `rw_semaphore`, and `semaphore`. In addition, we resolved the condition variable-induced task dependency in Linux journaling layer (i.e., jbd2). We registered jbd2 kernel thread as a helper task for the condition variables `j_wait_trasnaction_locked` and `j_wait_done_commit`. For the condition vari-

able j_wait_updates, a helper task is dynamically assigned and removed since a helper task can be any task having a user context. The three condition variables are specific to ext4, which can hinder the adoption of our scheme to other file systems. A good starting point for identifying condition variables causing runtime dependencies is to inspect the wait queues (i.e., wait_queue_head_t) defined in a file system. For example, only nine and three wait queues need to be inspected for integrating with xfs and f2fs, respectively.

The I/O priority inheritance for resolving the user-level dependency is implemented in Futex [6] and System V Semaphore [15] (SysV sem for short). The priority inheritance in Futex (FUTEX_LOCK_PI) is exploited and a similar method is implemented in SysV sem with an additional owner field to sembuf. A lock owner is recorded at user-level when acquiring a lock and is passed down to the kernel when a waiter is blocked. For the I/O dependencies, we implemented the I/O priority inheritance to wait_on_bit() and wait_on_bit_lock() methods that are used to synchronize with buffer pages. We attempt I/O prioritization at these methods when a task waits until a specific bit (PG_locked, PG_writeback, BH_Lock, and BH_Shadow) is cleared. Note that BH_Shadow is used to protect a buffer page that are under journaling I/O for guaranteeing file system consistency.

6 Application Studies

To validate the effectiveness of our scheme, we chose three widely deployed data-intensive applications: PostgreSQL [10] relational database v9.5, MongoDB [8] document store v3.2, and Redis [22] key-value store v3.0. For tagging foreground tasks, we inserted 15, 14, and 2 lines of code to PostgreSQL, MongoDB, and Redis, respectively. This result indicates that adopting the enlightenment API is not complicating for typical data-intensive applications.

PostgreSQL relational database. In PostgreSQL, *backend* is dedicated to client for serving requests while other processes, such as *checkpointer*, *background writer*, *log writer*, and *autovacuum worker*, carry out I/O jobs in background. The checkpointer periodically flushes all dirty data buffers to disk and writes a special checkpoint record to the log file, in order to truncate transaction logs reflected to database, thereby bounding storage space and crash recovery time. The background writer periodically writes some dirty buffers to disk to keep regular backend processes from having to write out dirty buffers. Similarly, the log writer periodically writes out the log buffer to disk in order to reduce the amount of synchronous writes backend processes should perform at commit time. The autovacuum worker reclaims storage occupied by dead tuples since tuples deleted or obsoleted by an update are not physically removed from their table.

We classified the backend processes as foreground tasks. In addition, since PostgreSQL utilizes SysV sem to implement *LWLocks*, which is a user-level mutex, we modified *LWLocks* to pass down lock owner information to the kernel. Note that the information is passed only when a waiter is blocked.

MongoDB document store. In WiredTiger, which is the default storage engine since MongoDB 3.2, background threads, such as *log threads*, *eviction threads*, and *a checkpoint thread*, conduct internal activities, such as logging and checkpointing, while *client threads* handle external requests in foreground. The log threads periodically flush log buffers to an on-disk journal file. Likewise, the eviction threads write dirty pages in the internal cache to OS buffer cache for making free pages. The checkpoint thread periodically flushes all dirty pages in the internal cache to disk for consistency.

We classified the client threads as foreground tasks. MongoDB extensively uses Pthread mutex and condition variable. To handle user-level dependency, we modified the protocol of Pthread mutex to PTHREAD_PRIO_INHERIT to distinguish Pthread mutex and condition variable at the kernel-level and to utilize the priority inheritance implemented in Futex.

Redis key-value store. Redis has two options to provide durability: snapshotting and command logging. The snapshotting periodically produces point-in-time snapshots of the dataset but does not provide complete durability since up to a few minutes of data can be lost. Meanwhile, the command logging guarantees the complete durability by synchronously writing an update log to an append-only file before responding back to the command. In the command logging, log rewriting is periodically conducted to constrain the size of the log file.

Similar to the other applications, the snapshotting and log rewriting are conducted by child processes in background while a main server process serves all requests sequentially. Hence, we classified the main server process as a foreground task.

7 Evaluation

7.1 Experimental Setup

The evaluation system is a Dell PowerEdge R530 server that is equipped with two Intel Xeon E5-2620 processors and 64 GB RAM. The CPU frequency is set to the highest level and hyper-threading is enabled. A single 1 TB Micron MX200 SSD is used to store data sets for the evaluation workloads. We used Ubuntu 14.04 with the modified Linux kernel version 3.13 as an OS and ext4 file system mounted with the default options.

We used CFQ as the baseline for our experiments. In addition, we used CFQ-IDLE to prioritize foreground

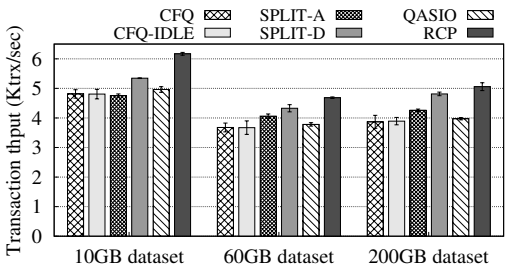


Figure 4: **PotsgreSQL request throughput.**

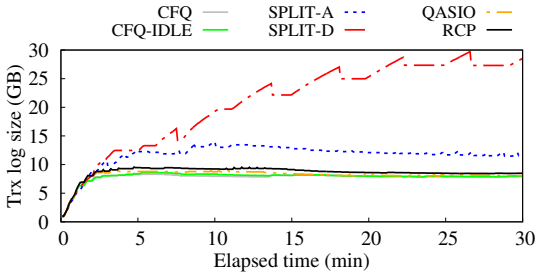


Figure 5: **PostgreSQL transaction log size.**

tasks by putting background tasks (e.g., checkpointer) to idle-priority [7]. We also used the split-level schedulers [43], including Split-AFQ (SPLIT-A) and Split-Deadline (SPLIT-D), and QASIO [32] for comparison.

For RCP, we configured 1% dirty ratio for non-critical writes and 20% dirty ratio (the default ratio in Linux) for critical writes. We separately allocated 128 slots for each critical and non-critical block-level queues. The number of non-critical I/Os outstanding to a storage device is limited to one, and the timeout for non-critical I/Os is set to 10 ms to prevent starvation at the block-level queue.

7.2 PostgreSQL Relational Database

We used the OLTP-Bench [28] to generate a TPC-C [17] workload for PostgreSQL. We simulated 50 clients running on a separate machine for 30 minutes. PostgreSQL was configured to have 40% buffer pool of the size of the initial database and to checkpoint every 30 seconds. For CFQ-IDLE and QASIO, we put the checkpointer to the idle-priority. For SPLIT-A, we set the highest and the lowest I/O priorities to backends and the checkpointer, respectively. For SPLIT-D, we set 5 ms and 200 ms `fsync()` deadlines to backends and the checkpointer, respectively; the configurations are dictated from those in [43]. We report transactions per second and transaction latency as the performance metrics.

Request throughput. Figure 4 shows transaction throughput averaged over three runs on an SSD with 100, 600, and 2000 TPC-C scale factors, which correspond to about 10 GB, 60 GB, and 200 GB of initial databases, respectively. We used unlimited request rate (i.e., zero idle/think time) for this experiment. CFQ-IDLE does not help to improve application throughput though we put the major background task (i.e., checkpointer) to idle-class priority because CFQ-IDLE prioritizes high-priority (foreground) I/Os only at the block-level scheduler. SPLIT-A improves transaction throughput only when read I/Os are dominant as scale factor increases. This is because SPLIT-A does not consider the I/O priority inversion problem and hinders foreground tasks from fully utilizing the OS buffer cache by scheduling writes at the system-call layer. As presented in [43], SPLIT-D is effective in improving the PostgreSQL's performance mainly because it procrastinates the checkpointing task. Though QASIO addresses I/O priority inversions induced by kernel-level dependencies, QASIO does not show any noticeable improvement due to the unresolved dependencies and the inability existed in the block-level scheduler as in CFQ-IDLE. RCP outperforms the existing schemes by about 15%–37%, 8%–28%, and 6%–31% for 100, 600, and 2000 scale factors, respectively.

Impact on background task. To analyze the impact of each scheme on the background task (i.e., checkpointer), we measured the size of transaction logs during the workload execution in the case of 100 scale factor. As shown in Figure 5, CFQ, CFQ-IDLE, and QASIO complete checkpointing task regularly as intended, thereby bounding the size of transaction logs to 8GB in total. SPLIT-A increases the size of transaction logs to 12GB. On the other hand, SPLIT-D increases the size of transaction logs by 3.6× over CFQ since it penalizes the regular checkpoint by delaying every `fsync()` calls made by the checkpointer until the dirty data of the requested file drops to 100 pages. As a result, SPLIT-D leads to using more storage space and possibly taking longer recovery time, which are undesirable in practice [12]. RCP completes the checkpointing task as frequent as CFQ while improving request throughput. Note that the log sizes with the other scale factors show similar trend.

Request latency. In order to show the effectiveness of RCP in terms of request latency, we ran rate-controlled TPC-C workload (i.e., fixed number of transactions) with 100 scale factor. Figure 6 demonstrates a complementary cumulative distribution function (CCDF), and so the point (x,y) indicates that y is the fraction of requests that experience a latency of at least x ms. This form of representation helps visualizing latency tails, as y-axis labels correspond to the 0^{th}, 90^{th}, 99^{th} (and so on) percentile latencies. Though CFQ-IDLE, SPLIT-D, and QASIO achieves better latency than CFQ, all the existing schemes induce several seconds request latency after 99^{th} percentile. This is because the critical path of request execution is arbitrarily prolonged at the various stages in the I/O path. On the other hand, RCP bounds request latency up to around 300 ms. We omit the latency results with 600 and 2000 scale factors because it

Latency (ms)	CFQ	CFQ-IDLE	SPLIT-A	SPLIT-D	QASIO	RCP
semtimedop	1351.8 (3.9)	3723.0 (4.2)	3504.4 (3.4)	1951.1 (3.4)	2241.7 (2.7)	247.8 (1.0)
j_wait_done_commit	1282.9 (55.9)	1450.1 (66.0)	342.4 (6.4)	1886.1 (10.6)	198.0 (3.8)	23.9 (2.5)
PG_writeback	490.2 (0.2)	1677.6 (0.1)	454 (0.2)	458.0 (0.2)	454.7 (0.1)	243.2 (0.1)
get_request	481.8 (3.0)	3722.8 (22.0)	405.2 (1.3)	240.1 (2.8)	2241.1 (3.7)	1.3 (0.1)
j_wait_transaction_locked	306.5 (68.8)	229.4 (53.2)	0.4 (0.1)	2.4 (0.2)	0.3 (0.1)	0.2 (0.1)
BH_Lock	201.3 (40.3)	1339.7 (356.7)	1.1 (0.1)	53.9 (11.1)	0.0 (0.0)	1.7 (0.5)
rwsem_down	92.4 (8.9)	357.1 (179.7)	0.8 (0.1)	33.4 (1.4)	0.0 (0.0)	2.3 (0.2)
BH_Shadow	46.5 (2.9)	15.9 (2.9)	208.9 (3.8)	236.1 (14.1)	1294.0 (36.9)	2.4 (0.2)
mutex_lock	32.7 (7.0)	16.3 (3.1)	18.6 (2.5)	944.3 (53.3)	53.3 (3.1)	0.4 (0.1)
write_entry	N/A	N/A	1703.2 (1.8)	0.0 (0.0)	N/A	N/A
fsync_entry	N/A	N/A	1084.4 (0.5)	0.0 (0.0)	N/A	N/A

Table 2: **PostgreSQL system latency breakdown.** *This table shows the maximum latency incurred at each kernel method in milliseconds; the corresponding average latency is presented in parenthesis.*

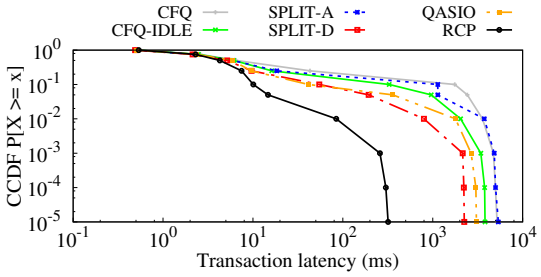


Figure 6: **PostgreSQL request latency.**

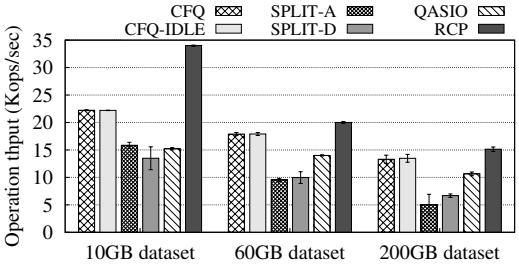


Figure 7: **MongoDB request throughput.**

is similar to that with 100 scale factor.

To analyze the reason behind the performance differences, we measured the maximum and average wait time of foreground tasks (i.e., PostgreSQL backends) in kernel functions using LatencyTOP [29]. As shown in Table 2, CFQ incurs tens to a thousand milliseconds latency at various synchronization methods. In particular, foreground tasks suffer from excessive latency at SysV sem (`semtimedop`) and `j_wait_done_commit` condition variable. CFQ-IDLE additionally causes several seconds latency waiting for the allocation of a block-level queue slot (`get_request`), the writeback of cache pages (`PG_writeback`), and the acquisition of a buffer lock (`BH_Lock`). SPLIT-A and -D are effective in resolving the file system-induced dependencies by scheduling writes at the system-call layer. However, SPLIT-A causes over one second latency at `write_entry` and `fsync_entry` because it prevents foreground tasks from fully utilizing the OS buffer cache. SPLIT-D also causes about one second latency at inode mutex (`mutex_lock`) due to the I/O priority inversion. Though QASIO resolves some dependency-induced latencies ocurred in CFQ-IDLE, it still incurs excessive latencies at `semtimedop`, `get_request`, and `BH_Shadow`. On the contrary, RCP bounds all the latencies below 250 ms.

7.3 MongoDB Document Store

For MongoDB, we used the update-heavy workload (Workload A) in the YCSB [25] benchmark suite. We simulated 150 clients running on a separate machine for 30 minutes. MongoDB was configured to have internal cache of size 40% of the initial data set. We applied the identical configuration to client threads and checkpoint thread as the PostgreSQL case for CFQ-IDLE, SPLIT-A, SPLIT-D, and QASIO. We report operations per second and operation latency as the performance metrics.

Request throughput. Figure 7 plots request throughput averaged over three runs with 10, 60, and 200 million objects, which correspond to about 10 GB, 60 GB, and 200 GB of initial data sets, respectively. As in the PostgreSQL case, CFQ-IDLE and QASIO do not help to mitigate the interference from background tasks. Unlike the case of PostgreSQL, SPLIT-D degrades request throughput rather than improving due to the different application design. MongoDB stores a collection of documents into a single file whereas PostgreSQL splits a database into multiple 1 GB-sized files. Hence, the checkpoint thread in MongoDB writes whole data set to the collection file and then calls `fsync()` to the file. In this case, SPLIT-D cannot help to prevent write entanglement since it does not schedule writes at the system-call layer. Meanwhile, scheduling writes at the system-call layer (SPLIT-A) is not also effective because buffered writes are handled slowly as in the PostgreSQL case. On the other hand, RCP improves request throughput by about 53%–152%, 12%–136%, and 12%–201% for 10, 60, and 200 million objects, respectively, compared to the existing schemes.

Request latency. Figure 8 shows CCDF of request latency measured during the execution of the rate-controlled YCSB workload (i.e., fixed number of operations) with 10 million objects. CFQ and CFQ-IDLE

Latency (ms)	CFQ	CFQ-IDLE	SPLIT-A	SPLIT-D	QASIO	RCP
futex_lock_pi	6092.6 (3.3)	11849.3 (3.4)	8300.6 (3.6)	12292.6 (3.7)	3717.4 (3.1)	305.6 (3.2)
j_wait_done_commit	6067.3 (2.3)	11846.2 (2.4)	4066.9 (2.4)	10598.4 (2.6)	3652.1 (2.0)	246.5 (2.0)
pg_writeback	239.5 (0.1)	240.1 (0.1)	426.2 (0.2)	241.0 (0.2)	241.8 (0.2)	64.2 (0.1)
get_request	35.0 (18.4)	48636 (26.6)	17.3 (5.5)	0.0 (0.0)	852.5 (6.8)	0.0 (0.0)
j_wait_transaction_locked	2.2 (1.0)	1790.3 (44.1)	1.4 (0.1)	0.0 (0.0)	1942.6 (24.0)	2.0 (0.9)
mutex_lock	0.0 (0.0)	3296.6 (544.2)	0 (0.0)	1.2 (0.1)	992.0 (77.2)	0.0 (0.0)
write_entry	N/A	N/A	7884.9 (27.4)	0.0 (0.0)	N/A	N/A
fsync_entry	N/A	N/A	8273.1 (26.2)	0.0 (0.0)	N/A	N/A

Table 3: **MongoDB system latency breakdown.** *This table shows the maximum latency incurred at each kernel method in milliseconds; the corresponding average latency is presented in parenthesis.*

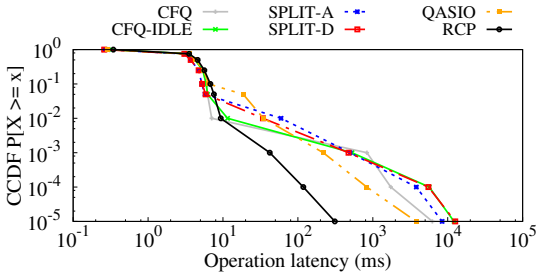


Figure 8: **MongoDB request latency.**

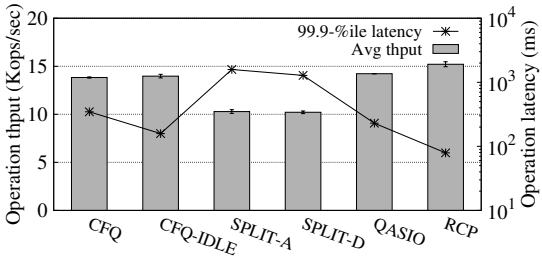


Figure 9: **Redis performance.**

significantly increase latency tail especially after 99^{th} percentile. SPLIT-A and SPLIT-D also cannot help to bound latency; about five seconds latency at 99.99^{th} percentile. Though QASIO improves request latency compared to the other existing schemes, it still incurs about four seconds latency at 99.999^{th} percentile. On the contrary, RCP completes all the requests within 310 ms by carefully handling the critical path of request execution. Note that the latency results with 60 and 200 million objects is similar to that of 10 million objects.

Table 3 shows the maximum and average system latencies in foreground tasks. All the existing schemes induce excessive latencies at various synchronization methods, such as Pthread mutex (`futex_lock_pi`) and `j_wait_done_commit` condition variable. Unlike the case of PostgreSQL, CFQ-IDLE and QASIO cause over a second latency at `j_wait_transaction_locked` condition variable, which is a barrier for starting a new file system transaction. RCP largely reduces dependency-induced system latencies .

7.4 Redis Key-Value Store

For Redis, we used the same workload as the MongoDB's except that we concurrently ran ten YCSB benchmarks against ten Redis instances to utilize our multicore testbed due to the single threaded design of Redis [23]. We enabled both snapshotting and command logging for data safety [24]. We report operations per second and operation latency as the performance metrics.

Figure 9 plots operation throughput averaged over three runs and 99.9^{th} percentile operation latency. CFQ-IDLE and QASIO slightly improves application performance by putting the background tasks including the snapshotting and log rewriting to idle-class priority. SPLIT-A and SPLIT-D deteriorate application performance because SPLIT-A does not fully utilize write buffer in the caching layer and SPLIT-D does not protect non-critical writes from file system-level entanglement. By handling the limitations in the existing prioritization schemes in terms of application performance, RCP improves request throughput by about 7%–49% compared to the existing schemes. In addition, RCP shows 78 ms request latency at 99.9^{th} percentile, thereby achieving 2×–20× improvement over the existing schemes.

7.5 Need for Holistic Approach

In order to show why a holistic approach is crucial for application performance, we selectively disabled one of the components in our scheme, the caching layer, the block layer, the kernel-level dependency handling, the user-level dependency handling, and the transitive dependency handling. Figure 10 shows average request throughput normalized to that of all the components are enabled in the 10 GB data set configurations. Disabling any one of the component degrades application throughput by about 7–33% and 6–45% for PostgreSQL and MongoDB each. This result justifies our claim that only a holistic approach can guarantee high degree of application performance.

7.6 Impact of Limiting I/Os

Due to hidden and unpredictable I/O scheduling inside storage, we limited the number of sojourn non-critical I/Os to one. This may lead to low utilization of storage devices when there is low foreground activity. To quantify the impact on system throughput, we concurrently

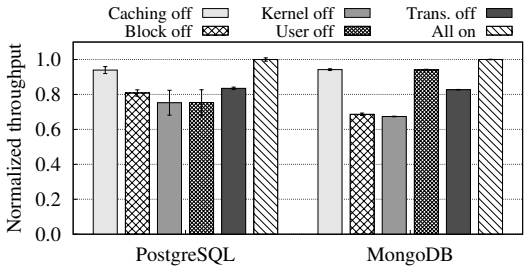


Figure 10: **Normalized request throughput with turning off one of the components in RCP.**

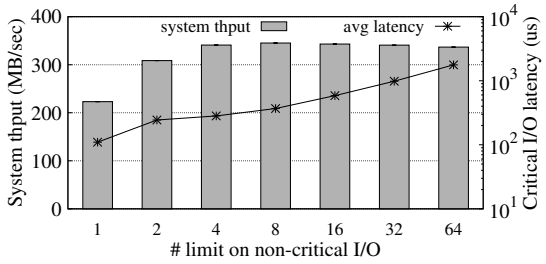


Figure 11: **Tradeoff between system throughput and foreground latency.**

ran a sequential write workload with 5 ms thinktime as a lightly-loaded foreground task and a random write workload without thinktime as a I/O-intensive background task. As shown in Figure 11, the system-wide throughput increases from 223 MB/sec to 345 MB/sec (55% improvement) while relaxing the limitation on non-critical I/Os. However, the improvement in system throughput comes with the degraded latency of foreground I/O. In particular, the average latency of foreground I/O gradually increases from 110 us to 1771 us (over 16× slowdown). If storage devices implement priority-based scheduling feature in storage interface standards (e.g., SCSI, ATA, and NVMe), this tradeoff would be minimized by exploiting the priority feature.

8 Discussion

Penalizing background tasks. Whether a foreground task really does not rely on the progress of background tasks depends on the semantic of an application. For example in MySQL, when the size of transaction logs is below a preconfigured maximum size, a foreground task does not wait for a checkpointer. However, the foreground task stops accepting updates when the log size exceeds the threshold. One workaround is to provide another threshold which is a point to give the critical I/O priority to the checkpointer. This sort of application modification requires understanding of application semantic. We believe application developers are likely willing to conduct such modifications since our scheme brings superior performance with a simple API.

User-level threading library. An application may use a user-level threading library, such as Fiber [1], though it is uncommon for data-intensive applications we targeted. In this case, our scheme cannot detect the user-level dependency. If using a user-level threading library is prevalent, implementing the I/O priority inheritance to the library based on the enlightenment API may be necessary.

User-level condition variable. Our scheme uses a simple history-based inference technique to track a helper task of a user-level condition variable. In the tested applications, this method was sufficient since observed helpers were mostly static. However, if an application has a complex relationship between condition variables and helpers, a more sophisticated inference technique is desired, which is our future work.

Implementation practicality. Our prototype implementation involves modifications to all the layers and the synchronization methods in the I/O path, thereby hindering our scheme from wide adoption. The most promising direction to be practically viable is exploiting the Split framework [43]. It provides a collection of handlers for an I/O scheduler to operate across all layers in the I/O path. We believe our scheme can be cleanly implemented based on the framework by controlling non-critical writes at the system-call level, before the caching and file system layer generates complex dependencies, and non-critical reads at the block-level.

9 Concluding Remarks

In this paper, we have shown that all the layers in the storage I/O path should be considered as a whole with I/O priority inversion in mind for effective I/O prioritization. Our experiments demonstrate that the proposed scheme can provide low and predictable request latency while minimally penalizing background tasks. We believe that our scheme can contribute to reducing total cost of ownership by alleviating the contention introduced by a burst of background I/Os and thereby relaxing the need for over-provisioning storage resources.

To handle the fairness issue which results from sharing a storage stack among multiple applications/tenants, we plan to explore integrating our scheme with an existing group-based I/O scheduler (e.g, CFQ). We also plan to investigate request handling in a distributed system with replicated data stores.

10 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Raju Rangaswami, for their valuable comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2014R1A2A1A10049626) and by Samsung Electronics.

References

[1] Boost Fiber. http://olk.github.io/libs/fiber/doc/html.

[2] Completely Fair Queuing. https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[3] Database checkpoints (SQL server). http://msdn.microsoft.com/en-us/library/ms189573.aspx.

[4] Ext4 filesystem. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[5] Facebook and the kernel. https://lwn.net/Articles/591780.

[6] futex(2) - Linux man page. http://linux.die.net/man/2/futex.

[7] ionice(1) - Linux man page. http://linux.die.net/man/1/ionice.

[8] MongoDB. https://www.mongodb.org.

[9] MongoDB backup methods. https://docs.mongodb.org/manual/core/backups.

[10] PostgreSQL. http://www.postgresql.org.

[11] PostgreSQL continuous archiving and point-in-time recovery. http://www.postgresql.org/docs/current/static/continuous-archiving.html.

[12] PostgreSQL pain points. https://lwn.net/Articles/591723.

[13] PostgreSQL routine vacuuming. http://www.postgresql.org/docs/9.5/static/routine-vacuuming.html.

[14] PostgreSQL WAL configuration. http://www.postgresql.org/docs/9.5/static/wal-configuration.html.

[15] semop(2) - Linux man page. http://linux.die.net/man/2/semop.

[16] Serial ATA native comminad queueing: An exciting new performance feature for serial ATA. Intel Corporation and Seagate Technology.

[17] The TPC-C benchmark. http://www.tpc.org/tpcc.

[18] AMVROSIADIS, G., AND BHADKAMKAR, M. Identifying trends in enterprise data protection systems. In *Proceedings of the 2015 USENIX Annual Technical Conference - USENIX ATC '15*.

[19] AVANTIKA MATHUR, MINGMING CAO, SUPARNA BHATTACHARYA, ANDREAS DILGER, ALEX TOMAS, LAURENT VIVIER. The new ext4 filesystem: Current status and future plans. In *In Proceedings of the Ottawa Linux Symposium - OLS '07*.

[20] BLAGOJEVIĆ, F., GUYOT, C., WANG, Q., TSAI, T., MATEESCU, R., AND BANDIĆ, Z. Priority IO scheduling in the Cloud. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing - HotCloud '13*.

[21] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation - OSDI '06*.

[22] CITRUSBYTE. Redis. http://redis.io/.

[23] CITRUSBYTE. Redis latency problems troubleshooting. http://redis.io/topics/latency.

[24] CITRUSBYTE. Redis persistence. http://redis.io/topics/persistence.

[25] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing - SoCC '10*.

[26] CORBET, J. No-I/O dirty throttling. https://lwn.net/Articles/456904.

[27] CUCINOTTA, T. Priority inheritance on condition variables. In *Proceedings of the 5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications - OSPERT '09*.

[28] DIFALLAH, D. E., PAVLO, A., CURINO, C., AND CUDRE-MAUROUX, P. OLTP-Bench: An extensible testbed for benchmarking relational databases. In *Proceedings of the VLDB Endowment - VLDB '13*.

[29] EDGE, JAKE. Finding system latency with LatencyTOP. http://lwn.net/Articles/266153/.

[30] GANGER, G. R., AND PATT, Y. N. The process-flow model: Examining I/O performance from the system's point of view. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems - SIGMETRICS '93*.

[31] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1992.

[32] JEONG, D., LEE, Y., AND KIM, J.-S. Boosting quasi-asynchronous I/O for better responsiveness in mobile devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies - FAST '15*.

[33] KIM, S., KIM, H., KIM, S.-H., LEE, J., AND JEONG, J. Request-oriented durable write caching for application performance. In *Proceedings of the 2015 USENIX Annual Technical Conference - USENIX ATC '15*.

[34] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35.

[35] LAMPSON, B. W., AND REDELL, D. D. Experience with processes and monitors in Mesa. *Communications of the ACM 23*, 2 (Feb. 1980), 105–117.

[36] LOSH, M. W. I/O Scheduling in CAM. In *Proceedings of the Technical BSD Conference - BSDCan '15*.

[37] MENAGE, P. B. Adding generic process containers to the Linux kernel. In *Proceedings of the Ottawa Linux Symposium - OLS '07*.

[38] MIKHAIL BAUTIN, GUOQIANG JERRY CHEN, PRITAM DAMANIA, PRAKASH KHEMANI, KARTHIK RANGANATHAN, NICOLAS SPIEGELBERG, LIYIN TANG, M. V. Storage infrastructure behind Facebook messages using HBase at scale. *IEEE Data Engineering Bulletin 35*, 2 (2012), 4–13.

[39] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles - SOSP '91*.

[40] RUSSINOVICH, M. E., SOLOMON, D. A., AND IONESCU, A. *Windows Internals, Part 2: Covering Windows Server 2008 R2 and Windows 7*, 6th ed. Microsoft Press, 2012.

[41] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference - USENIX ATC '96*.

[42] TING YANG, TONGPING LIU, EMERY D. BERGER, SCOTT F. KAPLAN, J. ELIOT, B. M. Redline: First class support for interactivity in commodity operating systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation - OSDI '08*.

[43] YANG, S., HARTER, T., AGRAWAL, N., KOWSALYA, S. S., KRISHNAMURTHY, A., AL-KISWANY, S., KAUSHIK, R. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Split-level I/O scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*.

[44] YU, Y. J., SHIN, D. I., EOM, H., AND YEOM, H. Y. NCQ vs. I/O scheduler. *ACM Transactions on Storage 6*, 1 (Mar. 2010), 1–37.

LightNVM: The Linux Open-Channel SSD Subsystem

Matias Bjørling[†*] Javier González[†] Philippe Bonnet[*]

[†]*CNEX Labs, Inc.* [*]*IT University of Copenhagen*

Abstract

As Solid-State Drives (SSDs) become commonplace in data-centers and storage arrays, there is a growing demand for predictable latency. Traditional SSDs, serving block I/Os, fail to meet this demand. They offer a high-level of abstraction at the cost of unpredictable performance and suboptimal resource utilization. We propose that SSD management trade-offs should be handled through *Open-Channel SSDs*, a new class of SSDs, that give hosts control over their internals. We present our experience building *LightNVM*, the Linux Open-Channel SSD subsystem. We introduce a new Physical Page Address I/O interface that exposes SSD parallelism and storage media characteristics. LightNVM integrates into traditional storage stacks, while also enabling storage engines to take advantage of the new I/O interface. Our experimental results demonstrate that LightNVM has modest host overhead, that it can be tuned to limit read latency variability and that it can be customized to achieve predictable I/O latencies.

1 Introduction

Solid-State Drives (SSDs) are projected to become the dominant form of secondary storage in the coming years [18, 19, 31]. Despite their success due to superior performance, SSDs suffer well-documented shortcomings: log-on-log [37,57], large tail-latencies [15,23], unpredictable I/O latency [12, 28, 30], and resource underutilization [1, 11]. These shortcomings are not due to hardware limitations: the non-volatile memory chips at the core of SSDs provide predictable high-performance at the cost of constrained operations and limited endurance/reliability. It is how tens of non-volatile memory chips are managed within an SSD, providing the same block I/O interface as a magnetic disk, which causes these shortcomings [5,52].

A new class of SSDs, branded as *Open-Channel SSDs*, is emerging on the market. They are an excellent platform for addressing SSD shortcomings and managing trade-offs related to throughput, latency, power consumption, and capacity. Indeed, open-channel SSDs expose their internals and enable a host to control data placement and physical I/O scheduling. With open-channel SSDs, the responsibility of SSD management is shared between host and SSD. Open-channel SSDs have been used by Tier 1 cloud providers for some time. For example, Baidu used open-channel SSDs to streamline the storage stack for a key-value store [55]. Also, Fusion-IO [27] and Violin Memory [54] each implement a host-side storage stack to manage NAND media and provide a block I/O interface. However, in all these cases the integration of open-channel SSDs into the storage infrastructure has been limited to a single point in the design space, with a fixed collection of trade-offs.

Managing SSD design trade-offs could allow users to reconfigure their storage software stack so that it is tuned for applications that expect a block I/O interface (e.g., relational database systems, file systems) or customized for applications that directly leverage open-channel SSDs [55]. There are two concerns here: (1) a block device abstraction implemented on top of open-channel SSDs should provide high performance, and (2) design choices and trade-off opportunities should be clearly identified. These are the issues that we address in this paper. Note that demonstrating the advantages of application-specific SSD management is beyond the scope of this paper.

We describe our experience building LightNVM, the Open-Channel SSD subsystem in the Linux kernel. LightNVM is the first open, generic subsystem for Open-Channel SSDs and host-based SSD management. We make four contributions. First, we describe the characteristics of open-channel SSD management. We identify the constraints linked to exposing SSD internals, discuss the associated trade-offs and lessons learned from the storage industry.

Second, we introduce the *Physical Page Address* (PPA) I/O interface, an interface for Open-Channel SSDs, that defines a hierarchical address space together with control and vectored data commands.

Third, we present LightNVM, the Linux subsystem that we designed and implemented for open-channel SSD management. It provides an interface where application-specific abstractions, denoted as *targets*, can be implemented. We provide a host-based Flash Translation Layer, called *pblk*, that exposes open-channel SSDs as traditional block I/O devices.

Finally, we demonstrate the effectiveness of LightNVM on top of a first generation open-channel SSD. Our results are the first measurements of an open-channel SSD that exposes the physical page address I/O interface. We compare against state-of-the-art block I/O SSD and evaluate performance overheads when running synthetic, file system, and database system-based workloads. Our results show that LightNVM achieves high performance and can be tuned to control I/O latency variability.

2 Open-Channel SSD Management

SSDs are composed of tens of storage chips wired in parallel to a controller via so-called *channels*. With open-channel SSDs, channels and storage chips are exposed to the host. The host is responsible for utilizing SSD resources in time (I/O scheduling) and space (data placement). In this section, we focus on NAND flash-based open-channel SSDs because managing NAND is both relevant and challenging today. We review the constraints imposed by NAND flash, introduce the resulting key challenges for SSD management, discuss the lessons we learned from early adopters of our system, and present different open-channel SSD architectures.

2.1 NAND Flash Characteristics

NAND flash relies on arrays of floating-gate transistors, so-called cells, to store bits. Shrinking transistor size has enabled increased flash capacity. SLC flash stores one bit per cell. MLC and TLC flash store 2 or 3 bits per cell, respectively, and there are four bits per cell in QLC flash. For 3D NAND, increased capacity is no longer tied to shrinking cell size but to flash arrays layering.

Media Architecture. NAND flash provides a read-/write/erase interface. Within a NAND package, storage media is organized into a hierarchy of die, plane, block, and page. A die allows a single I/O command to be executed at a time. There may be one or several dies within a single physical package. A plane allows similar flash commands to be executed in parallel within a die.

Within each plane, NAND is organized in blocks and pages. Each plane contains the same number of blocks, and each block contains the same number of pages. Pages are the minimal units of read and write, while the unit of erase is a block. Each page is further decomposed into fixed-size sectors with an additional out-of-bound area, e.g., a 16KB page contains four sectors of 4KB plus an out-of-bound area frequently used for ECC and user-specific data.

Regarding internal timings, NAND flash memories exhibit an order of magnitude difference between read and write/erase latency. Reads typically take sub-hundred microseconds, while write and erase actions take a few milliseconds. However, read latency spikes if a read is scheduled directly behind a write or an erase operation, leading to orders of magnitude increase in latency.

Write Constraints. There are three fundamental programming constraints that apply to NAND [41]: (i) a write command must always contain enough data to program one (or several) full flash page(s), (ii) writes must be sequential within a block, and (iii) an erase must be performed before a page within a block can be (re)written. The number of program/erase (PE) cycles is limited. The limit depends on the type of flash: 10^2 for TLC/QLC flash, 10^3 for MLC, or 10^5 for SLC.

Additional constraints must be considered for different types of NAND flash. For example, in multi-level cell memories, the bits stored in the same cell belong to different write pages, referred to as lower/upper pages. The upper page must be written before the lower page can be read successfully. The lower and upper page are often not sequential, and any pages in between must be written to prevent write neighbor disturbance [10]. Also, NAND vendors might introduce any type of idiosyncratic constraints, which are not publicly disclosed. This is a clear challenge for the design of cross-vendor, host-based SSD management.

Failure Modes. NAND Flash might fail in various ways [7, 40, 42, 49]:

- Bit Errors. The downside of shrinking cell size is an increase in errors when storing bits. While error rates of 2 bits per KB were common for SLC, this rate has increased four to eight times for MLC.

- Read and Write Disturb. The media is prone to leak currents to nearby cells as bits are written or read. This causes some of the write constraints described above.

- Data Retention. As cells wear out, data retention capability decreases. To persist over time, data must be rewritten multiple times.

- Write/Erase Error. During write or erase, a failure can occur due to an unrecoverable error at the block level. In that case, the block should be retired and data already written should be rewritten to another block.

- Die Failure. A logical unit of storage, i.e., a die on a NAND chip, may cease to function over time due to a defect. In that case, all its data will be lost.

2.2 Managing NAND

Managing the constraints imposed by NAND is a core requirement for any flash-based SSD. With open-channel SSDs, this responsibility is shared between software components running on the host (in our case a Linux device driver and layers built on top of it) and on the device controller. In this section we present two key challenges associated with NAND management: write buffering and error handling.

Write Buffering. Write buffering is necessary when the size of the sector, defined on the host side (in the Linux device driver), is smaller than the NAND flash page size, e.g., a 4KB sector size defined on top of a 16KB flash page. To deal with such a mismatch, the classical solution is to use a cache: sector writes are buffered until enough data is gathered to fill a flash page. If data must be persisted before the cache is filled, e.g., due to an application flush, then padding is added to fill the flash page. Reads are directed to the cache until data is persisted to the media. If the cache resides on the host, then the two advantages are that (1) writes are all generated by the host, thus avoiding interference between the host and devices, and that (2) writes are acknowledged as they hit the cache. The disadvantage is that the contents of the cache might be lost in case of a power failure.

The write cache may also be placed on the device side. Either the host writes sectors to the device and lets the device manage writes to the media (when enough data has been accumulated to fill a flash page), or the host explicitly controls writes to the media and lets the device maintain durability. With the former approach, the device controller might introduce unpredictability into the workload, as it might issue writes that interfere with host-issued reads. With the latter approach, the host has full access to the device-side cache. In NVMe, this can be done through a *Controller Memory Buffer* (CMB) [43]. The host can thus decouple (i) the staging of data on the device-side cache from (ii) the writing to the media through an explicit flush command. This approach avoids controller-generated writes and leaves the host in full control of media operations. Both approaches require that the device firmware has power-fail techniques to store the write buffer onto media in case of a power loss. The size of the cache is then limited by the power-capacitors available on the SSD.

Error Handling. Error handling concerns reads, writes, and erases. A read fails when all methods to recover data at sector level have been exhausted: ECC, threshold tuning, and possibly parity-based protection mechanisms (RAID/RAIN) [13, 20].

To compensate for bit errors, it is necessary to introduce Error Correcting Codes (ECC), e.g., BCH [53] or LDPC [16]. Typically, the unit of ECC encoding is a *sector*, which is usually smaller than a page. ECC parities are generally handled as metadata associated with a page and stored within the page's out-of-band area.

The bit error rate (BER) can be estimated for each block. To maintain BER below a given threshold, some vendors make it possible to tune NAND threshold voltage [7, 8]. Blocks which are write-cold and read-hot, for which BER is higher than a given threshold, should be rewritten [47]. It might also be necessary to perform *read scrubbing*, i.e., schedule read operations for the sole purpose of estimating BER for blocks which are write-cold and read-cold [9].

Given that manual threshold tuning causes several reads to be executed on a page, it may be beneficial to add RAID techniques to recover data faster, while also enable SSD to recover from die failures.

Note that different workloads might require different RAID configurations. Typically, high read workloads require less redundancy, because they issue fewer PE cycles. This is an argument for host-based RAID implementation. Conversely, for high write workloads, RAID is a source of overhead that might be compensated by hardware acceleration (i.e., a hardware-based XOR engine [14, 48]).

In the case of write failures, due to overcharging or inherent failures [51], recovery is necessary at the block level. When a write fails, part of a block might already have been written and should be read to perform recovery. Early NAND flash chips allow reads on partially written blocks, but multi-level NAND [10] requires that a set of pages (lower/upper) be written before data can be read, thus preventing reads of partially written blocks in the general case. Here, enough buffer space should be available to restore the contents of partially written blocks.

If a failure occurs on erase, there is no retry or recovery. The block is simply marked bad.

2.3 Lessons Learned

Open-channel SSDs open up a large design space for SSD management. Here are some restrictions on that design space based on industry trends and feedback from early LightNVM adopters.

1. Provide device warranty with physical access. Warranty to end-users is important in high-volume markets. A traditional SSD is often warrantied for either three or five years of operation. In its lifetime, enough good flash media must be available to perform writes. Contrary to spinning hard-drives, the lifetime for NAND media heavily depends on the number of writes to the media. Therefore, there is typically two types of guarantees for flash-based SSDs: Year warranty and Drive Writes Per Day (DWPD) warranty. DWPD guarantees that the drive can sustain X drive writes per day. Providing low thousands of PE cycles to NAND flash media, the number of writes per day is often limited to less than ten and is lower in consumer drives.

If PE cycles are managed on the host, then no warranty can be given for open-channel SSDs. Indeed, SSD vendors have no way to assess whether a device is legitimately eligible for replacement, or if flash simply wore out because of excessive usage. To provide warranty, PE cycles must be managed on the device. See Figure 1 for an illustration.

2. Exposing media characterization to the host is inefficient and limits media abstraction. Traditional SSD vendors perform media characterization with NAND vendors to adapt their embedded Flash Translation Layer to the characteristics of a given NAND chip. Such in-house NAND characterization is protected under IP. It is neither desirable nor feasible to let application and system developers struggle with the internal details of a specific NAND chip, in particular threshold tuning or ECC. These must be managed on the device. This greatly simplifies the logic in the host and lets the open-channel SSD vendor differentiate their controller implementation.

3. Write buffering should be handled on the host or the device depending on the use case. If the host handles write buffering, then there is no need for DRAM on the device, as the small data structures needed to maintain warranty and physical media information can be stored in device SRAM or persistent media if necessary. Power consumption can thus be drastically reduced. Managing the write buffer on the device, through a CMB, efficiently supports small writes but requires extra device-side logic, together with power-capacitors or similar functionality to guarantee durability. Both options should be available to open-channel SSD vendors.

4. Application-agnostic wear leveling is mandatory. As NAND ages, its access time becomes longer. Indeed, the voltage thresholds become wider, and more time must be spent to finely tune the appropriate voltage to read or write data. NAND specifications usually report both a typical access latency and a max latency. To make sure that latency does not fluctuate depending on the age of the block accessed, it is mandatory to perform wear-leveling independently from the application workload, even if it introduces an overhead.

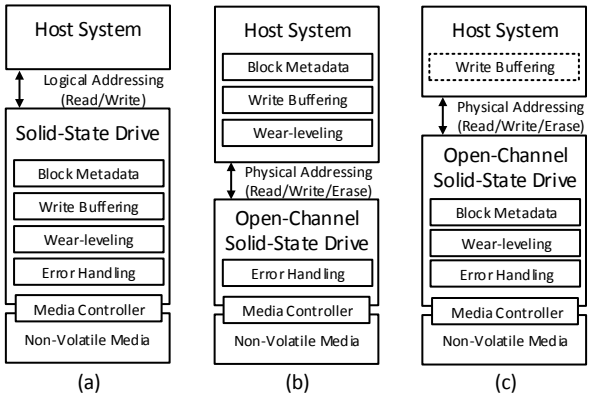


Figure 1: Core SSD Management modules on (a) a traditional Block I/O SSD, (b) the class of open-channel SSD considered in this paper, and (c) future open-channel SSDs.

It must be possible, either for the host or the controller, to pick free blocks from a die in a way that (i) hides bad blocks, (ii) implements dynamic wear leveling by taking the P/E cycle count into account when allocating a block, and possibly (iii) implements static wear leveling by copying cold data to a hot block. Such decisions should be based on metadata collected and maintained on the device: P/E cycle per block, read counts per page, and bad blocks. Managing block metadata and a level of indirection between logical and physical block addresses incurs a significant overhead regarding latency and might generate internal I/Os (to store the mapping table or due to static wear leveling) that might interfere with an application I/Os. This is the cost of wear-leveling [6].

2.4 Architectures

Different classes of open-channel SSDs can be defined based on how the responsibilities of SSD management are shared between host and SSD. Figure 1 compares (a) traditional block I/O SSD with (b) the class of open-channel SSDs considered in this paper, where PE cycles and write buffering are managed on the host, and (c) future open-channel SSDs that will provide warranties and thus support PE cycle management and wear-leveling on the device. The definition of the PPA I/O interface and the architecture of LightNVM encompass all types of open-channel SSDs.

3 Physical Page Address I/O Interface

We propose an interface for open-channel SSDs, the *Physical Page Address (PPA) I/O interface*, based on a hierarchical address space. It defines administration commands to expose the device geometry and let the host

take control of SSD management, and data commands to efficiently store and retrieve data. The interface is independent of the type of non-volatile media chip embedded on the open-channel SSD.

Our interface is implemented as a vendor-specific extension to the NVM Express 1.2.1 specification [43], a standard that defines an optimized interface for PCIe-attached SSDs.

3.1 Address Space

We rely on two invariants to define the PPA address space:

1. *SSD Architecture*. Open-channel SSDs expose to the host a collection of channels, each containing a set of *Parallel Units* (PUs), also known as LUNs. We define a PU as the unit of parallelism on the device. A PU may cover one or more physical die, and a die may only be a member of one PU. Each PU processes a single I/O request at a time.

2. *Media Architecture*. Regardless of the media, storage space is quantized on each PU. NAND flash chips are decomposed into blocks, pages (the minimum unit of transfer), and sectors (the minimum unit of ECC). Byte-addressable memories may be organized as a flat space of sectors.

The controller can choose the physical representation for the PUs. This way the controller can expose a performance model, at the PU level, that reflects the performance of the underlying storage media. If the controller chooses a logical definition for PUs (e.g., several NAND dies accessed through RAID) then the performance model for a PU must be constructed based on storage media characteristics and controller functionality (e.g., XOR engine acceleration). A logical representation might be beneficial for byte-addressable memories, where multiple dies are grouped together to form a single sector. In the rest of this paper, we assume that a PU corresponds to a single physical NAND die. With such a physical PU definition, the controller exposes a simple, well-understood, performance model of the media.

PPAs are organized as a decomposition hierarchy that reflects the SSD and media architecture. For example, NAND flash may be organized as a hierarchy of plane, block, page, and sector, while byte-addressable memories, such as PCM, is a collection of sectors. While the components of the SSD architecture, channel and PU, are present in all PPA addresses, media architecture components can be abstracted. This point is illustrated in Figure 2.

Each device defines its bit array nomenclature for PPA addresses, within the context of a 64-bit address. Put differently, the PPA format does not put constraints on the maximum number of channels per device or the maximum number of blocks per PU. It is up to each device to define these limitations, and possibly ignore some media-specific components. This flexibility is a major difference between the PPA format and the hierarchical CHS (Cylinder-Head-Sector) format introduced for early hard drives.

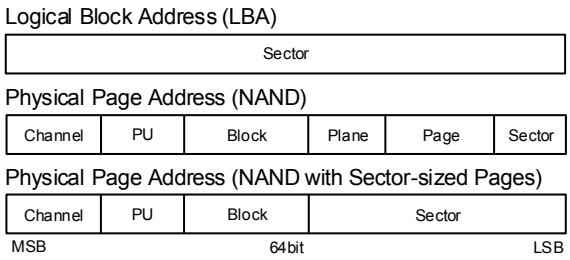


Figure 2: Logical Block Addresses compared to Physical Page Addresses for NAND flash.

Encoding the SSD and media architecture into physical addresses makes it possible to define hardware units, embedded on an open-channel SSD, that map incoming I/Os to their appropriate physical placement. The PPA format is also well-suited for a representation where the identifiers are independent variables as a power of two. This way, operations on the name-space (e.g., get next page on the next channel, or get next page on a different PU) are efficiently implemented by shifting bits.

The PPA address space can be organized logically to act as a traditional logical block address (LBA), e.g., by arranging NAND flash using "block, page, plane, and sector" [34]. This enables the PPA address space to be exposed through traditional read/write/trim commands. In contrast to traditional block I/O, the I/Os must follow certain rules. Writes must be issued sequentially within a block. Trim may be issued for a whole block, so that the device interprets the command as an erase. It is implementation specific whether a single read may cross multiple blocks in a single I/O.

In comparison with a traditional, linear LBA space, the PPA address space may contain invalid addresses, where I/Os are not accepted. Consider for example that there are 1067 available blocks per PU, then it would be represented by 11 bits. Blocks 0–1066 would be valid, while blocks 1067–2047 would be invalid. It is up to the controller to return an error in this case. In case the media configuration for each level in the hierarchy is not a power of two, then there will be such holes in the address space.

3.2 Geometry and Management

To let a host take control of SSD management, an open-channel SSD must expose four characteristics:

1. *Its geometry*, i.e., the dimensions of the PPA address space. How many channels? How many PUs within a channel? How many planes per PU? How many blocks per plane? How many pages per block? How many sectors per page? How large is the out-of-bound region per page? We assume that PPA dimensions are uniform for a given address space. If an SSD contains different types of storage chips, then the SSD must expose the storage as separate address spaces, each based on similar chips.

2. *Its performance*, i.e., statistics that capture the performance of data commands, channel capacity and controller overhead. The current version of the specification captures typical and max latency for page read, page write, and erase commands and the maximum number of in-flight commands addressed to separate PUs within a channel.

3. *Media-specific metadata*. For instance, NAND flash-specific metadata includes the type of NAND flash on the device, whether multi-plane operations are supported, the size of the user-accessible out-of-bound area, or page pairing information for MLC and TLC chips. As media evolves, and becomes more complex, it may be advantageous to let SSDs handle this complexity.

4. *Controller functionalities*. As we have seen in Section 2, a controller might support write buffering, failure handling, or provisioning. Each of these capabilities might be configured (e.g., RAID across PUs). If the controller supports write buffering, then a flush command enables the host to force the controller to write the contents of its buffer to the storage media.

3.3 Read/Write/Erase

The data commands directly reflect the read, write, and erase interface of NAND flash cells. The erase command is ignored for media that does not support it.

Vectored I/Os. Data commands expand upon traditional LBA access. A read or write command is no longer defined by a start LBA, some sectors to access, and a data buffer. Instead, a read or write is applied to a vector of addresses to leverage the intrinsic parallelism of the SSD. For example, let us consider 64KB of application data. Assuming a page size of 4KB, this data might be striped with a write command applied to 16 sectors simultaneously, thus efficiently supporting scattered access.

Concretely, each I/O is represented as an NVMe I/O read/write command. We replace the start LBA (SLBA) field with a single PPA address or a pointer to an address list, denoted *PPA list*. The PPA list contains an LBA for each sector to be accessed. Similarly, we utilize the NVMe I/O metadata field to carry out-of-band metadata. The metadata field is typically used for end-to-end data consistency (T10-PI/DIF/DIX [25, 43]). How to gracefully combine end-to-end and PPA metadata is a topic for future work.

When a data command completes, the PPA interface returns a separate completion status for each address. This way, the host can distinguish and recover from failures at different addresses. For the first iteration of the specification, the first 64 bits of the NVMe I/O command completion entry are used to signal the completion status. This limits the number of addresses in the PPA list to 64.

We considered alternatives to the PPA list. In fact, we evaluated three approaches: (i) NVMe I/O command, (ii) grouped I/Os, and (iii) Vectored I/Os. An NVMe I/O command issues commands serially. When a full page buffer is constituted, it is flushed to the media. Each command rings the doorbell of the controller to notify a new submission. With grouped I/Os, several pages constitute a submission, the doorbell is only rung once, but it is up to the controller to maintain the state of each submission. With vectored I/Os, an extra DMA is required to communicate the PPA list. We opt for the third option, as the cost of an extra DMA mapping is compensated by simplified controller design.

Media specific. Each I/O command provides media-specific hints, including plane operation mode (single, dual, or quad plane), erase/program suspend [56], and limited retry. The plane operation mode defines how many planes should be programmed at once. The controller may use the plane operation mode hint to efficiently program planes in parallel, as it accesses PUs sequentially by default. Similarly, the erase-suspend allows reads to suspend an active write or program, and thus improve its access latency, at the cost of longer write and erase time. Limited retry allows the host to let the controller know that it should not exhaust all options to read or write data, but instead fail fast to provide a better quality of service, if data is already available elsewhere.

4 LightNVM

LightNVM is the open-channel SSD subsystem in Linux. In this section, we give an overview of its architecture, and we present the pblk target in detail.

4.1 Architecture

LightNVM is organized in three layers (see Figure 3), each providing a level of abstraction for open-channel SSDs:

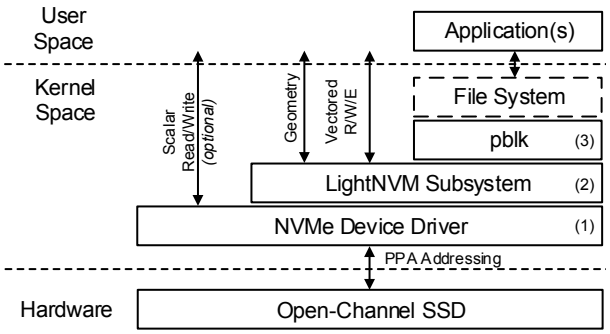


Figure 3: LightNVM Subsystem Architecture

1. **NVMe Device Driver.** A LightNVM-enabled NVMe device driver gives kernel modules access to open-channel SSDs through the PPA I/O interface. The device driver exposes the device as a traditional Linux device to user-space, which allows applications to interact with the device through ioctls. If the PPA interface is exposed through an LBA, it may also issue I/Os accordingly.

2. **LightNVM Subsystem.** An instance of the subsystem is initialized on top of the PPA I/O-supported block device. The instance enables the kernel to expose the geometry of the device through both an internal *nvm_dev* data structure and sysfs. This way FTLs and user-space applications can understand the device geometry before use. It also exposes the vector interface using the blk-mq [4] device driver private I/O interface, enabling vector I/Os to be efficiently issued through the device driver.

3. **High-level I/O Interface.** A *target* gives kernel-space modules or user-space applications access to open-channel SSDs through a high-level I/O interface, either a standard interface like the block I/O interface provided by pblk (see Section 4.2), or an application-specific interface provided by a custom target.

4.2 *pblk*: Physical Block Device

The *Physical Block Device* (pblk) is a LightNVM target implementing a fully associative, host-based FTL that exposes a traditional block I/O interface. In essence, pblk's main responsibilities are to (i) deal with controller- and media-specific constraints (e.g., caching the necessary amount of data to program a flash page), (ii) map logical addresses onto physical addresses (4KB granularity) and guarantee the integrity—and eventual recovery in the face of crashes—of the associated mapping table (L2P), (iii) handle errors, and (iv) implement garbage collection (GC). Since typical flash page sizes are bigger than 4KB, pblk must also (v) handle *flushes*. A flush forces pblk's in-flight data to be stored on the device before it completes. It might be required by a file system or an application (i.e., fsync).

4.2.1 Write Buffering

The pblk target is based on the architecture described in Section 2.4, where write buffering is managed on the host. The write buffer is managed as a circular ring buffer. It is internally decoupled into two buffers: a data buffer storing 4KB user data entries (4KB corresponds to the size of a sector), and a context buffer storing per-entry metadata. The size of the buffer is the product of flash page size (FPSZ), the number of flash pages to write (lower/upper pages), and the number of PUs (N). For example, if $FPSZ = 64KB, PP = 8, N = 128$, the write buffer is 64MB.

The write buffer is accessed by several producers and a single consumer:

Producers. Both pblk users and pblk's own garbage collector insert I/Os as entries into the write buffer. When a new entry is written, the L2P table is updated with the entry line and the write is acknowledged. If the buffer is full, the write is re-scheduled. In case that a mapping already exists for the incoming logical address, the old entry is invalidated.

Consumer. A single thread consumes buffered entries either when there is enough data to fill a flash page or when a flush command has been issued. If multi-plane programming is used then the number of planes must also be considered (e.g., 16KB pages with quad plane programming requires 64KB chunks for a single write). At this point, logical addresses are mapped to physical ones. By default, pblk's mapping strategy targets throughput and stripes data across channels and PUs at a page granularity. Other data placement strategies can be used. After mapping takes place, a vector write command is formed and sent to the device. Note that in case of a flush, if there is not enough data to fill a flash page, pblk adds padding (i.e., unmapped data) in the write command before it is sent to the device.

In order to respect the lower/upper page pairs (Section 2.1), the L2P table is not modified as pages are mapped. This way, reads are directed to the write buffer until all page pairs have been persisted. When this happens, the L2P table is updated with the physical address. L2P recovery is discussed in Section 4.2.2.

The number of channels and PUs used for mapping incoming I/Os can be tuned at run-time. We refer to them as active PUs. For example, let us consider 4 active PUs on an open-channel SSD with 4 channels and 8 PUs per channel. To start with, PU_0, PU_8, PU_{16}, and PU_{24} are

active. Pages are written on those PUs in a round-robin fashion. When a block fills up on PU_0, then that PU becomes inactive and PU_1 takes over as the active PU. At any point in time, only 4 PUs are active, but data is still striped across all available PUs at a page granularity.

When an application or file system issues a flush, pblk ensures that all outstanding data is written to the media. The consumer thread empties the write buffer and uses padding to fill up the last flash page if necessary. As data is persisted, the last write command holds an extra annotation that indicates that it must complete before the flush is successful.

4.2.2 Mapping Table Recovery

The L2P mapping table is essential for data consistency in a block device. Thus, we persist a redundant version of the mapping table in three forms: First, as a snapshot, which is stored (i) on power-down, in form of a full copy of the L2P, and (ii) periodically as checkpoints in form of an FTL log that persists operations on blocks (allocate and erase). Second, as block-level metadata, on the first and last page of each block. When a block is opened, the first page is used to store a block sequence number together with a reference to the previous block. When a block is fully written, the last pages are used to store (1) an FTL-log consisting of the portion of the L2P map table corresponding to data in the block, (2) the same sequence number as in the first page in order to avoid an extra read during recovery, and (3) a pointer to the next block. The number of pages needed depends on the size of the block. This strategy allows us to recover the FTL in an ordered manner and prevent old mappings from overwriting new ones. Finally, a portion of the mapping table is kept (iii) within the OOB area of each flash page that is written to the device. Here, for each persisted flash page, we store the logical addresses that correspond to physical addresses on the page together with a bit that signals that the page is valid. Since blocks can be reused as they are garbage collected, all metadata is persisted together with its CRC and relevant counters to guarantee consistency.

Any initialization of the device will trigger a full recovery. If an L2P mapping table snapshot is available (e.g., due to a graceful shutdown), then the mapping table is directly retrieved from disk and loaded into memory. In the case of a non-graceful shutdown, the mapping table must be recovered. We designed a two-phase recovery process.

To start with, we scan the last page of all available blocks and we classify them into free, partially written, and fully written. We can reduce the scanning by looking at the sequence numbers and only recovering written blocks. In the first phase, fully written blocks are ordered using the sequence number. The L2P table is then updated with the map portions stored on each last page. Similarly, in the second phase, partially written blocks are ordered. After this, blocks are scanned linearly until a page with an invalid bit on the OOB area is reached. Each valid mapping triggers an update in the L2P table. To ensure data correctness, it is paramount that half-written lower/upper pages are padded before reads can be issued. If the controller counts on a super capacitor, padding can be done in the device on ungraceful power-down. Otherwise, padding must be implemented on the second phase of recovery, as partially written blocks are recovered.

4.2.3 Error Handling

Unlike a traditional FTL that deals with read, write, and erase failures, pblk deals only with write and erase errors. As discussed in Section 2.2, ECC and threshold tuning are enforced by the device. If a read fails, then data is irrecoverable from the device's perspective; recovery must be managed by the upper layers of the system, above pblk.

When a write fails, pblk initiates two recovery mechanisms. First, the blocks corresponding to sectors on which a write failed are identified using the per-sector completion bits encoded in the command completion entry. These failed sectors are remapped and re-submitted to the device directly. They are not inserted in the write buffer because of the flush guarantee provided by pblk. In case a flush is attached to the failed command, subsequent writes will stop until the pointed I/O completes. Writes preceding that flush must be persisted before forward progress can be made. The second mechanism starts when the block corresponding to the failed sectors is marked as bad. Here, the remaining pages are padded and the block is sent for GC.

In the case of erase failures, the block is directly marked as bad. Since no writes have been issued at this point, there is no data to recover.

4.2.4 Garbage Collection

As any log-structured FTL, pblk must implement garbage collection. Blocks are re-purposed by garbage collecting any valid pages and returning blocks for new writes. Wear-leveling is assumed to happen either on the device or within the LightNVM core (Section 2.3). Therefore, pblk simply maintains a valid page count for each block, and selects the block with the lowest number of valid sectors for recycling.

The reverse logical to physical mapping table is not stored in host memory. To find a reverse mapping, we leverage the fact that a block is first recycled when it

is fully written. Thus, we can use the partial L2P table stored for recovery on the last pages of the block. In case a page in that block is still valid, it is queued for rewrite. When all pages have been safely rewritten, the original block is recycled.

To prevent user I/Os from interfering with garbage collection, pblk implements a PID controlled [44] rate-limiter, whose feedback loop is based on the total number of free blocks available. When the number of free blocks goes under a configurable threshold, GC starts. Note that GC can also be managed from *sysfs*. In the beginning, both GC and user I/Os compete equally for the write buffer. But as the number of available blocks decreases, GC is prioritized in order to guarantee the consistency of already persisted data. The feedback loop ensures that incoming I/Os and GC I/Os move towards a steady state, where enough garbage collection is applied given the user I/O workload. The rate-limiter uses write buffer entries as a natural way to control incoming I/Os; entries are reserved as a function of the feedback loop. If the device reaches its capacity, user I/Os will be completely disabled until enough free blocks are available.

5 Experimental Evaluation

The purpose of our experimental evaluation is threefold. First, we verify the correctness of the LightNVM stack, and we evaluate the overhead it introduces. Second, we characterize pblk on top of a first generation open-channel SSD (OCSSD) and compare it to a state-of-the-art NVMe SSD in terms of throughput, latency, and CPU utilization. We rely on *fio* [2] and application workloads for this study. Finally, we show how explicit PU write provisioning can be used to optimize I/O scheduling and achieve predictable latencies.

Our experimental setup consists of a server equipped with an Intel Xeon E5-2620v3, 32 GB of DDR4 RAM, an Open-Channel SSD (CNEX Labs Westlake SDK) with 2TB NAND MLC Flash, denoted OCSSD in the rest of this section, and an NVMe SSD (Intel P3700) with 400GB storage, denoted NVMe SSD. Both SSDs are datacenter/enterprise SSDs using MLC NAND, which makes them comparable in terms of hardware raw performance. A new instance of pblk is used for each run on the OCSSD; the NVMe SSD is formatted to 4K sector size and is low-level formatted before each run. The host runs Ubuntu 15.04 with Linux Kernel 4.8-rc4 and pblk patches applied.

The entire LightNVM stack amounts to approximately 10K LOC; pblk is responsible for almost 70% of that code.

Open-Channel Solid-State Drive	
Controller	CNEX Labs Westlake ASIC
Interface	NVMe, PCI-e Gen3x8
Channels	16
PUs per Channel	8 (128 total)
Channel Data Bandwidth	280MB/s
Parallel Unit Characteristics	
Page Size	16K + 64B user OOB
Planes	4
Blocks	1,067
Block Size	256 Pages
Type	MLC
Bandwidths	
Single Seq. PU Write	47MB/s
Single Seq. PU Read	105MB/s (4K), 280MB/s (64KB)
Single Rnd. PU Read	56MB/s (4K), 273MB/s (64KB)
Max Write	4GB/s
Max Read	4.5GB/s
pblk Factory Write (no GC)	4GB/s
pblk Steady Write (GC)	3.2GB/s

Table 1: Solid-State Drive Characterization.

5.1 Sanity Check

Table 1 contains a general characterization for the evaluated Open-Channel SSD. Per-PU sequential read and write bandwidth were gathered experimentally through a modified version [3] of fio that uses the PPA I/O interface and issues vector I/Os directly to the device. The pblk factory state and steady state (where garbage collection is active) are measured experimentally through standard fio on top of pblk. Note that we leave the detailed characterization of pblk for future work and only prove that the implementation works as expected. Unless specified otherwise, each experiment is conducted in factory state with pblk's rate-limiter disabled.

In terms of CPU utilization, pblk introduces an overhead of less than 1% CPU overhead for reads with 0.4μs additional latency (2.32μs with, and 1.97μs without, a difference of 18%). While for writes, it adds 4% CPU overhead with an additional 0.9μs latency (2.9μs with, and 2μs without, a difference of 45%). Overhead on the read path is due to an extra lookup into the L2P table and the overhead on the write path is due to buffer and device write I/O requests management. CPU overhead is measured by comparing the time it takes with and without pblk on top of a null block device [4] and does not include device I/O timings.

5.2 Uniform Workloads

Figure 4 captures throughput and latency for sequential and random reads issued with fio on 100GB of data. The preparation for the test has been performed with pblk using the full bandwidth of the device (128 PUs). This means that sequential reads are more easily parallelized internally by the controller since sequential logical addresses are physically striped across channels and PUs on a per-page basis.

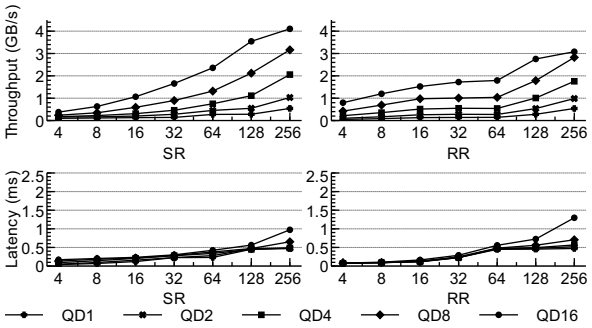


Figure 4: Throughput and corresponding latencies for sequential and random read workloads as a function of queue depths (QD) and block sizes (x axis in KB).

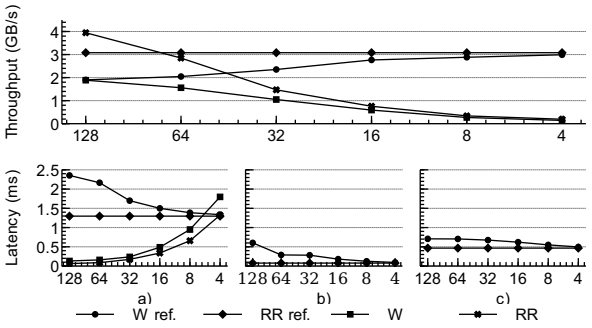


Figure 5: R/W throughput and latencies as a function of active write PUs configurations. Top graph, and a: throughput and corresponding latency (Writes: 256KB, QD1; Reads: 256KB, QD16); b: read latency (Writes: 256KB, QD1; Reads: 4KB, QD1); c: read latency (Write: 256KB, QD1, rate-limited at 200MB/s; Reads: 256KB, QD1).

We see that the OCSSD is capable of reaching 4GB/s using sequential reads at an average latency of 970μs using 256KB request size and a queue depth of 16. The 99^{th} percentile latency is reported at $1,200\mu$s. Similarly, we measure throughput and latency for 4KB reads using a queue depth of 1. Maximum throughput is 105MB/s, with 40μs average access latency and 99^{th} percentile at 400μs. The average access latency is lower than a single flash page access because the controller caches the flash page internally. Thus, all sectors located on the same flash page will be served from the controller buffer instead of issuing a new flash page read. Also, read throughput is limited by the flash page access time, as we only perform one read at a time.

Pure read and write workloads can be used to calibrate queue depths to reach full bandwidth. They show the optimal case, where reads and writes do not block each other. Let us now discuss mixed workloads, which are much more challenging for SSDs.

5.3 Mixed Workloads

For a mixed workload, we use the same write preparation as in the previous experiment (stripe across all PUs with a 100GB dataset).

Then, we proceed to write with an offset of 100GB, while we read from the first 100GB. We repeat this experiment, varying stripe size (number of active write PUs) for new writes. The hypothesis is that as the stripe size decreases, read performance predictability should increase as the probability of a read being stuck behind a write lowers.

Figure 5 depicts the behavior of pblk when reads and writes are mixed. In Figure 5(a), we show throughput for both writes and random reads together with their reference value, represented by 100% writes (4GB/s–200MB/s) and 100% random reads (3GB/s), respectively; Figure 5(b) depicts its latencies. The experiment consists of large sequential 256KB writes at queue depth 1, and 256KB random reads at queue depth 16. The write queue depth is 1, as it is enough to satisfy the full write bandwidth (defined by the capacity of the pblk write buffer). Note that reads are issued at queue depth 16 so that enough parallelism can be leveraged by the controller. This allows us to better visualize the worst-case latencies and the effect of fewer writing PUs.

We observe that when new writes are striped across all 128 PUs, throughput is halved for both reads and writes compared to the reference value, while average latency doubles for reads (maximum latency is increased by $4\times$). Write latencies are close to not being affected because they are buffered. This represents the typical case on a traditional SSD: reads are stacked behind writes, thus affecting read performance; host and controller queues are filled with read requests, thus affecting write performance. However, as soon as we start limiting the number of active write PUs, we observe how reads rapidly recover. For this experiment, we configured one block to be fully written on an active PU before switching to a different PU. Writes are still striped across all 128 PUs, but instead of being striped at page granularity, they are striped at block granularity. This lowers the probability of reads being issued to the same PU as new writes (because, reads and writes are striped at different granularities). If we lower the number of write-active PUs to 4, we see that reads are very close to the reference read workload, while still writing at 200MB/s.

Figure 5(c) shows latency for 4K reads at queue depth 1. Here, we emphasize the impact of a read being blocked by a write. As in Figure 5(b), latency variance reduces as we decrease the number of active write PUs. With 4 active write PUs, the maximum latency for random reads in the 99^{th} percentile is only 2μs higher than in the average case.

Figure 5(d) shows the same experiment as in a) and b), with the difference that writes are rate-limited to

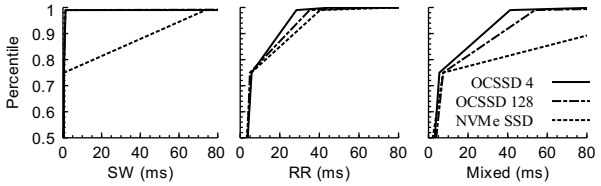


Figure 6: Latencies for RocksDB sequential writes, random reads and mixed workloads on OCSSD and NVMe SSD

	NVMe SSD	OCSSD 128	OCSSD 4
SW	276	396	80
RR	5064	5819	5319
Mixed	2208	3897	4825

Table 2: Throughput (MB/s) for RocksDB sequential writes, random reads, and mixed workloads on OCSSD and NVMe SSD

200MB/s. The motivation for this experiment is the expectation of consistent writes in next-generation SSDs. Note that current SSDs already define the maximal sustained write bandwidths over a three-year period. Examples are write-heavy (e.g., Intel DC P3608, 1.6GB, 5 DWPD) and read-heavy (e.g., Samsung 960 Pro, 2TB, 2.2 DWPD) SSDs, where the limits are 95MB/s and 26MB/s, respectively. The interesting output of this experiment is that even when writes are rated, the variance of reads is still very much affected by the number of active write PUs.

More generally, the experiments with mixed workloads show that informed decisions based on the actual workload of an application can be leveraged to optimize a traditional block device interface, without requiring an application-specific FTL.

5.4 Application Workloads

We evaluate pblk with a NoSQL database, and MySQL with both OLTP and OLAP workloads. The NoSQL database relies on an LSM-tree for storage and leans towards fewer flushes (sync is enabled to guarantee data integrity), while MySQL has tight bounds on persisting transactional data to disk. We evaluate both using the NVMe SSD and the OCSSD using 128 and 4 active write PUs.

NoSQL. For this experiment, we ran RocksDB [17] on top of an Ext4 file system and made use of RocksDB's *db_bench* to execute three workloads: sequential writes, random reads, and mixed (RocksDB *read-while-writing* test). Figure 6 shows user throughput and latencies for the three workloads. We show latency for the 95^{th}, 99^{th} and 99.9^{th} percentile of the latency distribution. Note that internally RocksDB performs its own garbage collection (i.e., sstable compaction). This consumes device bandwidth, which is not reported by db_bench.

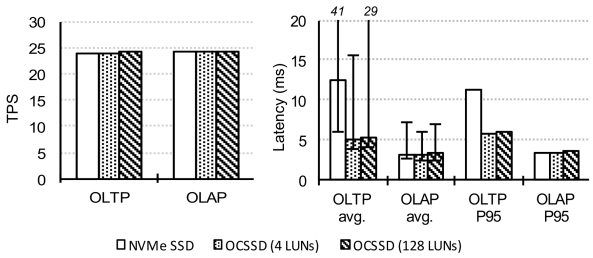


Figure 7: Transactions per second and latencies for OLTP and OLAP on NVMe SSD and OCSSD.

The user write throughput is 276MB/s for the NVMe SSD, 396MB/s for the OCSSD with 128 active PUs, and 88MB/s with 4 active PUs. The fewer active PUs clearly show that the write performance is limited. The performance of the random reads workload is comparable for both SSDs. There is a significant difference when writes are involved. First, both SSDs expose the same behavior for sequential workloads until we reach the 99.9^{th} percentile, where the OCSSD provides a lower latency, by a factor of two. Second, for mixed workload, the OCSSD provides a much lower latency (approximately a factor of three) already for the 99^{th} percentile. This is because reads get much more often stuck after writes on the NVMe SSD and that the OCSSD has more internal parallelism that can be leveraged by writes.

OLTP and OLAP. Figure 7 shows Sysbench's [32] OLTP and OLAP workloads on top of the MySQL database system and an Ext4 file system. The latency error bounds show the min/max for the workloads as well.

Both workloads are currently CPU bound and thus similar for all SSDs. When writing, however, the OLTP workload exhibits significant flush overheads. For 10GB write, 44,000 flushes were sent, with roughly 2GB data padding applied. For OLAP (as for RocksDB), only 400 flushes were sent, with only 16MB additional padding. Thus, for a transactional workload, a device-side buffer would significantly reduce the amount of padding required.

The latency results show the same trend as RocksDB. In the 95^{th} percentile, latency increases a lot for write-heavy OLTP on the traditional SSD compared to the average case, while the increase is insignificant for the open-channel SSD. For OLAP, the results are similar both in terms of throughput and latency due to the workload being CPU-intensive, and mostly read-only. Thus, there is no interference between reads and writes/erases. Tuning SQL databases for performance on open-channel SSDs is an interesting topic for future work (possibly via a KV-based storage engine [39]).

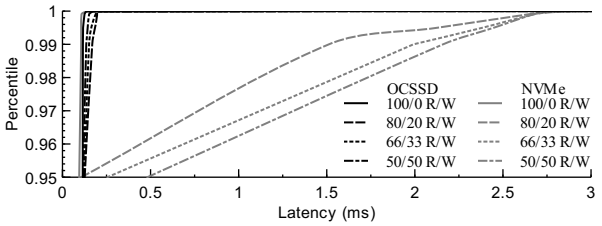


Figure 8: Latency comparison of OCSSD and NVMe SSD showing how writes impact read latencies. Note that the y-axis does not start at 0, but at 0.95.

5.5 Predictable Latency

This experiment illustrates the potential benefits of application-specific FTLs. We use again the modified version of fio to run two concurrent streams of vector I/Os directly to the device. One thread issues 4KB random reads at queue depth 1, while another thread issues 64K writes at the same queue depth. The streams for the OCSSD are isolated to separate PUs, while the NVMe SSD mixes both reads and writes. We measure the workload over five seconds and report the latency percentiles in Figure 8. We report the latency granularities as $100/0, 80/20, 66/33$, and $50/50$. As writes increase, performance remains stable on the OCSSD. While the NVMe SSDs has no method to separate the reads from the writes, and as such higher read latency are introduced even for light workloads (20% writes).

Our point is that the PPA I/O interface enables application developers to explicitly manage the queue for each separate PU in an SSD and thus achieve predictable I/O latency. Characterizing the potential of application-specific FTLs with open-channel SSDs is a topic for future work.

6 Related Work

As SSD shortcomings become apparent [12, 15, 30, 37], research has focused on organizing the cooperation between host and SSDs. One form of cooperation consists of passing hints from hosts to embedded FTLs. The work on multi-streaming falls in this category [28]. Other forms of cooperation consist in bypassing the FTL [24, 35, 50], or designing the upper layers of the system around the properties of a given FTL [29, 33, 36, 38, 46]. Finally, host-based FTLs let the host control data placement and I/O scheduling. This is the approach we have taken with LightNVM.

Host-side FTLs have been implemented by both FusionIO DFS [27] and Violin Memory [54], each moving the FTL into the host in order to expose a block I/O SSD. Similarly, Ouyang et al. [45] proposed Software-Defined Flash that allows a key-value store to integrate with the underlying storage media. Also, Lu et al. [37] defined a host-based object-based FTL (OFTL) on top of raw flash devices that correspond to Baidu's open-channel SSDs. In contrast, we specify a cross-vendor interface for open-channel SSDs, and we show how a tunable block I/O target can reduce read latency variability.

Lee et. al. [34] proposed a new SSD interface, compatible with the legacy block device interface, that exposes error-free append-only segments through read/write/trim operations. This work is based on a top-down approach, which shows how a state-of-art file system (F2FS) can be implemented on top of the proposed append-only interface. Our paper, in contrast, describes a bottom-up approach where the PPA interface reflects SSD characteristics independently of the upper layers of the system. First, our PPA interface allows write and erase errors to propagate up to the host for increased I/O predictability. We also define a vector I/O interface, allowing the host to leverage the device parallelism. Finally, we explicitly expose the read and write granularity of the media to the host so that write buffering can be placed on the host or the device. Demonstrating the benefits of application-specific SSD management with LightNVM is a topic for future research. Initial results with RocksDB [22] or multi-tenant I/O isolation [21, 26] are promising.

7 Conclusion

LightNVM is the open-channel SSD subsystem in the Linux kernel. It exposes any open-channel SSD to the host through the PPA I/O interface. LightNVM also provides a partition manager and a tunable Block I/O interface. Our experimental results show that LightNVM provides (i) low overhead with significant flexibility, (ii) reduced read variability compared to traditional NVMe SSDs, and (iii) the possibility of obtaining predictable latency. Future work includes characterizing the performance of various open-channel SSD models (products from three vendors have been announced at the time of writing), devising tuning strategies for relational database systems and designing application-specific FTLs for key-value stores.

8 Acknowledgments

We thank the anonymous reviewers and our shepherd, Erez Zadok, whose suggestions helped improve this paper. We also thank Alberto Lerner, Mark Callaghan, Laura Caulfield, Stephen Bates, Carla Villegas Pasco, Björn Þór Jónsson, and Ken McConlogue for their valuable comments during the writing of this paper. We thank the Linux kernel community, including Jens Axboe and Christoph Hellwig, for providing feedback on upstream patches, and improving the architecture.

References

[1] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference (ATC)* (2008), pp. 57–70.

[2] AXBOE, J. Fio - Flexible I/O tester. *URL http://freecode. com/projects/fio* (2014).

[3] BJØRLING, M. fio LightNVM I/O Engine. *URL https://github.com/MatiasBjorling/lightnvm-fio* (2016).

[4] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)* (2013).

[5] BJØRLING, M., BONNET, P., BOUGANIM, L., AND DAYAN, N. The necessary death of the block device interface. In *6th Biennial Conference on Innovative Data Systems Research (CIDR)* (2013), pp. 1–4.

[6] BONNET, P., AND BOUGANIM, L. Flash device support for database management. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings* (2011), pp. 1–8.

[7] 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), IEEE, pp. 521–526.

[8] 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* (San Jose, CA, USA, 2013).

[9] CAI, Y., LUO, Y., GHOSE, S., AND MUTLU, O. Read disturb errors in MLC NAND flash memory: Characterization, mitigation, and recovery. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2015), IEEE, pp. 438–449.

[10] CAI, Y., MUTLU, O., HARATSCH, E. F., AND MAI, K. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *IEEE 31st International Conference on Computer Design (ICCD)* (2013).

[11] CHEN, F., LEE, R., AND ZHANG, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *IEEE 17th International Symposium on High Performance Computer Architecture* (2011), IEEE, pp. 266–277.

[12] CHEN, F., LUO, T., AND ZHANG, X. CAFTL : A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. *9th USENIX Conference on File and Storage Technologies (FAST)* (2011).

[13] CRUCIAL. The Crucial M550 SSD. `http://www.crucial.com/usa/en/storage-ssd-m550`, 2013.

[14] CURRY, M. L., SKJELLUM, A., WARD, H. L., AND BRIGHTWELL, R. Accelerating reed-solomon coding in raid systems with gpus. In *IEEE International Symposium on Parallel and Distributed Processing* (April 2008), pp. 1–6.

[15] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80.

[16] DONG, G., XIE, N., AND ZHANG, T. On the use of soft-decision error-correction codes in NAND flash memory. *IEEE Transactions on Circuits and Systems I: Regular Papers 58*, 2 (2011), 429–439.

[17] DONG, S., CALLAGHAN, M., GALANIS, L., BORTHAKUR, D., SAVOR, T., AND STUMM, M. Optimizing Space Amplification in RocksDB. *8th Biennial Conference on Innovative Data Systems Research (CIDR)* (2017).

[18] FILKS, V., UNSWORTH, J., AND CHANDRASEKARAN, A. Magic Quadrant for Solid-State Arrays. Tech. rep., Gartner, 2016.

[19] FLOYER, D. Flash Enterprise Adoption Projections. Tech. rep., Wikibon, 2016.

[20] FUSION-IO. Introduces "Flashback" Protection Bring Protective RAID Technology to Solid State Storage and Ensuring Unrivaled Reliability with Redundancy. Businesswire, 2008.

[21] GONZÁLEZ, J., AND BJØRLING, M. Multi-Tenant I/O Isolation with Open-Channel SSDs. *Nonvolatile Memory Workshop (NVMW)* (2017).

[22] GONZÁLEZ, J., BJØRLING, M., LEE, S., DONG, C., AND HUANG, Y. R. Application-Driven Flash Translation Layers on Open-Channel SSDs. *Nonvolatile Memory Workshop (NVMW)* (2014).

[23] HAO, M., SOUNDARARAJAN, G., KENCHAMMANA-HOSEKOTE, D., CHIEN, A. A., AND GUNAWI, H. S. The tail at store: a revelation from millions of hours of disk and SSD deployments. In *14th USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 263–276.

[24] HARDOCK, S., PETROV, I., GOTTSTEIN, R., AND BUCHMANN, A. Noftl: Database systems on ftl-less flash storage. *Proceedings of the VLDB Endowment 6*, 12 (2013), 1278–1281.

[25] HOLT, K. Information technology—scsi block commands—3 (sbc-3). Tech. rep., T10/03-224, Working Draft, 2003.

[26] HUANG, J., BADAM, A., CAULFIELD, L., NATH, S., SENGUPTA, S., SHARMA, B., AND QURESHI, M. K. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *15th USENIX Conference on File and Storage Technologies (FAST)* (2017), USENIX.

[27] JOSEPHSON, W. K., BONGO, L. A., LI, K., AND FLYNN, D. DFS: A File System for Virtualized Flash Storage. *ACM Transactions on Storage 6*, 3 (Sept. 2010), 1–25.

[28] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The Multi-streamed Solid-State Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2014).

[29] KANG, W.-H., LEE, S.-W., MOON, B., OH, G.-H., AND MIN, C. X-FTL: transactional FTL for SQLite databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2013), pp. 97–108.

[30] KIM, J., LEE, D., AND NOH, S. H. Towards SLO Complying SSDs Through OPS Isolation. In *13th USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 183–189.

[31] KING, M., DELLETT, L., AND SULLIVAN, K. Annual High Performance Computing Trends Survey. Tech. rep., DDN, 2016.

[32] KOPYTOV, A. Sysbench: a system performance benchmark. *URL: http://sysbench. sourceforge. net* (2004).

[33] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 273–286.

[34] LEE, S., LIU, M., JUN, S., XU, S., KIM, J., AND ARVIND. Application-Managed Flash. In *14th USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 339–353.

[35] LIU, M., JUN, S.-W., LEE, S., HICKS, J., ET AL. minFlash: A minimalistic clustered flash array. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2016), IEEE, pp. 1255–1260.

[36] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 133–148.

[37] LU, Y., SHU, J., AND ZHENG, W. Extending the Lifetime of Flash-based Storage Through Reducing Write Amplification from File Systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (2013), USENIX Association, pp. 257–270.

[38] MÁRMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. *6th USENIX Workshop on Hot Topics in Storage and File Systems* (2014).

[39] MATSUNOBU, Y. Rocksdb storage engine for mysql. In *FOSDEM 2016* (2016).

[40] MEZA, J., WU, Q., KUMAR, S., AND MUTLU, O. A large-scale study of flash memory failures in the field. In *ACM SIGMETRICS Performance Evaluation Review* (2015), vol. 43, ACM, pp. 177–190.

[41] MICHELONI, R., MARELLI, A., AND ESHGHI, K. *Inside solid state drives (SSDs)*, vol. 37. Springer Science & Business Media, 2012.

[42] NARAYANAN, I., WANG, D., JEON, M., SHARMA, B., CAULFIELD, L., SIVASUBRAMANIAM, A., CUTLER, B., LIU, J., KHESSIB, B., AND VAID, K. SSD Failures in Datacenters: What, When and Why? In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (2016), ACM, pp. 407–408.

[43] NVMHCI WORK GROUP. NVM Express 1.2.1, 2016.

[44] O'DWYER, A. *Handbook of PI and PID controller tuning rules*, vol. 57. World Scientific, 2009.

[45] OUYANG, J., LIN, S., JIANG, S., AND HOU, Z. 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* (2014).

[46] OUYANG, X., NELLANS, D., WIPFEL, R., FLYNN, D., AND PANDA, D. Beyond block I/O: Rethinking traditional storage primitives. In *High Performance Computer Architecture (HPCA)* (2011), IEEE, pp. 301–311.

[47] PAN, Y., DONG, G., AND ZHANG, T. Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance. In *FAST* (2011), vol. 11, pp. 18–18.

[48] PLANK, J. S., GREENAN, K. M., AND MILLER, E. L. Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions. In *11th USENIX Conference on File and Storage Technologies (FAST)* (2013), USENIX Association, pp. 298–306.

[49] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 67–80.

[50] SONG, Y. H., JUNG, S., LEE, S.-W., AND KIM, J.-S. Cosmos OpenSSD: A PCIe-based Open Source SSD Platform OpenSSD Introduction. *Flash Memory Summit* (2014), 1–30.

[51] SUN, H., GRAYSON, P., AND WOOD, B. Quantifying reliability of solid-state storage from multiple aspects. *7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os* (2011).

[52] SWANSON, S., AND CAULFIELD, A. M. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *IEEE Computer 46*, 8 (2013).

[53] SWEENEY, P. *Error control coding*. Prentice Hall UK, 1991.

[54] VIOLIN MEMORY. All Flash Array Architecture, 2012.

[55] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)* (2014), 1–14.

[56] WU, G., AND HE, X. Reducing ssd read latency via nand flash program and erase suspension. In *11th USENIX Conference on File and Storage Technologies (FAST)* (2012), vol. 12, pp. 10–10.

[57] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., AND SUNDARARAMAN, S. Don't stack your log on my log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)* (2014).

FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs

Jian Huang[†] Anirudh Badam Laura Caulfield

Suman Nath Sudipta Sengupta Bikash Sharma Moinuddin K.Qureshi[†]

[†]Georgia Institute of Technology Microsoft

Abstract

A longstanding goal of SSD virtualization has been to provide performance isolation between multiple tenants sharing the device. Virtualizing SSDs, however, has traditionally been a challenge because of the fundamental tussle between resource isolation and the lifetime of the device – existing SSDs aim to uniformly age all the regions of flash and this hurts isolation. We propose utilizing flash parallelism to improve isolation between virtual SSDs by running them on dedicated channels and dies. Furthermore, we offer a complete solution by also managing the wear. We propose allowing the wear of different channels and dies to diverge at fine time granularities in favor of isolation and adjusting that imbalance at a coarse time granularity in a principled manner. Our experiments show that the new SSD wears uniformly while the 99th percentile latencies of storage operations in a variety of multi-tenant settings are reduced by up to 3.1x compared to software isolated virtual SSDs.

1 Introduction

SSDs have become indispensable for large-scale cloud services as their cost is fast approaching to that of HDDs. They out-perform HDDs by orders of magnitude, providing up to 5000x more IOPS, at 1% of the latency [21]. The rapidly shrinking process technology has allowed SSDs to boost their bandwidth and capacity by increasing the number of chips. However, the limitations of SSDs' management algorithms have hindered these parallelism trends from efficiently supporting multiple tenants on the same SSD.

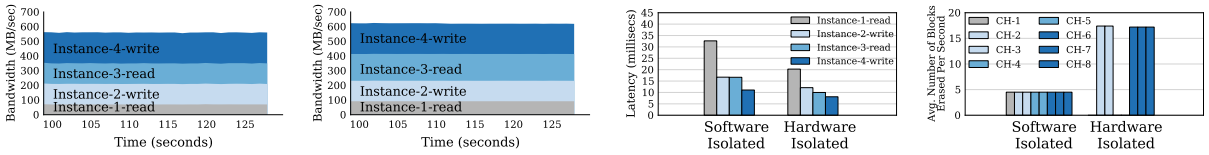
Tail latency of SSDs in multi-tenant settings is one such limitation. Cloud storage and database systems have started colocating multiple tenants on the same SSDs [14, 58, 79] which further exacerbates the already well known tail latency problem of SSDs [25, 26, 60, 78].

The cause of tail latency is the set of complex flash management algorithms in the SSD's controller, called the Flash Translation Layer (FTL). The fundamental goals of these algorithms are decades-old and were meant for an age when SSDs had limited capacity and little parallelism. The goals were meant to hide the idiosyncrasies of flash behind a layer of indirection and expose a block interface. These algorithms, however, conflate wear leveling (to address flash's limited lifetime) and resource utilization (to exploit parallelism) which increases interference between tenants sharing an SSD.

While application-level flash-awareness [31, 36, 37, 51, 75] improves throughput by efficiently leveraging the device level parallelism, these optimizations do not directly help reduce the interference between multiple tenants sharing an SSD. These tenants cannot effectively leverage flash parallelism for isolation even when they are individually flash-friendly because FTLs hide the parallelism. Newer SSD interfaces [38, 49] that propose exposing raw parallelism directly to higher layers provide more flexibility in obtaining isolation for tenants but they complicate the implementation of wear-leveling mechanisms across the different units of parallelism.

In this work, we propose leveraging the inherent parallelism present in today's SSDs to increase isolation between multiple tenants sharing an SSD. We propose creating virtual SSDs that are pinned to a dedicated number of channels and dies depending on the capacity and performance needs of the tenant. The fact that the channels and dies can be more or less operated upon independently helps such virtual SSDs avoid adverse impacts on each other's performance. However, different workloads can write at different rates and in different patterns, this could age the channels and dies at different rates. For instance, a channel pinned to a TPC-C database instance wears out 12x faster than a channel pinned to a TPC-E database instance, reducing the SSD lifetime dramatically. This non-uniform aging creates an unpredictable SSD lifetime behavior that complicates both provisioning and load-balancing aspects of data center clusters.

To address this problem, we propose a two-part wear-leveling model which balances wear within each virtual SSD and across virtual SSDs using separate strategies. Intra-virtual SSD wear is managed by leveraging existing SSD wear-balancing mechanisms while inter-virtual



(a) Software Isolated Instances (b) Hardware Isolated Instances (c) 99th Percentile Latency (d) Wear Leveling

Figure 1: Tenants sharing an SSD get better bandwidth (compare (a) vs. (b)) and tail latency as shown in (c) when using new hardware isolation. However, dedicating channels to tenants can lead to wear-imbalance between the various channels as shown in (d). Note that the number of blocks erased in the first, fourth and fifth channels is close to zero because they host workloads with only read operations. This imbalance of write bandwidth across different workloads creates wear-imbalance across channels. A new design for addressing such a wear-imbalance is proposed in this paper.

SSD wear is balanced at *coarse-time granularities* to reduce interference by using new mechanisms. We control the wear imbalance between virtual SSDs using a mathematical model and show that the new wear-leveling model ensures near-ideal lifetime for the SSD with negligible disruption to tenants. More specifically, this work makes the following contributions:

- We present a system named FlashBlox using which tenants can share an SSD with minimal interference by working on dedicated channels and dies.
- We present a new wear-leveling mechanism that allows measured amounts of wear imbalance to obtain better performance isolation between such tenants.
- We present an analytical model and a system that control the wear imbalance between channels and dies, so that they age uniformly with negligible interruption to the tenants.

We design and implement FlashBlox and its new wear-leveling mechanisms inside an open-channel SSD stack (from CNEX labs [18]), and demonstrate benefits for a Microsoft data centers' multi-tenant storage workloads: the new SSD delivers up to 1.6x better throughput and reduces the 99th percentile latency by up to 3.1x. Furthermore, our wear leveling mechanism provides 95% of the ideal SSD lifetime even in the presence of adversarial write workloads that execute all the writes on a single channel while only reading on other channels.

The rest of this paper is organized as follows: § 2 presents the challenges that we address in this work. Design and implementation of FlashBlox are described in § 3. Evaluation results are shown in § 4. § 5 presents the related work. We present the conclusions in § 6.

2 SSD Virtualization: Opportunity and Challenges

Premium storage Infrastructure-as-a-Service (IaaS) offerings [4, 7, 22], persistent Platform-as-a-Service (PaaS) systems [8] and Database-as-a-Service (DaaS) systems [2, 5, 9, 23] need SSDs to meet their service level objectives (SLO) that are usually outside the scope of HDD performance. For example, DocDB [5] guarantees 250, 1,000 and 2,500 queries per second respectively for the S1, S2 and S3 offerings [6].

Storage virtualization helps such services make efficient use of SSDs' high capacity and performance by slicing resources among multiple customers or instances. Typical database instances in DaaS systems are 10 GB – 1 TB [6, 10] whereas each server can have more than 20 TB of SSD capacity today.

Bandwidth, IOPS [48, 56] or a convex combination of both [57, 74] is limited on a per-instance basis using token bucket rate limiters or intelligent IO throttling [41, 59, 66] to meet SLOs. However, there is no analogous mechanism for sharing the SSD while maintaining low IO tail latency – an instance's latency still depends on the foreground reads/writes [25, 42, 73] and background garbage collection [34] of other instances.

Moreover, it is becoming increasingly necessary to co-locate diverse workloads (e.g. latency-critical applications and batch processing jobs), to improve resource utilization, while maintaining isolation [33, 42]. Virtualization and container technologies are evolving to exploit hardware isolation of memory [11, 47], CPU [16, 40], caches [28, 52], and networks [30, 72] to support such scenarios. We extend this line of research to SSDs by providing hardware-isolated SSDs, complete with a solution for the wear-imbalance problem that arises due to the physical flash partitioning across tenants with diverse workloads.

2.1 Hardware Isolation vs. Wear-Leveling

To understand this problem, we compare the two different approaches to sharing hardware. The first approach stripes data from all the workloads across all the flash channels (eight total), just as existing SSDs do. This scheme provides the maximum throughput for each IO, and uses the software rate limiter which has been used for Linux containers and Docker [12, 13] to implement weighted fair sharing of the resources (the scenario for Figure 1(a)). Note that instances in the software-isolated case do not share physical flash blocks with other colocated instances. This eliminates the interference due to

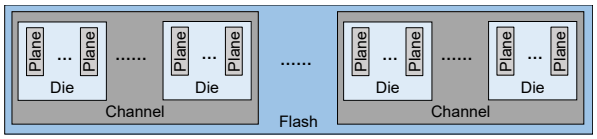


Figure 2: SSD Architecture: Internal parallelism in SSDs creates opportunities for hardware-level isolation.

garbage collection in one instance affecting another instance's read performance [34]. The second approach uses a configuration from our proposed mechanism that provides the hardware isolation by assigning a certain number of channels to each instance (the scenario for Figure 1(b)).

In both scenarios, there are four IO-intensive workloads. These workloads request 1/8th, 1/4th, 1/4th and 3/8th of the shared storage resource. The rate limiter uses these as weights in the first approach, while FlashBlox assigns 1, 2, 2 and 3 channels respectively. Workloads 2 and 4 perform 100% writes and workloads 1 and 3 perform 100% reads. All workloads issue sequentially-addressed and aligned 64 KB IOs.

Hardware isolation not only reduces the 99th percentile latencies by up to 1.7x (Figure 1(c)), but also increases the aggregate throughput by up to 10.8% compared to software isolation. However, pinning instances to channels prevents the hardware from automatically leveling the wear across all the channels, as shown in Figure 1(d). We exaggerate the variance of write rates to better motivate the problem of wear-imbalance that stems from hardware-isolation of virtual SSDs. Later in the paper, we will use applications' typical write rates (see Figure 5) to design our final solution. To motivate the problem further, we must first explore the parallelism available in SSD hardware, and the aspects of FTLs which cause interference in the first approach.

2.2 Leveraging Parallelism for Isolation

Typical SSDs organize their flash array into a hierarchy of channels, dies, planes, blocks and pages [1, 17]. As shown in Figure 2, each SSD has multiple channels, each channel has multiple dies, and each die has multiple planes. The number of channels, dies and planes varies by vendor and generation. Typically, there are 2 - 4 planes per die, 4 - 8 dies per channel, and 8 - 32 channels per drive. Each plane is composed of thousands of blocks (typically 4-9MB) and each block contains 128-256 pages.

This architecture plays an important role in defining isolation boundaries. Channels, which share only the resources common to the whole SSD, provide the strongest isolation. Dies execute their commands with complete independence, but they must share a bus with other dies on the same channel. Planes' isolation is limited because the die contains only one address buffer. The controller may isolate data to different planes, but operations on these data must happen at different times or to the same address on each plane in a die [32].

In current drives, none of this flexibility is exposed to the host. Drives instead optimize for a single IO pattern: extremely large or sequential IO. The FTL logically groups all planes into one large unit, creating "super-pages" and "super-blocks" are hundreds of times larger than their base unit. For example, a drive with 4MB blocks and 256 planes has a 1GB super-block.

Striping increases the throughput of large, sequential IOs, but introduces the negative side effect of interference between multiple tenants sharing the drive. As all data is striped, every tenant's reads, writes and erases can potentially conflict with every other tenant's operations.

Previous work had proposed novel techniques to help tenants place their data such that underlying flash pages are allocated from separate blocks. This helps improve performance by reducing the write amplification factor (WAF) [34]. Lack of block sharing has the desirable side effect of clumping garbage into fewer blocks, leading to more efficient garbage collection (GC), thereby reducing tail latency of SSDs [25, 42, 43, 73].

However, significant interference still exists between tenants because when data is striped, every tenant uses every channel, die and plane for storing data and the storage operations of one tenant can delay other tenants. Software isolation techniques [57, 67, 68] split the the SSD's resources fairly. However, they cannot maximally utilize the flash parallelism when resource contention exists at a layer below because of the forced sharing of independent resources such as channels, dies and planes.

New SSD designs, such as open-channel SSDs that explicitly expose channels, dies and planes to the operating system [44, 38, 49], can help tenants who share an SSD avoid some of these pitfalls by using dedicated channels. However, the wear imbalance problem between channels that ensues from different tenants writing at different rates remains unsolved. We propose a holistic approach to solve this problem by exposing flash channels and dies as virtual SSDs, while the system underneath wear-levels within each vSSD and balances the wear across channels and dies at coarse time granularities.

FlashBlox is concerned only with sharing of the resources within a single NVMe SSD. Fair sharing mechanisms that split PCIe bus bandwidth across multiple NVMe devices, network interface cards, graphic processing units and other PCIe devices is beyond the scope of this work.

3 Design and Implementation

Figure 3 shows the FlashBlox architecture. At a high level, FlashBlox consists of the following three components: (1) A resource manager that allows tenants to al-

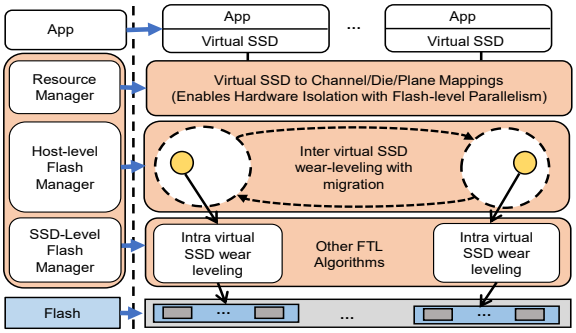


Figure 3: The system architecture of FlashBlox.

Table 1: Virtual SSD types supported in FlashBlox.

Virtual SSD Type	Isolation Level	Alloc. Granularity
Channel Isolated vSSD (§ 3.1)	High	Channel
Die Isolated vSSD (§ 3.2)	Medium	Die
Software Isolated vSSD (§ 3.3)	Low	Plane/Block
Unisolated vSSD (§ 3.3)	None	Block/Page

locate and deallocate virtual SSDs (vSSD); (2) A host-level flash manager that implements inter-vSSD wear-levelling by balancing wear across channels and dies at coarse time granularities; (3) An SSD-level flash manager that implements intra-vSSD wear-levelling and other FTL functionalities.

One of the key new abstractions provided by FlashBlox is that of a virtual SSD (vSSD) which can reduce tail latency. It uses dedicated flash hardware resources such as channels and dies that can be operated independently from each other. The following API creates a vSSD:

```
vssd_t AllocVirtualSSD(int isolationLevel,
      int tputLevel, size_t capacity);
```

Instead of asking tenants to specify absolute numbers, FlashBlox enables them to create different sizes and types of vSSDs with different levels of isolation and throughput (see Table 1). These parameters are compatible with the performance and economic cost levels such as the ones [3, 6] advertised in DaaS systems to ease usage and management. Tenants can scale up capacity by creating multiple vSSDs of supported sizes just as it is done in DaaS systems today. A vSSD is deallocated with `void DeallocVirtualSSD(vssd_t vSSD)`.

Channels, dies and planes are used for providing different levels of performance isolation. This brings significant performance benefits to multi-tenant scenarios (details discussed in § 4.2) because they can be operated independently from each other.

Higher levels of isolation have larger resource allocation granularities as channels are larger than dies. Therefore, channel-granular allocations can have higher internal fragmentation compared to die-granular allocations. However, this is less of a concern for FlashBlox's design for several reasons. First, a typical data center server can house eight NVMe SSDs [46]. Therefore, the maximum number of channel-isolated and die-isolated vSSDs we can support is 128 and 1024 respectively using 16-channel SSDs. Further, SSDs with 32 channels are on the horizon which can double the number of vSSDs which should be sufficient based on our conversations with the service providers at Microsoft.

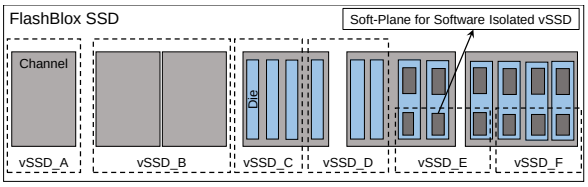


Figure 4: A FlashBlox SSD: vSSD_A and B use one and two channels respectively. vSSD_C and D use three dies each. vSSD_E, and F use three soft-planes each.

Second, the differentiated storage offerings of DaaS systems [3, 6, 10] allow tenants to choose from a certain fixed number of performance and capacity classes. This allows the cloud provider to reduce complexity. In such applications, the flexibility of dynamically changing capacity and IOPS is obtained by changing the number of partitions dedicated to the application. FlashBlox's design of bulk channel/die allocations aligns well with such a model. Third, the differentiated isolation levels match with the existing cost model for cloud storage platforms, in which better services are subject to increased pricing. This is a natural fit for FlashBlox where channels are more expensive and performant than dies.

In DaaS systems, capacity is simply scaled up by creating new partitions. For instance in Amazon RDS and Azure DocumentDB, applications scale capacity by increasing the number of partitions. Each partition is offered as a fixed unit containing a certain amount of storage and IOPS (or application-relevant operations per second). We designed FlashBlox for meeting the demands of DaaS applications. Finally, hardware-isolated vSSDs can coexist with software-isolated ones. For instance, a few channels of each SSD can be used for providing traditional software-isolated SSDs whereby the cloud provider further increases the number of differentiated performance and isolation levels.

Beyond providing different levels of hardware isolation, FlashBlox has to overcome the unbalanced wear-leveling challenge to prolong the SSD lifetime. We describe the design of each vSSD type and its corresponding wear-leveling mechanism respectively as follows.

3.1 Channel Isolated Virtual SSDs

A vSSD with high isolation receives its own dedicated set of channels. For instance, the resource manager of an SSD with 16 channels can host up to 16 channel-isolated vSSDs, each containing one or more channels inaccessible to any other vSSD. Figure 4 illustrates vSSD A and B that span one and two channels respectively.

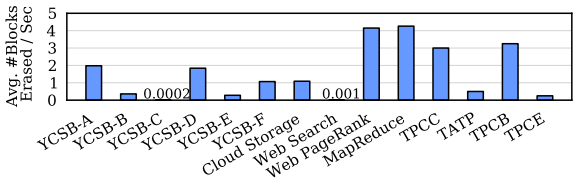


Figure 5: The average rate at which flash blocks are erased for various workloads, including NoSQL, SQL and batch processing workloads.

3.1.1 Channel Allocation

The throughput level and target capacity determine the number of channels allocated to a channel isolated vSSD. To this end, FlashBlox allows the data center/DaaS administrator to implement the `size_t tputToChannel(int tputLevel)` function that maps between throughput levels and required number of channels. The number of channels allocated to the vSSD is, therefore, the maximum of `tputToChannel(tputLevel)` and $\lceil$`capacity / capacityPerChannel`$\rceil$.

Within a vSSD, the system stripes data across its allocated channels similar to traditional SSDs. This maximizes the peak throughput by operating on the channels in parallel. Thus, the size of the super-block of vSSD_A in Figure 4 is half that of vSSD_B. Pages within the super-block are also striped across the channels similar to existing physical SSDs.

The hardware-level isolation present between the channels by virtue of hardware parallelism allows the read, program and erase operations on one vSSD to largely be unaffected by the operations on other vSSDs. Such an isolation enables latency sensitive applications to significantly reduce their tail latencies.

Compared to an SSD that stripes data from all applications across all channels, a vSSD (over fewer channels) delivers a portion of the SSD's all-channel bandwidth. Customers of DaaS systems are typically given and charged for a fixed bandwidth/IOPS level, and software rate-limiters actively keep their consumption in check. Thus, there is no loss of opportunity for not providing the peak-bandwidth capabilities for every vSSD.

3.1.2 Unbalanced Wear-Leveling Challenge

A significant side effect of channel isolation is the risk of uneven aging of the channels in the SSD as different vSSDs may be written at different rates. Figure 5 shows how various storage workloads erase blocks at different rates indicating that channels pinned naively to vSSDs will age at different rates if left unchecked.

Such uneven aging may exhaust a channel's life long before other channels fail. Premature death of even a single channel would render significant capacity losses ($> 6\%$ in our SSD). Furthermore, premature death of a single channel leads to an opportunity loss of never being able to create a vSSD that spans all the 16 channels for the rest of the server's lifetime. Such an imbalance in capability of servers represents lost opportunity costs given that other components in the server such as CPU, network and memory do not prematurely lose capabilities. Furthermore, unpredictable changes in capabilities also complicate the job of load-balancers which typically assume uniform or predictably non-uniform (by design) capabilities. Therefore, it is necessary to ensure that all the channels are aging at the same rate.

3.1.3 Inter-Channel Wear-Leveling

To ensure uniform aging of all channels, FlashBlox uses a simple yet effective wear-leveling scheme:

Periodically, the channel that has incurred the maximum wear thus far is swapped with the channel that has the minimum rate of wear.

A channel's wear rate is the average rate at which it erased blocks since the last time the channel was swapped. This prevents the most-aged channels from seeing high wear rates, thus intuitively extending their lifetime to match that of the other channels in the system.

Our experiments with workload traces from Microsoft's data center workloads show that such an approach works well in practice. We can ensure near-perfect wear-leveling with this mechanism and a swap frequency of once every few weeks. Furthermore, the impact on tail-latency remains low during the 15-minute migration period (see § 4.3.1). We analytically derive the minimum necessary frequency in § 3.1.4 and present the design of the migration mechanism in § 3.1.5.

3.1.4 Swap Frequency Analysis

Let σ_i denote the wear (total erase count of all the blocks till date) of the i^{th} channel. $\xi = \sigma_{max}/\sigma_{avg}$ denotes the wear imbalance[1] which must not exceed $1 + \delta$; where $\sigma_{max} = Max(\sigma_1, ..., \sigma_N)$, $\sigma_{avg} = Avg(\sigma_1, ..., \sigma_N)$, N is the total number of channels, and δ measures the imbalance.

When the device is new, it is obviously not possible to ensure that $\xi \leq 1 + \delta$ without aggressively swapping channels. On the other hand, it must be brought within bounds early in the lifetime of the server (L = 150–250 weeks typical) such that all the channels are available for as much of the server's lifetime as possible.

SSDs are provisioned with a target erase workload and we analyze for the same – let's say M erases per week. We mathematically study the wear-imbalance vs. frequency of migration (f) tradeoff and show that manage-

[1]The ratio of maximum to average is an effective way to quantify imbalance [45]. This is especially true in our case, as the lifetime of the new SSD is determined by the maximum wear of a single channel, whereas the lifetime of ideal wear-leveling is determined by the average wear of all the channels. The ratio of maximum to average thus represents the loss of lifetime due to imperfect wear leveling.

able values of f can provide acceptable wear imbalance where ξ comes below $1+\delta$ after αL weeks, where α is between 0 and 1.

The worst-case workload for FlashBlox is when all the writes go to a single channel.[2] The assumption that a single channel's bandwidth can handle the entire provisioned bandwidth is valid for modern SSDs: most SSDs are provisioned with 3,000-10,000 erases per cell to last 150–250 weeks. The provisioned erase rate for a 1TB SSD is therefore M=21–116 MBPS, which is lower than a channel's erase bandwidth (typically 64–128MBPS).

For an SSD with N channels, the wear imbalance of ideal wear-leveling is $\xi = 1$, while the worst case workload for FlashBlox gives a $\xi = N$: $\sigma_{max}/\sigma_{avg} = M*time/(M*time/N) = N$ before any swaps. A simple swap strategy of cycling the write workload through the N channels (write workload spends $1/f$ weeks per channel) is analyzed. Let's assume that after K rounds of cycling through all the channels, $KN/f \geq \alpha L$ holds true – that is αL weeks have elapsed and ξ has become less than $1+\delta$ and continues to remain there. At that very instant ξ equals 1. Therefore, $\sigma_{max} = MK$ and $\sigma_{avg} = MK$, then after the next swap, $\sigma_{max} = MK + M$ and $\sigma_{avg} = MK + M/N$. In order to guarantee that the imbalance is always limited, we need:

$$\xi = \sigma_{max}/\sigma_{avg} = (MK+M)/(MK+M/N) \leq (1+\delta)$$

This implies $K \geq (N-1-\delta)/(N\delta)$ which is upper bounded by $1/\delta$. Therefore, to guarantee that $\xi \leq (1+\delta)$, it is enough to swap $NK = N/\delta$ times in the first αL weeks. This implies that, over a period of five years, if α were 0.9 then a swap must be performed once every 12 days ($= 1/f$) for a $\delta = 0.1$ ($N = 16$). Table 2 shows how the frequency of swaps increases with the number of channels (shown as decreasing time period). This also implies that $\frac{2}{16}^{th}$ of the SSD is erased to perform the swap once every 12 days, which is negligible compared to the 3,000–10,000 cycles that typical SSDs have. However, for realistic workloads that do not have such a skewed write pattern with a constant bandwidth, swaps must be adaptively performed according to workload patterns (see Table 5) to reduce the number of swaps needed while maintaining balanced wear.

3.1.5 Adaptive Migration Mechanism

We assume a constant write rate of M for analysis purposes, but in reality writes are bursty. High write rates must trigger frequent swaps while swapping may not be needed as often during periods of low write rates. To achieve this, FlashBlox maintains a counter per channel to represent the amount of space erased (MB) in each channel since the last swap. Once one of the counters goes beyond a certain threshold γ, a swap is performed, and the counters are cleared. γ is set to the amount of space erased if the channel experiences the worst-case write workload between two swaps (i.e., M/f).

[2]This worst-case is from a non-adversarial point of view. An adversary could change the vSSD write bandwidth at runtime such that no swapping strategy can keep up. But most data center workloads are not adversarial and have predictable write patterns. We leave it to a security watch dog to kill over-active workloads that are not on a whitelist.

Table 2: The frequency of swaps increases as the number of channels increase to maintain balanced wear – swap periods shown below for the SSD to last five years.

Number of Channels	8	16	32	64
Swap Period (days)	26	12	6	3

The rationale behind this mechanism is that the channels must always be positioned in a manner to be able to catch up in the worst-case. FlashBlox then swaps the channels with σ_{max} and λ_{min}, where λ_i denotes the wear rate of the i^{th} channel and $\lambda_{min} = Min(\lambda_1, ..., \lambda_N)$.

FlashBlox uses an atomic block-swap mechanism to gradually migrate the candidate channels to their new locations without any application involvement. The mechanism uses an erase-block granular mapping table (described in § 3.4) for each vSSD that is maintained in a consistent and durable manner.

The migration happens in four steps. First, FlashBlox stops and queues all of the in-flight read, program and erase operations associated with the two erase-blocks being swapped. Second, the erase-blocks are read into a memory buffer. Third, the erase-blocks are written to their new locations. Fourth, the stopped operations are then dequeued. Note that only the IO operations for the swapping erase blocks in the vSSD are queued and delayed. The IO requests for other blocks are still issued with higher priority to mitigate the migration overhead.

The migrations affect the throughput and latency of the vSSDs involved. However, they are rare (happen less than once in a month for real workloads) and take only 15 minutes to finish (see § 4.3.1).

As a future optimization, we wish to modify the DaaS system to perform the read operations on other replicas to further reduce the impact. For systems that perform reads only on the primary replica, the migration can be staged within a replica-set such that the replica that is currently undergoing a vSSD migration is, if possible, first converted into a backup. Such an optimization would reduce the impact of migrations on the reads in applications that are replicated.

3.2 Die-Isolated Virtual SSDs

For applications which can tolerate some interference (i.e., *medium* isolation) such as the non-premium cloud database offerings (e.g., Amazon's small database instance [3] and Azure's standard database service [62]), FlashBlox provides die-level isolation. The num-

ber of dies in such a vSSD is the maximum of `tputToDie(tputLevel)` (defined by the administrator) and $\lceil$`capacity / capacityPerDie`$\rceil$. Their super-blocks and pages stripe across all the dies within the vSSD to maximize throughput. Figure 4 illustrates vSSD_C, and D containing three dies each (vSSD_D has dies from different channels). These vSSDs, however, have weaker isolation guarantees since dies within a channel must share a bus.

The wear-leveling mechanism has to track wear at the die level as medium-level isolated vSSDs are pinned to dies. Thus, we split the wear-leveling mechanism in FlashBlox into two sub-mechanisms: channel level and die level. The job of the channel-level wear-balancing mechanism is to ensure that all the channels are aging at roughly the same rate (see § 3.1). The job of the die-level wear-balancing mechanism is to ensure that all the dies within a channel are aging roughly at the same rate.

As shown in § 3.1.4, an N channel SSD has to swap at least N/δ times to guarantee $\xi \leq (1+\delta)$ within a target time period. This analysis also holds true for dies within a channel. For the SSDs today, in which each channel has 4 dies, FlashBlox has to swap dies in each channel 40 times in the worst case during the course of the SSD's lifetime or once every month.

As an optimization, we leverage the channel-level migration to opportunistically achieve the goal of die-level wear-leveling, based on the fact that dies have to migrate along with the channel-level migration. During each channel-level migration, the dies within the migrated channels with the largest wear is swapped with the dies that have the lowest write rate in the respective channels. Experiments with real workloads show that such a simple optimization can effectively provide satisfactory lifetime for SSDs (see § 4.3.2).

3.3 Software Isolated Virtual SSDs

For applications that have even lower requirements of isolation like Azure's basic database service [62], the natural possibility of using plane level isolation arises. However, planes within a die do not provide the same level of flexibility as channels and dies with respect to operating them independently from each other: Each die allows operating either one plane at a time or all the planes at the same address offset. Therefore, we use an approach where all the planes are operated simultaneously but their bandwidth/IOPS is split using software.

Each die is split into four regions of equal size called soft-planes by default, the size of each soft-plane is 4 GB in FlashBlox (other configurations are also supported). Planes are physical constructs inside a die. Soft-planes however are simply obtained by striping data across all the planes in the die. Further, each soft-plane in a die obtains an equal share of the total number of blocks within a die. They also receive *fair* share of bandwidth of the die. The rationale behind this is to make it easier for data center/PaaS administrator to map the throughput levels required from tenants to quantified numbers of soft-planes.

vSSDs created using soft-planes are otherwise indistinguishable from traditional virtual SSDs where software rate limiters are used to split an SSD across multiple tenants. Similar to such settings, we use the state-of-the-art token bucket rate-limiter [13, 67, 78] which has been widely used for Linux containers and Docker [12] to improve isolation and utilization at the same time. Our actual implementation is similar to the weighted fair-share mechanisms in prior work [64]. In addition, separate queues are used for enqueuing requests to each die.

The number of soft-planes used for creating these vSSDs is determined similarly to the previous cases: as the maximum of `tputToSoftPlane(tputLevel)` and $\lceil$`capacity / capacityPerSoftPlane`$\rceil$. Figure 4 illustrates vSSDs E and F that contain three soft-planes each. The super-block used by such vSSDs is simply striped across all the soft-planes used by the vSSD. We use such vSSDs as the baseline for our comparison of channel and die isolated vSSDs.

The software mechanism allows the flash blocks of each vSSD to be trimmed in isolation, which can reduce the GC interference. However, it cannot address the situation where erase operations on one soft-planes occasionally block all the operations of other soft-planes on the shared die. Thus, such vSSDs can only provide software isolation which is lower than die-level isolation.

Besides these isolated vSSDs, FlashBlox also supports an **unisolated vSSD** model which is similar to software isolated vSSD, but a fair sharing mechanism is not used to isolate such vSSDs from each other. To guarantee the fairness between vSSDs in today's cloud platforms, software isolated vSSDs are enabled by default in FlashBlox to meet *low* isolation requirements.

For both software isolated and unisolated vSSDs, their wear-balancing strategy is kept the same rather than swapping soft-planes. The rationale for this is that isolation between soft-planes of a die is provided using software and not by pinning vSSDs to physical flash planes. Therefore, a more traditional wear-leveling mechanism of simply rotating blocks between soft-planes of a die is sufficient to ensure that the soft-planes within a die are all aging roughly at the same rate. We describe this mechanism in more detail in the next section.

3.4 Intra Channel/Die Wear-Leveling

The goals of intra die wear-leveling are to ensure that the blocks in each die are aging at the same rate while enabling applications to access data efficiently by avoiding the pitfalls of multiple indirection layers and redundant functionalities across these layers [27, 35, 54, 77].

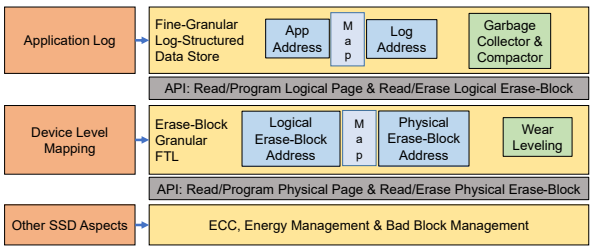


Figure 6: In FlashBlox, applications manage a fine-granular log-structured data store and align compaction units to erase-blocks. A device level indirection layer is used to ensure all erase-blocks are aging at the same rate.

With both die-level (see § 3.3) and intra-die wear leveling mechanisms, FlashBlox inevitably achieves the goal of intra-channel wear-leveling as well: all the dies in each channel and all the blocks in each die age uniformly.

The intra-die wear-leveling in FlashBlox is illustrated in Figure 6. We leverage flash-friendly application or file system logic to perform GC and compaction, and simplify the device level mapping. We also leverage the drive's capabilities to manage bad blocks without having to burden applications with error correction, detection and scrubbing. We base our design for intra-die wear-levelling on existing open SSDs [27, 38, 49]. We describe our specific design for completeness.

3.4.1 Application/Filesystem Level Log

The API of FlashBlox, as shown in Table 3, is designed with log-structured systems in mind. The only restriction it imposes is that the application or the file system perform the log-compaction at a granularity that is the same as the underlying vSSD's erase granularity.

When a FlashBlox based log-structured application or a filesystem needs to clean an erase-block that contains a live object (say O) then, (1) It first allocates a new block via `AllocBlock`; (2) It reads object O via `ReadData`; (3) It writes object O in to the new block via `WriteData`; (4) It modifies its index to reflect the new location of object O; (5) It frees the old block via `FreeBlock`. Note that the newly allocated block still has many pages that can be written to, which can be used as the head of the log for writing live data from other cleaned blocks or for writing new data.

FlashBlox does not assume that the log-structured system frees all the allocated erase-blocks at the same rate. Such a restriction would force the system to implement a sequential log cleaner/compactor as opposed to techniques that give weight to other aspects such as garbage collection efficiency [37, 38]. Instead, FlashBlox ensures uniform wear of erase-blocks at a lower level.

3.4.2 Device-Level Mapping

The job of the lower layers is to ensure: (1) that all erase-blocks within a die are being erased at roughly the same rate and (2) that erase-blocks that have imminent failures have their data migrated to a different erase-block and the erase-block be permanently hidden from applications; both without requiring application changes.

Table 3: FlashBlox API

`vssd_t AllocVirtualSSD( int isolationLevel, int tputLevel, size_t capacity )` /*Creates a virtual SSD*/
`void DeallocVirtualSSD( vssd_t vSSD )` /*Deletes a virtual SSD*/
`size_t GetBlockSize( vssd_t vSSD )` /*Erase-block size of vSSD: depends on the number of channels/dies used*/
`int ReadData( vssd_t vSSD, void* buf, off_t offset, size_t size )` /*Reads data; contiguous data is read faster with die-parallel reads*/
`block_t AllocBlock( vssd_t vSSD )` /*Allocates a new block; it can be written to only once and sequentially*/
`int WriteData( vssd_t vSSD, block_t logical_block_id, void* buf, size_t size )` /*Writes page aligned data to a previously allocated (erased) block; contiguous data is written faster with die-parallel writes*/
`void FreeBlock( vssd_t vSSD, block_t logical_block_id )` /*Frees a previously allocated block*/

With device-level mapping, the physical erase-blocks' addresses are not exposed to applications – only logical erase-block addresses are exposed to upper software. That is, the device exposes each die as an individual SSD that uses a block-granular FTL, while application-level log in FlashBlox ensures that upper layers only issue block-level allocation and deallocation calls. The indirection overhead is small since they are maintained at erase-block granularity (requiring 8MB per TB of SSD).

Unlike tradtional SSDs, in FlashBlox, tenants cannot share pre-erased blocks. While this has the advantage that the tenants control their own write-amplification factors, write and GC performance, the disadvantage is that bursty writes within a tenant cannot opportunistically use pre-erased blocks from the entire device.

In FlashBlox, each die is given its own private block-granular mapping table, and a IO queue with a depth of 256 by default (it is configurable) to support basic storage operations and software rate limiter for software isolated vSSDs. The out-of-band metadata (16 bytes used) in each block is used to note the logical address of the physical erase-block; this enables atomic, consistent and durable operations. The logical address is a unique and global 8 bytes number consisting of die ID and block ID within the die. The other 8 bytes of the metadata are used for a 2 bytes erase counter and a 6 byte erase timestamp. FlashBlox caches the mapping table and all other out-of-band metadata in the host memory. Upon system crashes, FlashBlox leverages the reverse mappings and timestamps in out-of-metadata to recover the mapping table [24, 80]. More specifically, we use the implementation from our prior work [27].

The device-level mapping layer can be implemented either in the host or in the drive's firmware itself [49] if the device's controller has under-utilized resources; we

implement it in the host. Error detection, correction and masking, and other low-level flash management systems remain unmodified in FlashBlox.

Both the application/filesystem level log and the device-level mapping need to over provision, but for different reasons. The log needs to over-provision for the sake of garbage collection efficiency. Here, we rely on the existing logic within log-structured, flash-aware applications and file systems to perform their own over-provisioning appropriate for their workloads. The device-level mapping needs its own over-provisioning for the sake of retiring error-prone erase-blocks. In our implementation, we set this to 1% based on the error rate analysis from our prior work [29].

3.5 Implementation Details

Prototype SSD. We implement FlashBlox using a CNEX SSD [18] which is an open-channel SSD [44] containing 1 TB Toshiba A19 flash memory and an open controller that allows physical resource access from the host. It has 16 channels, each channel has 4 dies, each die has 4 planes, each plane has 1024 blocks, each block has 256 pages with 16 KB page size. This hardware provides basic I/O control commands to issue read, write and erase operations against flash memory. We use a modified version of the CNEX firmware/driver stack that allows us to independently queue requests to each die. FlashBlox is implemented using the C programming language in 11,219 lines of code (LoC) layered on top of the CNEX stack.

Prototype Application and Filesystem. We were able to modify LevelDB key-value store and the Shore-MT database engine to use FlashBlox using only 38 and 22 LoC modifications respectively. These modifications are needed to use the APIs in Table 3. Additionally, we implemented a user-space log-structured file system (vLFS) with 1,809 LoC (only 26 LoC are from FlashBlox API) based on FUSE for applications which cannot be modified.

Resource Allocation. For each call to create a vSSD, the resource manager performs a linear search of all the available channels, dies and soft-planes to satisfy the requirements. A set of free lists of them are maintained for this purpose. During deallocation, the resource manager takes the freed channels, dies and soft-planes and coalesces them when possible. For instance, if all the four dies of a channel become free then the resource manager coalesces the dies into a channel and adds the channel to the free channel set. In the future, we wish to explore admission control and other resource allocation strategies.

4 Evaluation

Our evaluation demonstrates that: (1) FlashBlox has overheads (WAF and CPU) comparable to state-of-the-art FTLs (§ 4.1); (2) Different levels of hardware isolation can be achieved by utilizing flash parallelism, and they perform better than software isolation (§ 4.2.1); (3) Hardware isolation enables latency-sensitive applications such as web search to effectively share an SSD with bandwidth-intensive workloads like MapReduce jobs (§ 4.2.2); (4) The impact of wear-leveling migrations on data center applications' performance is low (§ 4.3.1) and (5) FlashBlox's wear-leveling is near to ideal (§ 4.3.2).

Table 4: Application workloads used for evaluation.

	Workload	I/O Pattern
Key-Value Store	YCSB-A	50% read, 50% update
	YCSB-B	95% read, 5% update
	YCSB-C	100% read
	YCSB-D	95% read, 5% insert
	YCSB-E	95% scan, 5% insert
	YCSB-F	50% read, 50% read-modify-write
Data Center	Cloud Storage	26.2% read, 73.8% write
	Web Search	83.0% read, 17.0% write
	Web PageRank	17.7% read, 82.2% write
	MapReduce	52.9% read, 47.1% write
Databases	TPC-C	mix (65.5% read, 34.5% write)
	TATP	mix (81.2% read, 18.8% write)
	TPC-B	account update (100% write)
	TPC-E	mix (90.7% read, 9.3% write)

Experimental Setup: We used FIO benchmarks [20] and 14 different workloads for the evaluation (Table 4): six NoSQL workloads from the Yahoo Cloud Serving Benchmarks (YCSB) [19], four database workloads: TPC-C [70], TATP [65], TPC-B [69] and TPC-E [71], and four storage workload traces collected from Microsoft's data centers.

YCSB is a framework for evaluating the performance of NoSQL stores. All of the six core workloads consisting of A, B, C, D, E and F are used for the evaluation. LevelDB [39] is modified to run using the vSSDs from FlashBlox with various isolation levels. The open-source SQL database Shore-MT [55] is modified to work over the vSSDs of FlashBlox. The table size of the four database workloads TPC-C, TATP, TPC-B and TPC-E range from 9 - 25 GB each. A wear-imbalance factor limit of 1.1 is used for all our experiments to capture realistic swapping frequencies. The number of dies, channels and planes used for each experiment is specified separately for each experiment.

Storage intensive and latency sensitive applications from Microsoft's data centers are instrumented to collect traces for cloud storage, web search, PageRank and MapReduce workloads. These applications are the first-party customers of Microsoft's storage IaaS system.

4.1 Microbenchmarks

We benchmark two vSSDs that each run an FIO benchmark to evaluate FlashBlox's WAF. Compared to the unmodified CNEX SSD's page-level FTL, FlashBlox deliv-

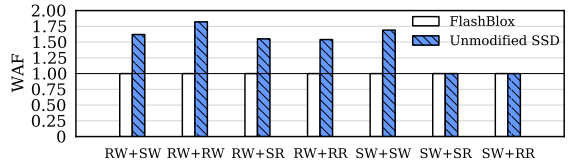


Figure 7: WAF comparison between FlashBlox and a traditional SSD. RW/RR: random write/read; SW/SR: sequential write/read.

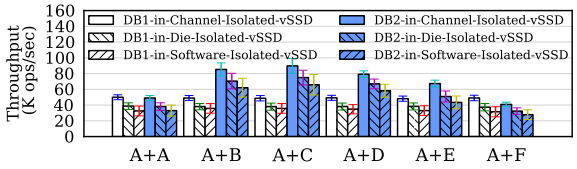
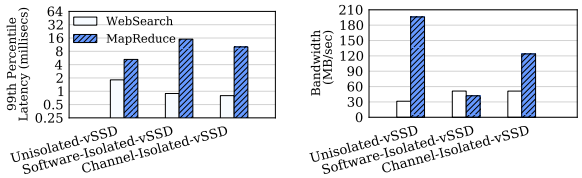


Figure 8: The throughput of LevelDB+YCSB workloads running at various levels of storage isolation.



(a) 99th Percentile Latency (b) Bandwidth

Figure 11: The performance of colocated Web Search and MapReduce workload traces.

ers lower WAFs as shown in Figure 7 because of the fact that FlashBlox's vSSDs never share the same physical flash blocks for storing their pages. As shown by previous work [34], this reduces WAF because of absence of false sharing of blocks at the application level. The different types of vSSDs of FlashBlox have similar WAFs because they all use separate blocks, yet they provide different throughput and tail latency levels (shown in Section 4.2) because of higher levels of isolation.

In addition, FlashBlox has up to 6% higher total system CPU usage compared to the unmodified CNEX SSD when running FIO. Despite merging the file system's index with that of the FTL's by using FlashBlox's APIs which reduces latency as shown by existing open-channel work [38, 49], the additional CPU overhead is due to the device-level mapping layer that is accesed in every critical path. As a future optimization for the production SSD, we plan to transparently move the device-level mapping layer into the SSD.

4.2 Isolation Level vs. Tail Latency

In this section, we demonstrate that higher levels of isolation provide lower tail latencies. Multiple instances of application workloads are run on individual vSSDs of different kinds. In each workload, the number of client threads executing transactions is gradually increased until the throughput tapers off. The maximum throughput achieved for the lowest number of threads is then recorded. The average and tail latencies of transactions are recorded for the same number of threads.

4.2.1 Hardware Isolation vs. Software Isolation

In this experiment, the channel and die isolated vSSDs are evaluated against the software isolated vSSDs (with weighted fair sharing of storage bandwidth enabled). We begin with a scenario of two LevelDB instances. They run on two vSSDs in three different configurations, each using a different isolation level: high, medium, and low; they contain one channel, four dies and sixteen soft-planes respectively to ensure that the resources are consistent across experiments. The two instances run a YCSB workload each. The choice of YCSB is made for this experiment to show how removing IO interference can improve the throughput and reduce latency for IO-bottlenecked applications.

Each LevelDB instance is first populated with 32 GB of data and each key-value pair is 1 KB. The YCSB client threads perform 50 million CRUD (i.e., create, read, update and delete) operations against each LevelDB instance. We pick the size of the database and number of operations such that GC is always triggered. YCSB C is read-only, thus we report results for read operations only.

The total number of dies in each setting is the same. In the channel isolation case, two vSSDs are allocated from two different channels. In the die isolation case, both vSSDs share the channels, but are isolated at the die level within the channel. In the software isolation case, both vSSDs are striped across all the dies in two channels.

Figure 8 shows that on average, channel isolated vSSD provides 1.3x better throughput compared to die isolated vSSD and 1.6x compared to the software isolated vSSD. Similarly, higher levels of isolation lead to lower average latencies as shown in Figure 9 (a) and Figure 9 (b). This is because higher levels of isolation suffer from less interference between read and write operations from other instances. Die isolated vSSDs have to share the bus with each other, thus, their performance is worse than channel isolated vSSDs, which are fully isolated in hardware. Software isolated vSSDs share the same dies with each other, suffering from higher interference.

Tail latency improvements are much more significant. As shown in Figure 9 (c) and Figure 9 (d), channel isolated vSSDs provide up to 1.7x lower tail latency compared to die isolated vSSDs and up to 2.6x lower tail latency compared to vSSDs that stripe data across all the dies akin to software isolated vSSDs whose operations are not fully isolated from each other.

A similar experiment with four LevelDB instances is also performed. Tail latency results are shown in Figure 10 where channel isolated vSSDs provide up to 3.1x lower tail latency compared the software isolated vSSDs.

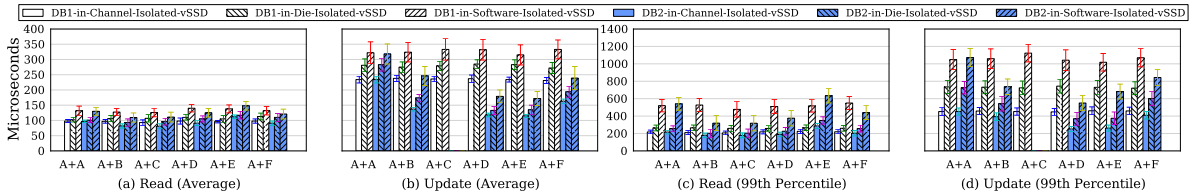
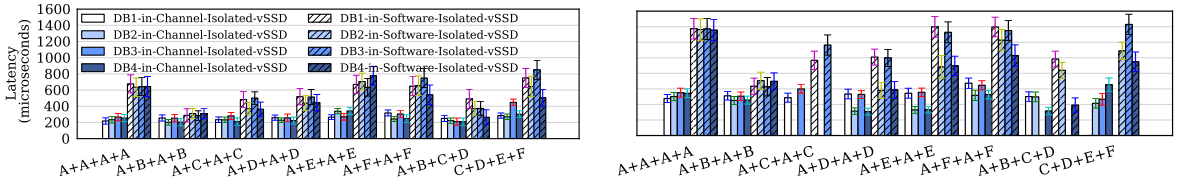


Figure 9: The average and 99th percentile latencies of LevelDB+YCSB workloads running at various levels of storage isolation. Compared to die and software isolated vSSDs, channel isolated vSSD reduces the average latency by 1.2x and 1.4x respectively, and decreases the 99th percentile latency by 1.2 - 1.7x and 1.9 - 2.6x respectively. Note that the update latencies are not applicable for workload C which is read-only.



(a) Read Latency (99th Percentile)

(b) Update Latency (99th Percentile)

Figure 10: The 99th percentile latency of running four LevelDB instances with various levels of storage isolation. The channel isolated vSSD reduces the 99th percentile latency by 1.3 - 2.7x and 1.5 - 3.1x for read and update operation respectively, compared to software isolated vSSD.

4.2.2 Latency vs. Bandwidth Sensitive Tenants

We now evaluate how hardware isolation provides benefits for instances that share the same physical SSD when one is latency sensitive while others are not (for resource efficiency [42]). Channel, software isolated and unisolated vSSDs are used in this experiment. The total number of dies is the same in all three settings and is eight. The workloads from a large cloud provider are used for performing this experiment. Web search is the instance that requires lower tail latencies while MapReduce jobs are not particularly latency sensitive.

Results shown in Figure 11 demonstrate three trends: First, channel isolated vSSDs provide the best compromise between throughput and tail latency: tail latency of the web search workloads decreases by over 2x for a 36% reduction of bandwidth of the MapReduce job when compared to an unisolated vSSD. The fall in throughput of MapReduce is expected because it only has half of the channels of the unisolated case where its large sequential IOs end up consuming the bandwidth unfairly due to the lack of any isolation techniques.

Second, software isolated vSSDs for web search and MapReduce can reduce the tail latency of web search to the same level as the channel-isolated case, but the bandwidth of the MapReduce job decreases by more than 4x when compared to the unisolated vSSD. This is also expected because the work that an SSD can perform is a convex combination of IOPS and bandwidth. Web search takes a significant number of small IOPS when sharing bandwidth fairly with MapReduce and this in-turn reduces the total bandwidth available for MapReduce.

4.3 Wear-Leveling Efficacy and Overhead

Wear-leveling in FlashBlox is supported in two different layers. One layer ensures that all the dies in the system are aging at the same rate overall with channel migrations, while the other layer ensures that blocks within a given die are aging at the same rate overall. Its overhead and efficacy are evaluated in this section.

4.3.1 Migration Overhead

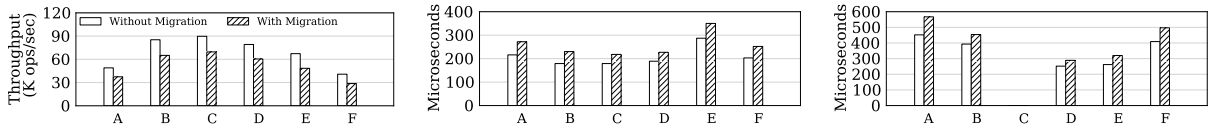
We first evaluate the overhead of the migration mechanism. We migrate one channel and measure the change in throughput and 99th percentile latency on a variety of YCSB workloads that are running on the channel.

The throughput of LevelDB running on that channel drops by no more than 33.8% while the tail latencies of reads and updates increase by up to 22.1% (Figure 12). For simplicity, we show results for migrating 1 GB of the 64GB channel. We use a single thread and the data moves at a rate of 78.9MBPS. Moving all of the 64 GB of data would take close to 15 minutes.

The impact of migration on web search and MapReduce workloads is shown in Figure 13. During migration, the bandwidth of the MapReduce job decreases by 36.7%, the tail latencies of reads and writes of the web search increase by 34.2%. These performance slowdowns bring channel-isolation numbers on par with the software isolation. This implies that a 36.7% drop for 15 minutes when amortized over our recommended swap rate represents a 0.04% overall drop.

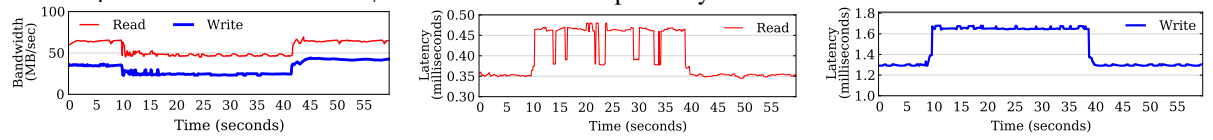
4.3.2 Migration Frequency Analysis

To evaluate the wear-leveling efficacy, we built a simulator and used it to understand how the device ages for var-



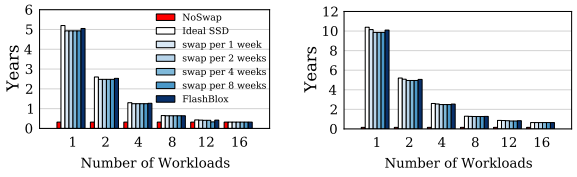
(a) Throughput (b) Read (99th Percentile) (c) Update (99th Percentile)

Figure 12: The impact of a channel migration on workloads: LevelDB's throughput falls by 33.8%, its 99th percentile read and update latencies increase by 22.1% and 18.7% respectively.



(a) Bandwidth of MapReduce (b) Read Latency of Web Search (c) Write Latency of Web Search

Figure 13: The overhead of migrating 1GB of data as MapReduce and web search are running on the channels involved: MapReduce's bandwidth falls by up to 36.7% while web search's latency increases by up to 34.2%.



(a) Channel Killer (b) Die Killer

Figure 14: SSD lifetime of running adversarial write workloads that stress a single channel or a die.

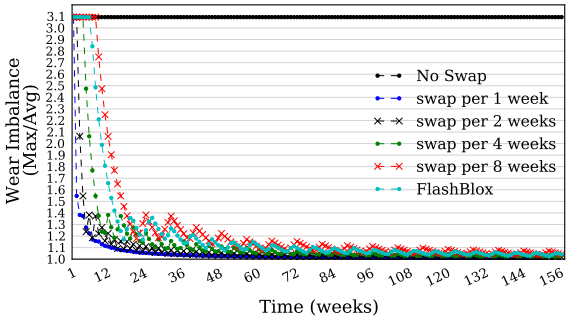


Figure 15: Wear imbalance of FlashBlox with different wear-leveling schemes. The ideal wear imbalance is 1.0.

Table 5: Monte Carlo simulation (10K runs) of SSD lifetime with randomly sampled workloads on the channels.

#vSSD	NoSwap Lifetime (Years)		Ideal vs. FlashBlox Lifetime (Years)		Wear Imbalance	Swap Once in Days (Avg)
	99th	50th	99th	50th		
4	1.2	1.6	6.2/6.1	13.8/13.5	1.02	94
8	1.2	1.3	3.7/3.6	6.7/6.6	1.02	22
16	1.2	1.2	2.1/2.1	3.4/3.3	1.01	19

ious workloads. For workload traces that are not from a log-structured application, we first execute the workload on the log-structured file system vLFS built using FlashBlox and trace FlashBlox API calls. We measure the block consumption rate of these traces to evaluate the efficacy of wear-leveling. For the CNEX SSD, $\gamma = M/f =$ 24 TB (discussed in § 3.1.5). The supported number of program erase (PE) cycles is 10 K in our drive. Our absolute lifetimes scale linearly for other SSDs and factor improvements remain the same regardless of the number of supported PE cycles.

Worst-case workloads. To evaluate the possible worst cases for SSDs, we run the most write-intensive workloads against a few channels (channel killer) and dies (die killer). We gradually increase the number of such workloads to stress the SSD. Each workload is pinned to exactly one channel or one die while keeping other channels or dies for read-only operations.

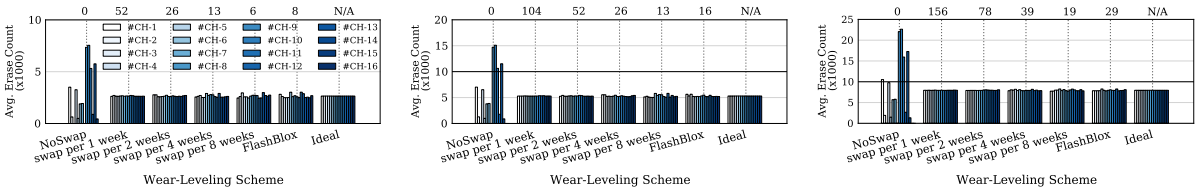
Figure 14 shows the SSD's lifetime for a variety of wear-leveling schemes. Without wear-leveling (NoSwap in Figure 14), the SSD dies after less than 4 months, while FlashBlox can always guarantee 95% of the ideal lifetime within migration frequency of once per $\leq$ 4 weeks for both channel and die killer workloads. The adaptive wear-leveling scheme in FlashBlox automatically migrates a channel by adjusting to write-rates.

Mixed workloads. In real-world scenarios, a mix of various kinds of workloads would run on the same SSD. We use all the 14 workloads (Table 4) simultaneously in the experiment, and measure FlashBlox's wear leveling. Fourteen channel isolated vSSDs are created for running these workloads and migrations. Figure 5 shows how the erase rates of these applications vary.

For the scheme without any migrations, the wear imbalance is 3.1, and the SSD dies after 1.2 years. Also, results show that blocks are more or less evenly aged for a migration frequency as high as once in four weeks, as shown in Figure 15. This indicates that for realistic scenarios, where write traffic is more evenly matched, significantly fewer swaps could be tolerated.

Figure 16 shows the absolute erase counts of the channels (including the erases needed for migrations and GC). Compared to the ideal wear-leveling, the absolute erase counts are almost the same with the migration frequency of a week.

To further evaluate FlashBlox's wear-leveling efficacy, we run a Monte Carlo simulation (10K runs) of the SSD lifetime. We create various number of vSSDs and assign



(a) After 1 year (b) After 2 years (c) After 3 years

Figure 16: Erase counts over three years for workloads in Table 4. The erase count per block in each channel of FlashBlox is close to that of the ideal SSD. The numbers on the top shows the cumulative migration count.

them uniformly at random to one of the fourteen workloads. The SSD is then simulated to end-of-life.

We report the 99th and 50th percentile lifetime of ideal SSD, SSD without swapping (NoSwap) and FlashBlox in Table 5. For the case of running 16 instances, 99% of the ideal SSDs last 2.1 years, and half of them can work for 3.4 years. With adaptive wear-leveling scheme, FlashBlox's lifetime is close to ideal and its wear imbalance is close to the ideal case. In real world, where not all applications are adversarial (channel/die-killer workloads), the swap frequency automatically increases.

5 Related Work

Open Architecture SSDs. Recent research has proposed exposing flash parallelism directly to the host [38, 49, 61, 76]. This is immensely helpful for applications where each unit of flash parallelism receives more or less similar write workloads. However, this is often not the case in multi-tenant cloud platform where workloads with a variety of write-rates co-exist on the same SSD. FlashBlox takes a holistic approach to solve this problem, it not only provides hardware isolation but also ensures all the units of parallelism are aging uniformly.

SSD-level Optimizations. Recent work has successfully improved SSDs' performance by enhancing how FTLs leverage the flash parallelism [17, 32]. We extend this line of research for performance isolation for applications in a multi-tenant setting. FlashBlox uses dedicated channels and dies for each application to improve isolation and balances inter-application wear using a new strategy, while existing FTL optimizations are relevant for intra-application wear-leveling.

SSD Interface. Programmable and flexible SSD interfaces have been proposed to improve the communication between applications and SSD hardware [15, 50, 53]. SR-IOV [63] is a hardware bus standard that helps virtual machines bypass the host to safely share hardware to reduce CPU overhead. These techniques are complimentary to FlashBlox which helps applications use dedicated flash regions. Multi-streamed SSDs [33] addresses a similar problem with a stream tag, isolating each stream to dedicated flash blocks but sharing all channels, dies and planes to achieve maximum per-stream throughput. OPS isolation [34] has been proposed to dedicate flash blocks to each virtual machine sharing an SSD. They reduce fragmentation and GC overheads. FlashBlox builds upon this work and extends the isolation to channels and dies without compromising on wear-leveling.

Storage Isolation. Recent research has demonstrated that making software aware of the underlying hardware constraints can improve isolation. Shared SSD performance [56, 57] can be improved by observing the convex-dependency between IOPS and bandwidth, and also by predicting future workloads [64]. In contrast, FlashBlox identifies the relation between flash isolation and wear when using hardware isolation, and makes software schedulers aware of it. It solves this problem by helping software perform coarse time granular wear-levelling across channels and dies.

6 Conclusions and Future Work

In this paper, we propose leveraging channel and die-level parallelism present in SSDs to provide isolation for latency sensitive applications sharing an SSD. Furthermore, FlashBlox provides near-ideal lifetime despite the fact that individual applications write at different rates to their respective channels and dies. FlashBlox achieves this by migrating applications between channels and dies at coarse time granularities. Our experiments show that FlashBlox can improve throughput by 1.6x and reduce tail latency by up to 3.1x. We also show that migrations are rare for real world workloads and do not adversely impact applications' performance. In the future, we wish to take FlashBlox in two directions. First, we would like to investigate how to integrate with the virtual hard drive stack such that virtual machines can leverage FlashBlox without modification. Second, we would like to understand how FlashBlox should be integrated with multi-resource data center schedulers to help applications obtain predictable end-to-end performance.

Acknowledgments

We would like to thank our shepherd Ming Zhao as well as the anonymous reviewers. This work was supported in part by the Center for Future Architectures Research (C-FAR), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. We would also like to thank the great folks over at CNEX for supporting our research by providing early access to their open SSDs.

References

[1] N. Agarwal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. USENIX ATC*, Boston, MA, June 2008.

[2] Amazon Relational Database Service. https://aws.amazon.com/rds/.

[3] Amazon Relational Database Service Pricing. https://aws.amazon.com/rds/pricing/.

[4] Amazon's SSD Backed EBS. https://aws.amazon.com/blogs/aws/new-ssd-backed-elastic-block-storage/.

[5] Azure DocumentDB. https://azure.microsoft.com/en-us/services/documentdb/.

[6] Azure DocumentDB Pricing. https://azure.microsoft.com/en-us/pricing/details/documentdb/.

[7] Azure Premium Storage. https://azure.microsoft.com/en-us/documentation/articles/storage-premium-storage/.

[8] Azure Service Fabric. https://azure.microsoft.com/en-us/services/service-fabric/.

[9] Azure SQL Database. https://azure.microsoft.com/en-us/services/sql-database/.

[10] Azure SQL Database Pricing. https://azure.microsoft.com/en-us/pricing/details/sql-database/.

[11] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-aware Contention Management on Multicore Systems. In *Proc. USENIX ATC'11*, Berkeley, CA, June 2011.

[12] Block IO Bandwidth (Blkio) in Docker. https://docs.docker.com/engine/reference/run/#block-io-bandwidth-blkio-constraint.

[13] Block IO Controller. https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt.

[14] Y. Bu, H. Lee, and J. Madhavan. Comparing SSD-placement Strategies to scale a Database-in-the-Cloud. In *Proc. SoCC'13*, Santa Clara, CA, Oct. 2013.

[15] A. M. Caulfield, T. I. Mollov, L. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proc. ACM ASPLOS'12*, London, United Kingdom, Mar. 2012.

[16] CGROUPS. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[17] F. Chen, R. Lee, and X. Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing. In *Proc. HPCA'11*, San Antonio, Texas, Feb. 2011.

[18] CNEX Labs. http://www.cnexlabs.com/index.php.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. SoCC'12*, Indianapolis, Indiana, June 2010.

[20] FIO Benchmarks. https://linux.die.net/man/1/fio.

[21] Fusion-io ioDrive. https://www.sandisk.com/business/datacenter/products/flash-devices/pcie-flash/sx350.

[22] Google Cloud Platform: Local SSDs. https://cloud.google.com/compute/docs/disks/local-ssd.

[23] Google Cloud SQL. https://cloud.google.com/sql/.

[24] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proc. ACM ASPLOS*, Washington, DC, Mar. 2009.

[25] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proc. FAST'16*, Santa Clara, CA, Feb. 2016.

[26] J. He, D. Nguyen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Reducing File System Tail Latencies with Chopper. In *Proc. FAST'15*, Santa Clara, CA, Feb. 2015.

[27] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan. Unified Address Translation for Memory-Mapped SSD with FlashMap. In *Proc. ISCA'15*, Portland, OR, June 2015.

[28] Intel Inc. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *White Paper*, 2015.

[29] Iyswarya Narayanan and Di Wang and Myeongjae Jeon and Bikash Sharma and Laura Caulfield and Anand Sivasubramaniam and Ben Cutler and Jie Liu and Badriddine Khessib and Kushagra Vaid. SSD Failures in Datacenters: What? When? and Why? In *Proc. ACM SYSTOR'16*, Haifa, Israel, June 2016.

[30] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *Proc. NSDI'13*, Berkeley, CA, Apr. 2013.

[31] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A File System for Virtualized Flash Storage. *ACM Trans. on Storage*, 6(3):14:1–14:25, 2010.

[32] M. Jung and M. K. Ellis H. Wilson III. Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks. In *Proc. ISCA'12*, Portland, OR, June 2012.

[33] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. The Multi-Streamed Solid-State Drive. In *Proc. HotStorage'14*, Philadelphia, PA, June 2014.

[34] J. Kim, D. Lee, and S. H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *Proc. FAST'15*, Santa Clara, CA, Feb. 2015.

[35] W.-H. Kim, B. Nam, D. Park, and Y. Won. Resolving Journaling of Journal Anomaly in Android IO: Multi-version B-tree with Lazy Split. In *FAST'14*, Santa Clara, CA, Feb. 2014.

[36] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux implementation of a log-structured file system. *SIGOPS OSR*, 40(3), 2006.

[37] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *Proc. FAST'15*, Santa Clara, CA, Feb. 2015.

[38] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind. Application-Managed Flash. In *Proc. FAST'16*, Santa Clara, CA, Feb. 2016.

[39] LevelDB. `https://github.com/google/leveldb`.

[40] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proc. EuroSys'14*, Amsterdam, Netherlands, Apr. 2014.

[41] N. Li, H. Jiang, D. Feng, and Z. Shi. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proc. EuroSys'16*, London, United Kingdom, Apr. 2016.

[42] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proc. ISCA'15*, Portland, OR, June 2015.

[43] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proc. MICRO'11*, Porto Alegre, Brazil, Dec. 2011.

[44] Matias Bjorling and Javier Gonzalez and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proc. USENIX FAST'17*, Santa Clara, CA, Feb. 2016.

[45] H. Menon and L. Kale. A Distributed Dynamic Load Balancer for Iterative Applications. In *Proc. SC'13*, Denver, Colorado, Nov. 2013.

[46] Microsoft's Open Source Cloud Hardware. `https://azure.microsoft.com/en-us/blog/microsoft-reimagines-open-source-cloud-hardware/`.

[47] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In *Proc. MICRO'11*, Porto Alegre, Brazil, Dec. 2011.

[48] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds. In *Proc. EuroSys'12*, Paris, France, Apr. 2010.

[49] J. Ouyang, S. Lin, S. Jiang, Y. Wang, W. Qi, J. Cong, and Y. Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *Proc. ACM ASPLOS*, Salt Lake City, UT, Mar. 2014.

[50] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *Proc. HPCA'11*, San Antonio, Texas, Feb. 2014.

[51] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. on Computer Systems*, 10(1):26–52, Feb. 1992.

[52] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proc. ISCA'11*, San Jose, CA, June 2011.

[53] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A User-Programmable SSD. In *Proc. OSDI'14*, Broomfield, CO, Oct. 2014.

[54] K. Shen, S. Park, and M. Zhu. Journaling of journal is (almost) free. In *Proc. FAST'14*, Berkeley, CA, 2014.

[55] Shore-MT. `https://sites.google.com/site/shoremt/`.

[56] D. Shue and M. J. Freedman. From Application Requests to Virtual IOPs: Provisioned Key-Value Storage with Libra. In *Proc. EuroSys'14*, Amsterdam, Netherlands, Apr. 2014.

[57] D. Shue, M. J. Freedman, and A. Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proc. OSDI'12*, Hollywood, CA, Oct. 2012.

[58] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundama, M. G. Guajardo, A. Wawrzyniak, S. Boshra, R. Ferreira, M. Nassar, M. Koltachev, J. Huang, S. Sengupta, J. Levandoski, and D. Lomet. Schema-agnostic indexing with azure documentdb. In *Proc. VLDB'15*, Kohala Coast, Hawaii, Sept. 2015.

[59] A. Singh, M. Korupolu, and D. Mohapatra. Server-Storage Virtualization: Integration and Load Balancing in Data Centers. In *Proc. SC'08*, Austin, Texas, Nov. 2008.

[60] D. Skourtis, D. Achlioptas, N. Watkins, C. Maltzahn, and S. Brandt. Flash on rails: consistent flash performance through redundancy. In *Proc. USENIX ATC'14*, Philadelphia, PA, June 2014.

[61] X. Song, J. Yang, and H. Chen. Architecting Flash-based Solid-State Drive for High-performance I/O Virtualization. *IEEE Computer Architecture Letters*, 13:61–64, 2014.

[62] SQL Database Options and Performance: Understand What's Available in Each Service Tier. `https://azure.microsoft.com/en-us/documentation/articles/sql-database-service-tiers/#understanding-dtus`.

[63] SR-IOV for SSDs. http://www.snia.org/sites/default/files/Accelerating%20Storage%20Perf%20in%20Virt%20Servers.pdf.

[64] Sungyong Ahn and Kwanghyun La and Jihong Kim. Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems. In *Proc. USENIX HotStorage'16*, Denver, CO, June 2016.

[65] TATP Benchmark. http://tatpbenchmark.sourceforge.net/.

[66] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A Software-Defined Storage Architecture. In *Proc. SOSP'13*, Farmington, PA, Nov. 2013.

[67] Throtting IO with Linux. https://fritshoogland.wordpress.com/2012/12/15/throttling-io-with-linux.

[68] Token Bucket Algorithm. https://en.wikipedia.org/wiki/token_bucket.

[69] TPCB Benchmark. http://www.tpc.org/tpcb/.

[70] TPCC Benchmark. http://www.tpc.org/tpcc/.

[71] TPCE Benchmark. http://www.tpc.org/tpce/.

[72] Traffic Control HOWTO. http://linux-ip.net/articles/Traffic-Control-HOWTO/.

[73] B. Trushkowsky, P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *Proc. FAST'11*, Santa Clara, CA, Feb. 2016.

[74] H. Wang and P. Varman. Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation. In *Proc. FAST'14*, Santa Clara, CA, Feb. 2014.

[75] J. Wang and Y. Hu. WOLF: A Novel reordering write buffer to boost the performance of log-structured file systems. In *Proc. FAST'02*, Monterey, CA, Jan. 2002.

[76] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An Effective Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *Proc. EuroSys'14*, Amsterdam, the Netherlands, Apr. 2014.

[77] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman. Don't stack your Log on my Log. In *Proc. INFLOW'14*, Broomfield, CO, Oct. 2014.

[78] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. AI-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Split-Level I/O Scheduling. In *Proc. SOSP'15*, Monterey, CA, Oct. 2015.

[79] N. Zhang, J. Tatemura, J. M. Patel, and H. Hacigumus. Re-evaluating Designs for Multi-Tenant OLTP Workloads on SSD-based I/O Subsystems. In *Proc. SIGMOD'14*, Snowbird, UT, June 2014.

[80] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proc. 10th USENIX FAST*, San Jose, CA, Feb. 2012.

DIDACache: A Deep Integration of Device and Application for Flash based Key-value Caching

Zhaoyan Shen[†] *Feng Chen*[‡] *Yichen Jia*[‡] *Zili Shao*[†]
[†]*Department of Computing* [‡]*Computer Science & Engineering*
Hong Kong Polytechnic University *Louisiana State University*

Abstract

In recent years, flash-based key-value cache systems have raised high interest in industry, such as Facebook's McDipper and Twitter's Fatcache. These cache systems typically use commercial SSDs to store and manage key-value cache data in flash. Such a practice, though simple, is inefficient due to the huge *semantic gap* between the key-value cache manager and the underlying flash devices. In this paper, we advocate to reconsider the cache system design and directly open device-level details of the underlying flash storage for key-value caching. This co-design approach bridges the semantic gap and well connects the two layers together, which allows us to leverage both the domain knowledge of key-value caches and the unique device properties. In this way, we can maximize the efficiency of key-value caching on flash devices while minimizing its weakness. We implemented a prototype, called DIDACache, based on the Open-Channel SSD platform. Our experiments on real hardware show that we can significantly increase the throughput by 35.5%, reduce the latency by 23.6%, and remove unnecessary erase operations by 28%.

1 Introduction

High-speed key-value caches, such as Memcached [31] and Redis [37], are the "first line of defense" in today's low-latency Internet services. By caching the working set in memory, key-value cache systems can effectively remove time-consuming queries to the back-end data store (e.g., MySQL or LevelDB). Though effective, the in-memory key-value caches heavily rely on large amount of expensive and power-hungry DRAM for high cache hit ratio [18]. As the workload size rapidly grows, an increasing concern with such memory-based cache systems is their cost and scalability [2]. Recently, a more cost-efficient alternative, *flash-based key-value caching*, has raised high interest in the industry [12, 45].

NAND flash memory provides a much larger capacity and lower cost than DRAM, which enables a low Total Cost of Ownership (TCO) for a large-scale deployment of key-value caches. Facebook, for example, deploys a Memcached-compatible key-value cache system based on flash memory, called McDipper [12]. It is reported that McDipper allows Facebook to reduce the number of deployed servers by as much as 90% while still delivering more than 90% "get responses" with sub-millisecond latencies [23]. Twitter also has a similar key-value cache system, called Fatcache [45].

Typically, these flash-based key-value cache systems directly use commercial flash SSDs and adopt a Memcached-like scheme to manage key-value cache data in flash. For example, key-values are organized into slabs of different size classes, and an in-memory hash table is used to maintain the key-to-value mapping. Such a design is simple and allows a quick deployment. However, it disregards an important fact – the key-value cache systems and the underlying flash devices both have very *unique properties*. Simply treating flash SSDs as a faster storage and the key-value cache as a regular application not only fails to exploit various optimization opportunities but also raises several critical concerns, namely *redundant mapping*, *double garbage collection*, and *over-overprovisioning*. All these issues cause enormous inefficiencies in practice, which motivated us to reconsider the software/hardware structure of the current flash-based key-value cache systems.

In this paper, we will discuss the above-mentioned three key issues (Section 3) caused by the huge *semantic gap* between the key-value caches and the underlying flash devices, and further present a cohesive cross-layer design to fundamentally address these issues. Through our studies, we advocate to open the underlying details of flash SSDs for key-value cache systems. Such a co-design effort not only enables us to remove the unnecessary intermediate layers between the cache manager and the storage devices, but also allows us to leverage the precious domain knowledge of key-value cache systems, such as the unique access patterns and mapping structures, to effectively exploit the great potential of flash storage while avoiding its weakness.

By reconsidering the division between software and hardware, a variety of new optimization opportunities can be explored: (1) A single, unified mapping structure can directly map the "keys" to physical flash pages storing the "values", which completely removes the redundant mapping table and saves a large amount of on-device memory; (2) An integrated Garbage Collection (GC) procedure, which is directly driven by the cache system, can optimize the decision of when and how to recycle *semantically invalid* storage space at a fine granularity, which removes the high overhead caused by

the unnecessary and uncoordinated GCs at both layers; (3) An on-line scheme can determine an optimal size of Over-Provisioning Space (OPS) and dynamically adapt to the workload characteristics, which will maximize the usable flash space and greatly increase the cost efficiency of using expensive flash devices.

We have implemented a fully functional prototype, called *DIDACache*, based on a PCI-E Open-Channel SSD hardware to demonstrate the effectiveness of this new design scheme. A thin intermediate library layer, `libssd`, is created to provide a programming interface to facilitate applications to access low-level device information and directly operate the underlying flash device. Using the library layer, we developed a flash-aware key-value cache system based on Twitter's Fatcache [45]. Our experiments show that this approach can increase the throughput by 35.5%, reduce the latency by 23.6%, and remove erase operations by 28%.

The rest of paper is organized as follows. Section 2 and Section 3 give background and motivation. Section 4 describes the design and implementation. Experimental results are presented in Section 5. Section 6 gives the related work. The final section concludes this paper.

2 Background

This section briefly introduces three key technologies, flash memory, SSDs, and the current flash-based key-value cache systems.

• **Flash Memory**. NAND flash memory is a type of EEPROM device. A flash memory chip consists of multiple *planes*, each of which consists of thousands of *blocks* (a.k.a. erase blocks). A block is further divided into hundreds of *pages*. Flash memory supports three main operations, namely `read`, `write`, and `erase`. Reads and writes are normally performed in units of pages. A read is typically fast (e.g., 50µs), while a write is relatively slow (e.g., 600µs). A constraint is that pages in a block must be written sequentially, and pages cannot be overwritten in place, meaning that once a page is programmed (written), it cannot be written again until the entire block is erased. An erase is typically slow (e.g., 5ms) and must be done in block granularity.

• **Flash SSDs**. A typical flash SSD includes a host interface logic, an SSD controller, a dedicated buffer, and flash memory controllers connecting to flash memory chips via multiple channels. A *Flash Translation Layer* (FTL) is implemented in firmware to manage flash memory. An FTL has three major roles: (1) *Logical block mapping*. An in-memory mapping table is maintained in the on-device buffer to map logical block addresses to physical flash pages dynamically. (2) *Garbage collection*. Due to the erase-before-write constraint, upon a write, the corresponding logical page is written to a new location, and the FTL simply marks the old page invalid. A GC procedure recycles obsolete pages later, which is similar to a Log-Structured File System [38]. (3) *Wear Leveling*. Since flash cells could wear out after a certain number of Program/Erase cycles, the FTL shuffles read-intensive blocks with write-intensive blocks to even out writes over flash memory. A previous work [13] provides a detailed survey of FTL algorithms.

• **Flash-based key-value caches**. In-memory key-value cache systems, such as Memcached, adopt a slab-based allocation scheme. Due to its efficiency, flash-based key-value cache systems, such as Fatcache, inherit a similar structure. Here we use Fatcache as an example; based on open documents [12], McDipper has a similar design. In Fatcache, the SSD space is first segmented into *slabs*. Each slab is further divided into an array of *slots* (a.k.a. chunks) of equal size. Each slot stores a "value" item. Slabs are logically organized into different *slab classes* based on the slot sizes. An incoming key-value item is stored into a class whose slot size is the best fit of its size. For quick access, a *hash mapping table* is maintained in memory to map the keys to the slabs containing the values. Querying a key-value pair (`GET`) is accomplished by searching the in-memory hash table and loading the corresponding slab block from flash into memory. Updating a key-value pair (`SET`) is realized by writing the updated value into a new location and updating the key-to-slab mapping in the hash table. Deleting a key-value pair (`DELETE`) simply removes the mapping from the hash table. The deleted or obsolete value items are left for GC to reclaim later.

Despite the structural similarity to Memcached, flash-based key-value cache systems have several distinctions from their memory-based counterparts. First, the I/O granularity is much larger. For example, Memcached can update the value items individually. In contrast, Fatcache has to maintain an in-memory slab to buffer small items in memory first and then flush to storage in bulk later, which causes a unique "large-I/O-only" pattern on the underlying flash SSDs. Second, unlike Memcached, which is byte addressable, flash-based key-value caches cannot update key-value items in place. In Fatcache, all key-value updates are written to new locations. Thus, a GC procedure is needed to clean/erase slab blocks. Third, the management granularity in flash-based key-value caches is much coarser. For example, Memcached maintains an object-level LRU list, while Fatcache uses a simple slab-level FIFO policy to evict the oldest slab when free space is needed.

3 Motivation

As shown in Figure 1, in a flash-based key-value cache, the *key-value cache manager* and the *flash SSD* run at the application and device levels, respectively. Both layers have complex internals, and the interaction between the

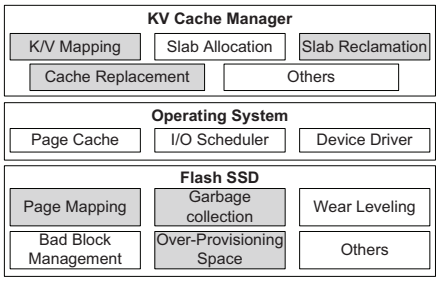


Figure 1: Architecture of flash-based key-value cache.

two raises three critical issues, which have motivated the work presented in this paper.

• **Problem 1: Redundant mapping**. Modern flash SSDs implement a complex FTL in firmware. Although a variety of mapping schemes, such as DFTL [17], exist, high-end SSDs often still adopt fine-grained *page-level mapping* for performance reasons. As a result, for a 1TB SSD with a 4KB page size, a page-level mapping table could be as large as 1GB. Integrating such a large amount of DRAM on device not only raises production cost but also reliability concerns [17, 53, 54]. In the meantime, at the application level, the key-value cache system also manages another mapping structure, an in-memory hash table, which translates the keys to the corresponding slab blocks. The two mapping structures exist at two levels simultaneously, which unnecessarily doubles the memory consumption.

A fundamental problem is that the page-level mapping is designed for general-purpose file systems, rather than key-value caching. In a typical key-value cache, the slab block size is rather large (in Megabytes), which is typically 100-1,000x larger than the flash page size. This means that the fine-grained page-level mapping scheme is an *expensive over-kill*. Moreover, a large mapping table also incurs other overheads, such as the need for a large capacitor or battery, increased design complexity, reliability risks, etc. If we could directly map the hashed keys to the physical flash pages, we can completely remove this redundant and highly inefficient mapping for lower cost, simpler design, and improved performance.

• **Problem 2: Double garbage collection**. GC is the main performance bottleneck of flash SSDs [3, 7]. In flash memory, the smallest read/write unit is a page (e.g., 4KB). A page cannot be overwritten in place until the entire erase block (e.g., 256 pages) is erased. Thus, upon a write, the FTL marks the obsolete page "invalid" and writes the data to another physical location. At a later time, a GC procedure is scheduled to recycle the invalidated space for maintaining a pool of clean erase blocks. Since valid pages in the to-be-cleaned erase block must be first copied out, cleaning an erase block often takes hundreds of milliseconds to complete. A key-value cache system has a similar GC procedure to recycle the slab space occupied by obsolete key-value pairs.

Running at different levels (application vs. device), these two GC processes not only are redundant but also could interfere with one another. For example, from the FTL's perspective, it is unaware of the semantic meaning of page content. Even if no key-value pair is valid (i.e., no key maps to any value item), the entire page is still considered as "valid" at the device level. During the FTL-level GC, this page has to be moved unnecessarily. Moreover, since the FTL-level GC has to assume all valid pages contain useful content, it cannot selectively recycle or even aggressively invalidate certain pages that contain semantically "unimportant" (e.g., LRU) key-value pairs. For example, even if a page contains only one valid key-value pair, the entire page still has to be considered valid and cannot be erased, although it is clearly of relatively low value. Note that `TRIM` command [43] cannot address this issue as well. If we merge the two-level GCs and control the GC process based on semantic knowledge of the key-value caches, we could completely remove all the above-mentioned inefficient operations and create new optimization opportunities.

• **Problem 3: Over-overprovisioning**. In order to minimize the performance impact of GC on foreground I/Os, the FTL typically reserves a portion of flash memory, called Over-Provisioned Space (OPS), to maintain a pool of clean blocks ready for use. High-end SSDs often reserve 20-30% or even larger amount of flash space as OPS. From the user's perspective, the OPS space is nothing but an expensive unusable space. We should note that the factory setting for OPS is mostly based on a conservative estimation for worst-case scenarios, where the SSD needs to handle extremely intensive write traffic. In key-value cache systems, in contrast, the workloads are often read-intensive [5]. Reserving such a large portion of flash space is a significant waste of expensive resource. In the meantime, key-value cache systems possess rich knowledge about the I/O patterns and have the capability of accurately estimating the incoming write intensity. Based on such estimation, a suitable amount of OPS could be determined during runtime for maximizing the usable flash space for effective caching. Considering the importance of cache size for cache hit ratio, such a 20-30% extra space could significantly improve system performance. If we could leverage the domain knowledge of the key-value cache systems to determine the OPS management at the device level, we would be able to maximize the usable flash space for caching and greatly improve the overall cost efficiency as well as system performance.

In essence, all the above-mentioned issues stem from a fundamental problem in the current I/O stack design: the key-value cache manager runs at the application level and views the storage abstraction as a sequence of sectors; the flash memory manager (i.e., the FTL)

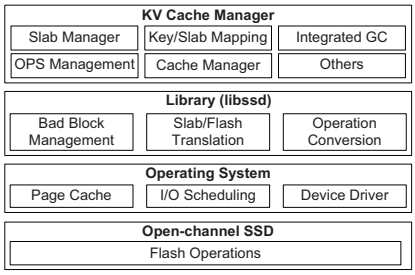


Figure 2: The architecture overview of DIDACache.

runs at the device firmware layer and views incoming requests simply as a sequence of individual I/Os. This abstraction, unfortunately, creates a huge *semantic gap* between the key-value cache and the underlying flash storage. Since the only interface connecting the two layers is a strictly defined block-based interface, no semantic knowledge about the data could be passed over. This enforces the key-value cache manager and the flash memory manager to work individually and prevents any collaborative optimizations. This motivates us to study how to bridge this semantic gap and build a highly optimized flash-based key-value cache system.

4 Design

As an unconventional hardware/software architecture (see Figure 2), our key-value cache system is highly optimized for flash and eliminates all unnecessary intermediate layers. Its structure includes three layers.

- *An enhanced flash-aware key-value cache manager*, which is highly optimized for flash memory storage, runs at the application level, and directly drives the flash management;
- *A thin intermediate library layer*, which provides a slab-based abstraction of low-level flash memory space and an API interface for directly and easily operating flash devices (e.g., `read`, `write`, `erase`);
- *A specialized flash memory SSD hardware*, which exposes the physical details of flash memory medium and opens low-level *direct* access to the flash memory medium through the `ioctl` interface.

With such a holistic design, we strive to completely bypass multiple intermediate layers in the conventional structure, such as file system, generic block I/O, scheduler, and the FTL layer in SSD. Ultimately, we desire to let the application-level key-value cache manager leverage its domain knowledge and directly drive the underlying flash devices to operate only necessary functions while leaving out unnecessary ones. In this section, we will discuss each of the three layers.

4.1 Application Level: Key-value Cache

Our key-value cache manager has four major components: (1) a *slab management module*, which manages memory and flash space in slabs; (2) a *unified direct mapping module*, which records the mapping of key-value items to their physical locations; (3) an *integrated GC module*, which reclaims flash space occupied by obsolete key-values; and (4) an *OPS management module*, which dynamically adjusts the OPS size.

4.1.1 Slab Management

Similar to Memcached, our key-value cache system adopts a slab-based space management scheme – the flash space is divided into equal-sized *slabs*; each slab is divided into an array of *slots* of equal size; each slot stores a key-value item; slabs are logically organized into different *slab classes* according to the slot size.

Despite these similarities to in-memory key-value caches, caching key-value pairs in flash has to deal with several unique properties of flash memory, such as the "out-of-place update" constraint. By directly controlling flash hardware, our slab management can be specifically optimized to handle these issues as follows.

• **Mapping slabs to blocks**: Our key-value cache directly maps (logical) slabs to physical flash blocks. We divide flash space into equal-sized slabs, and each slab is statically mapped to one or several flash blocks. There are two possible mapping schemes: (1) *Per-channel mapping*, which maps a slab to a sequence of contiguous physical flash blocks in one channel, and (2) *Cross-channel mapping*, which maps a slab across multiple channels in a round-robin way. Both have pros and cons. The former is simple and allows to directly infer the logical-to-physical mapping, while the latter could yield a better bandwidth through channel-level parallelism.

We choose the simpler per-channel mapping for two reasons. First, key-value cache systems typically have sufficient slab-level parallelism. Second, this allows us to directly translate "slabs" into "blocks" at the library layer with minimal calculation. In fact, in our prototype, we directly map a flash slab to a physical flash block, since the block size (8MB) is appropriate as one slab. For flash devices with a smaller block size, we can group multiple contiguous blocks in one channel into one slab.

• **Slab buffer**: Unlike DRAM memory, flash does not support random in-place overwrite. As so, a key-value item cannot be directly updated in its original place in flash. For a `SET` operation, the key-value item has to be stored in a new location in flash (appended like a log), and the obsolete item will be recycled later. To enhance performance, we maintain an *in-memory slab* as a buffer for each slab class. Upon receiving a `SET` operation, the key-value pair is first stored in the corresponding in-memory slab and completion is immediately returned. When the in-memory slab is full, it is flushed into an *in-flash slab* for persistent storage.

The slab buffer brings two benefits. First, the in-memory slab works as a write-back buffer. It not only speeds up accesses but also makes incoming requests asynchronous, which greatly improves the throughput. Second, and more importantly, the in-memory slab

merges small key-value slot writes into large slab writes (in units of flash blocks), which completely removes the unwanted small flash writes. Our experiments show that a small slab buffer is sufficient for performance.

• **Channel selection and slab allocation**: For load balance considerations, when an in-memory slab is full, we first select the channel with the lowest load. The load of each channel is estimated by counting three key flash operations (`read`, `write`, and `erase`). Once a channel is selected, a free slab is allocated. For each channel, we maintain a *Free Slab Queue* and a *Full Slab Queue* to manage clean slabs and used slabs separately. The slabs in a free slab queue are sorted in the order of their erase counts, and we always select the slab with the lowest erase count first for wear-leveling purposes. The slabs in a full slab queue are sorted in the Least Recently Used (LRU) order. When running out of free slabs, the GC procedure is triggered to produce clean slabs, which we will discuss in more details later.

With the above optimizations, a fundamental effect is, all I/Os seen at the device level are shaped into large-size slab writes, which completely removes small page writes as well as the need for generic GC at the FTL level.

4.1.2 Unified Direct Mapping

In order to address the double mapping problem, a key change is to remove all the intermediate mappings, and directly map the SHA-1 hash of the key to the corresponding physical location (i.e., the slab ID and the offset) in the in-memory hash table.

Figure 3 shows the structure of the in-memory hash table. Each hash table entry includes three fields: `<md, sid, offset>`. For a given key, `md` is the SHA-1 digest, `sid` is the ID of the slab that stores the key-value item, and `offset` is the slot number of the key-value item within the slab. Upon a request, we first calculate the hash value of the "key" to locate the bucket in the hash table, and then use the SHA-1 digest (`md`) to retrieve the hash table entry, in which we can find the slab (`sid`) containing the key-value pair and the corresponding slot (`offset`). The found slab could be in memory (i.e., in the slab buffer) or in flash. In the former case, the value is returned in a memory access; in the latter case, the item is read from the corresponding flash page(s).

4.1.3 Garbage Collection

Garbage collection is a must-have in key-value cache systems, since operations (e.g., `SET` and `DELETE`) can create obsolete value items in slabs, which need to be recycled at a later time. When the system runs out of free flash slabs, we need to reclaim their space in flash.

With the semantic knowledge about the slabs, we can perform a fine-grained GC in one single procedure, running at the application level only. There are two possible strategies for identifying a victim slab: (1) *Space-based eviction*, which selects the slab containing the largest number of obsolete values, and (2) *Locality-based eviction*, which selects the coldest slab for cleaning based on the LRU order. Both policies are used depending on the runtime system condition.

• **Space-based eviction**: As a greedy approach, this scheme aims to maximize the freed flash space for each eviction. To this end, we first select a channel with the lowest load to limit the search scope, and then we search its *Full Slab Queue* to identify the slab that contains the least amount of valid data. As the slot sizes of different slab classes are different, we use the number of valid key-value items times their size to calculate the valid data ratio for a given flash slab. Once the slab is identified, we scan the slots of the slab, copy all valid slots into the current in-memory slab, update the hash table mapping accordingly, then erase the slab and place the cleaned slab back in the *Free Slab Queue* of the channel.

• **Locality-based eviction**: This policy adopts an aggressive measure to achieve fast reclamation of free slabs. Similar to *space-based eviction*, we first select the channel with the lowest load. We then select the LRU slab as the victim slab to minimize the impact to hit ratio. This can be done efficiently as the full flash slabs are maintained in their LRU order for each channel. A scheme, called *quick clean*, is then applied by simply dropping the entire victim slab, including all valid slots. It is safe to remove valid slots, since our application is a key-value cache (rather than a key-value store) – all clients are already required to write key-values to the back-end data store first, so it is safe to aggressively drop any key-value pairs in the cache without any data loss.

Comparing these two approaches, *space-based eviction* needs to copy still-valid items in the victim slab, so it takes more time to recycle a slab but retains the hit ratio. In contrast, *locality-based eviction* allows to quickly clean a slab without moving data, but it aggressively erases valid key-value items, which may reduce the cache hit ratio. To reach a balance between the hit ratio and GC overhead, we apply these two policies *dynamically* during runtime – when the system is under high pressure (e.g., about to run out of free slabs), we use the fast but imprecise *locality-based eviction* to quickly release free slabs for fast response; when the system pressure is low, we use *space-based eviction* and try to retain all valid key-values in the cache for hit ratio.

To realize the above-mentioned dynamic selection policies, we set two watermarks, low (W_{low}) and high (W_{high}). We will discuss how to determine the two watermarks in the next section. The GC procedure checks the number of free flash slabs, S_{free}, in the current system periodically. If S_{free} is between the high watermark, W_{high}, and the low watermark, W_{low}, it means that the pool of free slabs is running low but under moderate pressure. So we activate the less aggressive

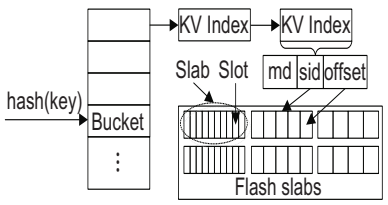


Figure 3: Unified mapping structure.

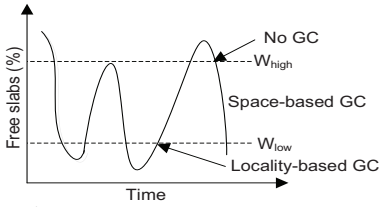


Figure 4: Low and high watermarks

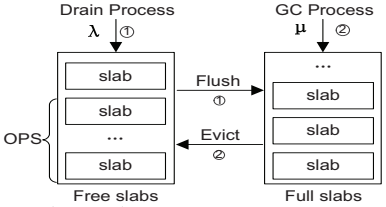


Figure 5: M/M/1 queuing model.

space-based eviction policy to clean slabs. This process repeats until the number of free slabs, S_{free}, reaches the high watermark. If S_{free} is below the low watermark, which means that the system is under high pressure, the aggressive *space-based eviction* policy kicks in and uses *quick clean* to erase the entire LRU slab and discard all items immediately. This fast-response process repeats until the number of free slabs in the system, S_{free}, is brought back to W_{low}. If the system is idle, the GC procedure switches to the *space-based eviction* policy and continues to clean slabs until reaching the high watermark. Figure 4 illustrates this process.

4.1.4 Over-Provisioning Space Management

In conventional SSDs, a large portion of flash space is reserved as OPS, which is invisible and unusable by applications. In our architecture, we can leverage the domain knowledge to dynamically adjust OPS and maximize the usable flash space for caching.

In our system, the two watermarks, W_{low} and W_{high}, drive the GC procedure. The two watermarks effectively determine the available OPS size – W_{low} is the dynamically adjusted OPS size, and W_{high} can be viewed as the upper bound of allowable OPS. We set the difference between the two watermarks, $W_{high} - W_{low}$, as a constant (15% of the flash space in our prototype). Ideally, we desire to have the number of free slabs, S_{free}, fluctuating in the window between the two watermarks.

Our goal is to keep just enough flash space for over-provisioning. However, it is challenging to appropriately position the two watermarks and make them adaptive to the workload. It is desirable to have an automatic, self-tuning scheme to dynamically determine the two watermarks based on runtime situation. In our prototype, we have designed two schemes, a *feedback-based heuristic model* and a *queuing theory based model*.

Our heuristic scheme is simple and works as follows: when the low watermark is hit, which means that the current system is under high pressure, we lift the low watermark by doubling W_{low} to quickly respond to increasing writes, and the high watermark is correspondingly updated. As a result, the system will activate the aggressive *quick clean* to produce more free slabs quickly. This also effectively reserves a large OPS space for use. When the number of free slabs reaches the high watermark, which means the current system is under light pressure, we linearly drop the watermarks. This effectively returns free slabs back to the usable cache space (i.e., reduced OPS size). In this way, the OPS space automatically adapts to the incoming traffic.

The second scheme is based on the well-known queuing theory, which builds slab allocation and reclaim processes as a M/M/1 queue. As Figure 5 shows, in this system, we maintain queues for free flash slabs and full flash slabs for each channel, separately. The slab drain process consumes free slabs, and the GC process produces free slabs. Therefore we can view the drain process as the consumer process, the GC process as the producer process, and the free slabs as resources. The drain process consumes flash slabs at a rate λ, and the GC process generates free flash slabs at a rate μ. Prior study [5] shows that in real applications, the incoming of key-value pairs can be seen as a Markov process, so the drain process is also a Markov process. For the GC process, when S_{free} is less than W_{low}, the locality-based eviction policy is adopted. The time consumed for reclaiming one slab is equal to the flash erase time plus the schedule time. The flash block erase time is a constant, and the schedule time can be viewed as a random number. Thus the locality-based GC process is also a Markov process with a service rate μ. Based on the analysis, the process can be modeled as a M/M/1 queue with arrival rate λ, service rate μ, and one server.

According to Little's law, the expected number of slabs waiting for service is $\lambda/(\mu-\lambda)$. If we reserve at least this number of free slabs before the locality-based GC process is activated, we can always eliminate the synchronous waiting time. So, for the system performance benefit, we set

$$W_{low} = \lambda/(\mu-\lambda) \tag{1}$$

In the above equation, λ is the slab consumption rate of the drain process, and μ is the slab reclaim rate of GC, which equals $1/(t_{evict}+t_{other})$, where t_{evict} is the block erase time, and t_{other} is other system time needed for GC.

In Equation 2, the arrival rate is decided by the incoming rate of key-value pairs and their average size, which are both measurable. Assuming the arrival rate of key-values is λ_{KV}, the average size is S_{KV}, and the slab size is S_{slab}, λ can be calculated as follows.

$$\lambda = \frac{\lambda_{KV} \times S_{KV}}{S_{slab}} \tag{2}$$

So, we have

$$W_{low} = \frac{\lambda_{KV} \times S_{KV} \times (t_{evict}+t_{other})}{S_{slab} - \lambda_{KV} \times S_{KV} \times (t_{evict}+t_{other})} \tag{3}$$

By using the above-mentioned equations, we can periodically update the settings of the low and high watermarks. In this way, we can adaptively tune the OPS size based on real-time workload demands.

4.1.5 Other Technical Issues

Flash memory wears out after a certain number of Program/Erase (P/E) cycles. In our prototype, for wear leveling, when allocating slabs in the drain process and reclaiming slabs in the GC process, we take the erase count of each slab into consideration and always use the block with the smallest erase count. As our channel-slab selection and slab-allocation scheme can evenly distribute the workloads across all channels, wears can be approximately distributed across channels as well. Other additional wear-leveling measures, such as dynamically shuffling cold/hot slabs, could also be included to further even out the wear distribution.

Crash recovery is also a challenge. We may simply drop the entire cache upon crashes. However, due to the excessively long warm-up time, it is preferred to retain the cached data through crashes [52]. In our system, all key-value items are stored in persistent flash but the hash table is maintained in volatile memory. There are two potential solutions to recover the hash table. One simple method is to scan all the valid key-value items in flash and rebuild the hash table, which is a time-consuming process. A more efficient solution is to periodically checkpoint the in-memory hash table into (a designated area of) the flash. Upon recovery, we only need to reload the latest hash table checkpoint into memory and then apply changes by scanning the slabs written after the checkpoint. Crash recovery is currently not implemented in our prototype.

4.2 Library Level: `libssd`

As an intermediate layer, the library, `libssd`, connects the application and device layers. Unlike Liblightnvm [15], `libssd` is highly integrated with the key-value cache system. It has three main functions: (1) *Slab-to-block mapping*, which statically maps a slab to one (or multiple contiguous) flash memory block(s) in a channel. In our prototype, it is a range of blocks in a flash LUN (logic unit number). Such a mapping can be calculated through a mathematical conversion and does not require another mapping table. (2) *Operation transformation*, which converts key slab operations, namely `read`, `write`, and `erase`, to flash memory operations. This allows the key-value cache system to operate in units of slabs, rather than flash pages/blocks. (3) *Bad block management*, which maintains a list of flash blocks that are detected as "bad" and ineligible for allocation, and hides them from the key-value cache.

4.3 Hardware Level: Open-Channel SSD

We use an Open-Channel SSD manufactured by Memblaze [30]. This hardware is similar to that used in SDF [35]. This PCIe based SSD contains 12 channels, each of which connects to two Toshiba 19nm MLC flash chips. Each chip contains two planes and has a capacity of 66GB. Unlike SDF [35], our SSD exposes several key device-level properties: first, the SSD exposes the entire flash memory space to the upper level. The SSD hardware abstracts the flash memory space in 192 LUNs, and an LUN is the smallest parallelizable unit. The LUNs are mapped to the 12 channels in a sequential manner, i.e., channel #0 contains LUNs 0-15, channel #1 contains LUNs 16-31, and so on. Therefore, we know the physical mapping of slabs on flash memory and channels. Second, unlike SDF, which presents the flash space as 44 block devices, our SSD provides direct access to raw flash memory through the `ioctl` interface. It allows us to directly operate the target flash memory pages and blocks by specifying the LUN ID and page number to compose commands added to the device command queue. Third, all FTL-level functions, such as address mapping, wear-leveling, bad block management, are bypassed. This allows us to remove the device-level redundant operations and make them completely driven by the user-level applications.

5 Evaluation

5.1 Prototype System

We have prototyped the proposed key-value cache on the Open-Channel SSD hardware platform manufactured by Memblaze [30]. Our implementation of the key-value cache manager is based on Twitter's Fatcache [45]. It includes 1,500 lines of code in the stock Fatcache and 620 lines of code in the library.

In Fatcache, when a `SET` request arrives, if running out of in-memory slabs, it selects and flushes a memory slab to flash. If there is no free flash slab, a victim flash slab is chosen to reclaim space. During this process, incoming requests have to wait synchronously. To fairly compare with a cache system with non-blocking flush and eviction, we have enhanced the stock Fatcache by adding a drain thread and a slab eviction thread. The other part remains unchanged. We have open-sourced our asynchronous version of Fatcache for public downloading [1]. In our experiments, we denote the stock Fatcache working in the synchronous mode as "Fatcache-Sync", and the enhanced one working in the asynchronous mode as "Fatcache-Async". For each platform, we configure the slab size to 8 MB, the flash block size. The memory slab buffer is set to 128MB.

For performance comparison, we also run Fatcache-Sync and Fatcache-Async on a commercial PCI-E SSD manufactured by Memblaze. The SSD is built on

the exact same hardware as our Open-Channel SSD but adopts a typical, conventional SSD architecture design. This SSD employs a page-level mapping and the page size is 16KB. Unlike the Open-Channel SSD, the commercial SSD has 2GB of DRAM on the device, which serves as a buffer for the mapping table and a write-back cache. The other typical FTL functions (e.g., wear-leveling, GC, etc.) are active on the device.

5.2 Experimental Setup

Our experiments are conducted on a workstation, which features an Intel i7-5820K 3.3GHZ processor and 16GB memory. An Open-Channel SSD introduced in Section 4.3 is used as DIDACache's underlying cache storage. Since the SSD capacity is quite large (1.5TB), it would take excessively long time to fill up the entire SSD. To complete our tests in a reasonable time frame, we only use part of the flash space, and we ensure the used space is evenly spread across all the channels and flash LUNs. For the software, we use Ubuntu 14.04 with Linux kernel 3.17.8. Our back-end database server is MySQL 5.5 with InnoDB storage engine running on a separate workstation, which features an Intel Core 2 Duo processor (3.13GHZ), 8GB memory and a 500GB hard drive. The database server and the cache server are connected in a 1Gbps local Ethernet network. Fatcache-Sync and Fatcache-Async use the same system configurations, except that they run on the commercial SSD rather than the Open-Channel SSD.

5.3 Overall Performance

Our first set of experiments simulate a production data-center environment to show the overall performance. In this experiment, we have a complete system setup with a workload generator (client simulator), a key-value cache server, and a MySQL database server in the back-end.

To generate key-value requests to the cache server, we adopt a workload model presented in prior work [6]. This model is built based on real Facebook workloads [5], and we use it to generate a key-value object data set and request sequences to exercise the cache server. The size distribution of key-value objects in the database follows a truncated Generalized Pareto distribution with location $\theta = 0$, scale $\psi = 214.4766$, and shape $k = 0.348238$. The object popularity, which determines the request sequence, follows a Normal distribution with mean μ_t and standard deviation σ, where μ_t is a function of time. We first generate 800 million key-value pairs (about 250GB data) to populate our database, and then use the object popularity model to generate 200 million requests. We have run experiments with various numbers of servers and clients with the above-mentioned workstation, but due to the space constraint, we only present the representative experimental results with 32 clients and 8 key-value cache servers.

We test the system performance by varying the cache size (in percentage of the data set size). Figure 6 shows the throughput, i.e., the number of operations per second (ops/sec). We can see that as the cache size increases from 5% to 12%, the throughput of all the three schemes improves significantly, due to the improved cache hit ratio. Comparing the three schemes, DIDACache outperforms Fatcache-Sync and Fatcache-Async substantially. With a cache size of 10% of the data set (about 25GB), DIDACache outperforms Fatcache-Sync and Fatcache-Async by 9.7% and 9.2%, respectively. The main reason is that the dynamic OPS management in DIDACache adaptively adjusts the reserved OPS size according to the request arrival rate. In contrast, Fatcache-Sync and Fatcache-Async statically reserve 25% flash space as OPS, which affects the cache hit ratio (see Figure 7). Another reason is the reduced overhead due to the application-driven GC. The effect of GC policies will be examined in Section 5.4.2.

We also note that Fatcache-Async only outperforms Fatcache-Sync marginally in this workload. This is because for this workload, Fatcache-Async adopts the same static OPS policy as Fatcache-Sync, which leads to the same cache hit ratio. Figure 7 shows the hit ratios of these three cache systems. We can see that, as the cache size increases, DIDACache's hit ratio ranges from 76.5% to 94.8%, which is much higher than that of Fatcache-Sync, ranging from 71.1% to 87.3%.

5.4 Cache Server Performance

In this section we focus on studying the performance details of the cache servers. In this experiment, we directly generate `SET`/`GET` operations to the cache server. We create objects with sizes ranging from 64 bytes to 4KB and first populate the cache server up to 25GB in total. Then we generate `SET` and `GET` requests of various key-value sizes to measure the average latency and throughput. All experiments use 8 key-value cache servers and 32 clients.

5.4.1 Random `SET`/`GET` Performance

Figure 8 shows the throughput of `SET` operations. Among the three schemes, our DIDACache achieves the highest throughput and Fatcache-Sync performs the worst. With the object size of 64 bytes, the throughput of DIDACache is 2.48×10^5 ops/sec, which is 1.3 times higher than that of Fatcache-Sync and 35.5% higher than that of Fatcache-Async. The throughput gain is mainly due to our unified slab management policy and the integrated application-driven GC policy. DIDACache also selects the least loaded channel when flushing slabs to flash. Thus, the SSD's internal parallelism can be fully utilized, and with software and hardware knowledge, the GC overhead is significantly reduced. Compared with Fatcache-Async, the relative performance gain of DIDACache is smaller and decreases as the key-value object size

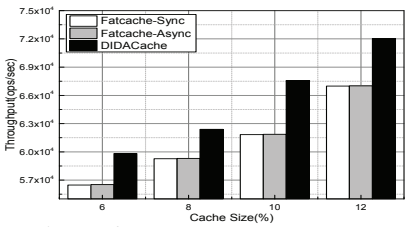


Figure 6: Throughput vs. cache size

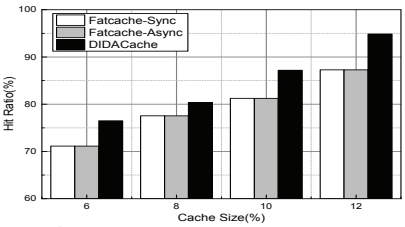


Figure 7: Hit ratio vs. cache size.

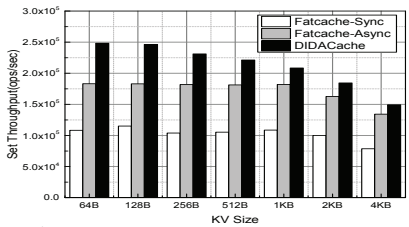


Figure 8: SET throughput vs. KV size

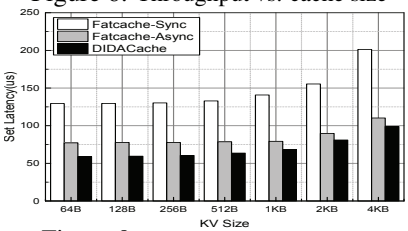


Figure 9: SET latency vs. KV size

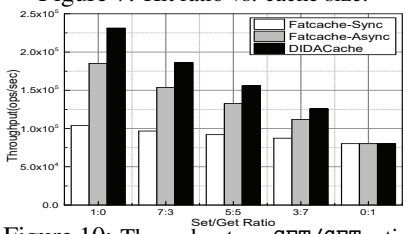


Figure 10: Throughput vs. SET/GET ratio.

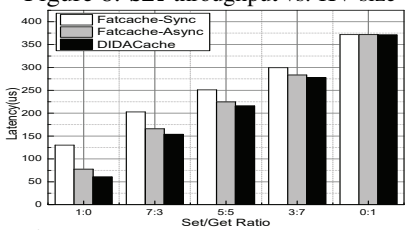


Figure 11: Latency vs. SET/GET ratio.

increases. As the object size increases, the relative GC efficiency improves and the valid data copy overhead is decreased. It is worth noting that the practical systems are typically dominated by small key-value objects, on which DIDACache performs particularly well.

Figure 9 gives the average latency for SET operations with different key-value object sizes. Similarly, it can be observed that Fatcache-Sync performs the worst, and DIDACache outperforms the other two significantly. For example, for 64-byte objects, compared with Fatcache-Sync and Fatcache-Async, DIDACache reduces the average latency by 54.5% and 23.6%, respectively.

Figures 10 and 11 show the throughput and latency for workloads with mixed SET/GET operations. Due to the space constraint, we only show results for the case with 256-byte key-value items, and other cases with different key-value sizes show similar trend. We can observe that DIDACache outperforms Fatcache-Sync and Fatcache-Async across the board, but as the portion of GET operations increases, the related performance gain reduces. Although we also optimize the path of processing GET, such as removing intermediate mapping, the main performance bottleneck is the raw flash read. Thus, with the workload of 100% GET, the latency and throughput of the three schemes are nearly the same. Figure 12 shows the latency distributions for key-value items of 64 bytes with different SET/GET ratios.

5.4.2 Memory Slab Buffer

Memory slab buffer enables the asynchronous operations of the drain and GC processes. To show the effect of slab buffer size, we vary the slab buffer size from 128MB to 1GB and test the average latency and throughput with the workloads generated with the truncated Generalized Pareto distribution. As shown in Figure 13 and Figure 14, for both SET and GET operations, the average latency and throughput are insensitive to the slab buffer size, indicating that a small in-memory slab buffer size (128M) is sufficient.

Table 1: Garbage collection overhead.

GC Scheme	Key-values	Flash Page	Erase
DIDACache-Space	7.48GB	N/A	4,231
DIDACache-Locality	0	N/A	3,679
DIDACache	2.05GB	N/A	3,829
Fatcache-Greedy	7.48GB	5.73GB	5,024
Fatcache-Kick	0	3.86GB	4,122
Fatcache-FIFO	15.35GB	0	5,316

5.4.3 Garbage Collection

Our cross-layer solution also effectively reduces the GC overhead, such as erase and valid page copy operations. In our cache-driven system, we can easily count erase and page copy operations in the library code. However, we cannot directly obtain these values on the commercial SSD as they are hidden at the device level. For effective comparison, we use the SSD simulator (extension to DiskSim [19]) from Microsoft Research and configure it with the same parameters of the commercial SSD. We first run the stock Fatcache on the commercial SSD and collect traces by using `blktrace` in Linux, and then replay the traces on the simulator. We compare our results with the simulator-generated results. In our experiments, we confine the available SSD size to 30GB, and preload it with 25GB data with workloads generated with the truncated Generalized Pareto distribution, and then do SET operations (80 million requests, about 30GB), following the Normal distribution.

Table 1 shows GC overhead in terms of valid data copies (key-values and flash pages) and block erases. We compare DIDACache using space-based eviction only ("DIDACache-Space"), locality-based eviction only ("DIDACache-Locality"), the adaptively selected eviction approach ("DIDACache") with the stock Fatcache using three schemes ("Fatcache-Greedy", "Fatcache-Kick", and "Fatcache-FIFO"). In Fatcache, the application-level GC has two options, copying valid key-value items from the victim slab for retaining hit ratio or directly dropping the entire slab for speed. This incurs different overheads of key-value copy

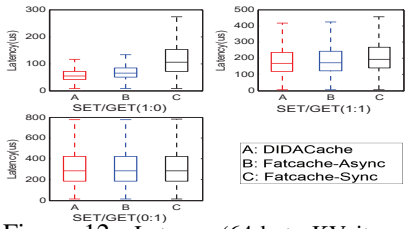


Figure 12: Latency (64-byte KV items) with different SET/GET ratios.

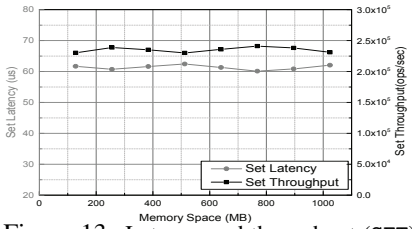


Figure 13: Latency and throughput (SET) with different buffer sizes.

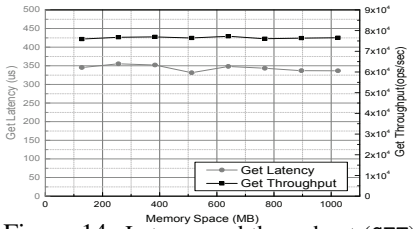


Figure 14: Latency and throughput (GET) with different buffer sizes.

operations, denoted as "Key-values". In this experiment, both Fatcache-Greedy and Fatcache-Kick use a greedy algorithm to find a victim slab, but the former performs key-value copy operations while the latter does not. Fatcache-FIFO uses a FIFO algorithm to find the victim slab and copies still-valid key-values. In the table, the flash page copy and block erase operations incurred by the device-level GC are denoted as "Flash Page" and "Erase", respectively.

Fatcache schemes show high GC overheads. For example, both Fatcache-Greedy and Fatcache-FIFO recycle valid key-value items at the application level, incurring a large volume of key-value copies. Fatcache-Kick, in contrast, aggressively drops victim slabs without any key-value copy. However, since it adopts a greedy policy (as Fatcache-Greedy) to evict the slabs with least valid key-value items, erase blocks are mixed with valid and invalid pages, which incurs flash page copies by the device-level GC. Fatcache-FIFO fills and erases all slabs in a sequential FIFO manner, thus, no device-level flash page copy is needed. All three Fatcache schemes show a large number of block erases.

The GC process in our scheme is directly driven by the key-value cache. It performs a fine-grained, single-level, key-value item-based reclamation, and no flash page copy is needed (denoted as "N/A" in Table 1). The locality-based eviction policy enjoys the minimum data copy overhead, since it aggressively evicts the LRU slab without copying any valid key-value items. The space-based eviction policy needs to copy 7.48 GB key-value items and incurs 4,231 erase operations. DIDACache dynamically chooses the most appropriate policy at runtime, so it incurs a GC overhead between the above two (2.05 GB data copy and 3,829 erases). Compared to Fatcache schemes, the overheads are much lower (e.g., 28% lower than Fatcache-FIFO).

5.4.4 Dynamic Over-Provisioning Space

To illustrate the effect of our dynamic OPS management, we run DIDACache on our testbed that simulates the data center environment in Section 5.3. We use the same data set containing 800 million key-value pairs (about 250GB), and the request sequence generated with the Normal distribution model. We set the cache size as 12% (around 30GB) of the data set size. In the experiment, we first warm up the cache server with the generated data, and then change the request coming rates to test our dynamic OPS policies.

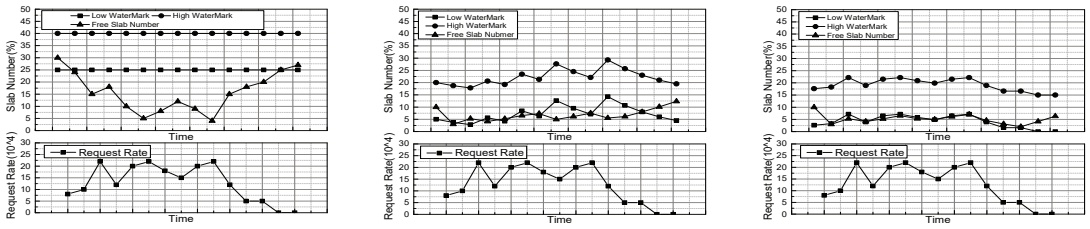
Table 2: Effect of different OPS policies.

GC Scheme	Hit Ratio	GC	Latency	Throughput
Static	87.7 %	2716	79.95	198,076
Heuristic	94.1 %	2480	64.24	223,146
Queuing	94.8 %	2288	62.41	229,956

Figure 15 shows the dynamic OPS and the number of free slabs with the varying request incoming rates for three different policies. The static policy reserves 25% of flash space as OPS to simulate the conventional SSD. For the heuristic policy, we set the initial W_{low} with 5%. For the queuing theory policy, we use the model built in Equation 3 to determine the value of W_{low} at runtime. We set W_{high} 15% higher than W_{low}. The GC is triggered when the number of free slabs drops below W_{high}.

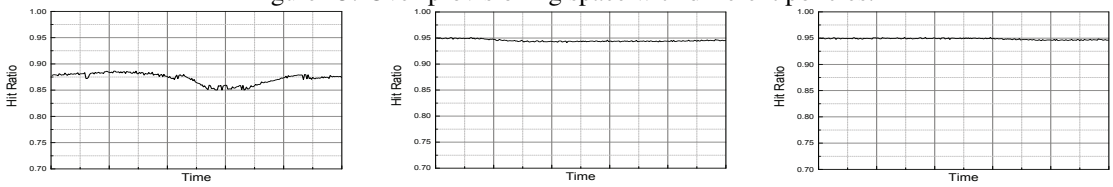
As shown in Figure 15(a), the static policy reserves a portion of flash space for over-provisioning. The number of free slabs fluctuates, responding to the incoming request rate. In Figure 15(b), our heuristic policy dynamically changes the two watermarks. When the arrival rate of requests increases, the low watermark, W_{low}, increases to aggressively generate free slabs by using *quick clean*. The number of free slabs approximately follows the trend of the low watermark, but we can also see a lag-behind effect. Our queuing policy in Figure 15(c) performs even better, and it can be observed that the free slab curve almost overlaps with the low watermark curve. Compared with the static policy, both heuristic and queuing theory policies enable a much larger flash space for caching. Accordingly, we can see in Figure 16 that the two dynamic OPS policies are able to maintain a hit ratio close to 95%, which is 7% to 10% higher than the static policy. Figure 17 shows the GC cost, and we can find that the two dynamic policies incur lower overhead than the static policy. In fact, compared with the static policy and the heuristic policy, the queuing theory policy erases 15.7% and 8% less flash blocks, respectively. Correspondingly, in Figure 18, it can be observed that the queuing policy can most effectively reduce the number of requests with high latencies.

To further study the difference of these three policies, we also compared their runtime throughput in Table 2. We can see that the static policy has the lowest throughput (198,076 ops/sec). The heuristic and queuing theory policies can deliver higher throughput, 223,146 and 229,956 ops/sec, respectively.



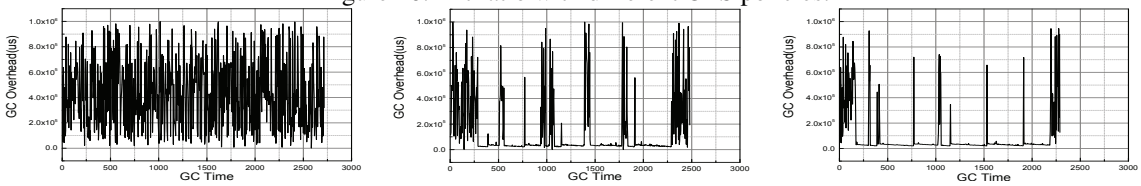
(a) Static policy. (b) Heuristic policy. (c) Queuing theory policy.

Figure 15: Over-provisioning space with different policies.



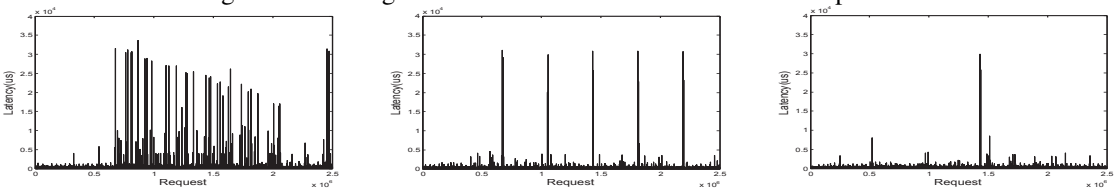
(a) Static policy. (b) Heuristic policy. (c) Queuing theory policy.

Figure 16: Hit ratio with different OPS policies.



(a) Static policy. (b) Heuristic policy. (c) Queuing theory policy.

Figure 17: Garbage collection overhead with different OPS policies.



(a) Static policy. (b) Heuristic policy. (c) Queuing theory policy.

Figure 18: Request latency with different OPS policies.

5.5 Overhead Analysis

DIDACache is highly optimized for key-value caching and moves certain device-level functions up to the application level. This could raise consumption of host-side resources, especially memory and CPU.

Memory Utilization: In DIDACache, memory is mainly used for three purposes. (1) In-memory hash table. DIDACache maintains a host-side hash table with 44-byte mapping entries (`<md, sid, offset>`), which is identical to the stock Fatcache. (2) Slab buffer. DIDACache performance is insensitive to the slab buffer size. We use a 128MB memory for slab buffer, which is also identical to the stock Fatcache. (3) Slab metadata. For slab allocation and GC, DIDACache introduces two additional queues (*Free Slab Queue* and *Full Slab Queue*) for each channel. Each queue entry is 8 bytes, corresponding to a slab. Each slab also maintains an erase count and a valid data ratio, each requiring 4 bytes. Thus, in total, DIDACache adds 16-byte metadata for each slab. For a 1TB SSD with a regular slab size of 8MB, it consumes at most 2MB memory. In our experiments, we found that the memory consumptions of DIDACache and Fatcache are almost identical during runtime. Also note that the device-side demand for memory is significantly decreased, such as the removed FTL-level mapping table.

Table 3: CPU utilization of different schemes.

Scheme	SET	GET	SET/GET (1:1)
DIDACache	47.7%	20.5 %	37.4 %
Fatcache-Async	42.3 %	20 %	33.8 %
Fatcache-Sync	40.1 %	20 %	31.3 %

CPU utilization: DIDACache is multi-threaded. In particular, we maintain 12 threads for monitoring the load of each channel, one global thread for garbage collection, and one load-monitoring thread for determining the OPS size. To show the related computational cost, we compare the CPU utilization of DIDACache, Fatcache-Async, and Fatcache-Sync in Table 3. It can be observed that DIDACache only incurs marginal increase of the host-side CPU utilization. In the worst case (100% `SET`), DIDACache only consumes extra 7.6% and 5.4% CPU resources over Fatcache-Sync (40.1%) and Fatcache-Async (42.3%), respectively. Finally it is worth noting that DIDACache removes much device-level processing, such as GC, which simplifies device hardware.

Cost implications: DIDACache is cost efficient. As an application-driven design, the device hardware can be greatly simplified for lower cost. For example, the DRAM required for the on-device mapping table can be removed and the reserved flash space for OPS can be saved. At the same time, our results also show that the host-side overhead, as well as the additional utilization of the host-side resources are minor.

6 Other Related Work

Both flash memory [3, 7–9, 11, 16, 20, 22, 26, 29, 41, 42] and key-value systems [4, 5, 10, 14, 24, 25, 47, 49] are extensively researched. This section discusses prior studies most related to this paper.

A recent research interest in flash memory is to investigate the interaction between applications and underlying flash storage devices. Yang et al. investigate the interactions between log-structured applications and the underlying flash devices [48]. Differentiated Storage Services [32] proposes to optimize storage management with semantic hints from applications. Nameless Writes [50] is a de-indirection scheme to allow writing only data into the device and let the device choose the physical location. Similarly, FSDV [51] removes the FTL level mapping by directly storing physical flash addresses in the file systems. Willow [40] exploits on-device programmability to move certain computation from the host to the device. FlashTier [39] uses a customized flash translation layer optimized for caching rather than storage. OP-FCL dynamically manages OPS on SSD to balance the space needs for GC and for caching [34]. RIPQ [44] optimizes the photo caching in Facebook particularly for flash by reshaping the small random writes to a flash-friendly workload. Our solution shares a similar principle of removing unnecessary intermediate layers and collapsing multi-layer mapping into only one, but we particularly focus on tightly connecting key-value cache systems and the underlying flash SSD hardware.

Key-value cache systems recently show its practical importance in Internet services [5, 14, 25, 49]. A report from Facebook discusses their efforts of scaling Memcached to handle the huge amount of Internet I/O traffic [33]. McDipper [12] is their latest effort on flash-based key-value caching. Several prior research studies specifically optimize key-value store/cache for flash. Ouyang et al. propose an SSD-assisted hybrid memory for Memcached in high performance network [36]. This solution essentially takes flash as a swapping device. NVMKV [27, 28] gives an optimized key-value store based on flash devices with several new designs, such as dynamic mapping, transactional support, and parallelization. Unlike NVMKV, our system is a key-value cache, which allows us to aggressively integrate the two layers together and exploit some unique opportunities. For example, we can invalidate all slots and erase an entire flash block, since we are dealing with a cache rather than storage.

Some prior work also leverages Open-Channel SSDs for domain optimizations. For example, Ouyang et al. present SDF [35] for web-scale storage. Wang et al. further present a design of LSM-tree based key-value store on the same platform, called LOCS [46]. Instead of simplifying redundant functions at different levels, they focus on enabling applications to take use of internal parallelism of flash channels through using Open-Channel SSD. Lee et al. [21] also propose an application-managed flash for file systems. We share the common principle of bridging the semantic gap and aim to deeply integrate device and key-value cache management.

7 Conclusions

Key-value cache systems are crucial to low-latency high-throughput data processing. In this paper, we present a co-design approach to deeply integrate the key-value cache system design with the flash hardware. Our solution enables three key benefits, namely a single-level direct mapping from keys to physical flash memory locations, a cache-driven fine-grained garbage collection, and an adaptive over-provisioning scheme. We implemented a prototype on real Open-Channel SSD hardware platform. Our experimental results show that we can significantly increase the throughput by 35.5%, reduce the latency by 23.6%, and remove unnecessary erase operations by 28%.

Although this paper focuses on key-value caching, such an integrated approach can be generalized and applied to other semantic-rich applications. For example, for file systems and databases, which have complex mapping structures in different levels, our unified direct mapping scheme can also be applied. For read-intensive applications with varying patterns, our dynamic OPS approach would be highly beneficial. Various applications may benefit from different policies or different degrees of integration with our schemes. As our future work, we plan to further generalize some functionality to provide fine-grained control on flash operations and allow applications to flexibly select suitable schemes and reduce development overheads.

Acknowledgments

We thank our shepherd, Gala Yadgar, and the anonymous reviewers for their constructive comments. This work is partially supported by National Natural Science Foundation of China (Project 61373049), Research Grants Council of Hong Kong (GRF 152736/16E and GRF 15222315/15E), Hong Kong Polytechnic University (4-BCBB), Louisiana Board of Regents LEQSF(2014-17)-RD-A-01, and U.S. National Science Foundation (CCF-1453705, CCF-1629291).

References

[1] Fatcache-Async. https://github.com/polyu-szy/Fatcache-Async-2017.

[2] Whitepaper: memcached total cost of ownership (TCO). https://davisfields.files.wordpress.com/2011/06/gear6_white_paper_tco.pdf.

[3] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference (ATC 08)* (2008).

[4] ANAND, A., MUTHUKRISHNAN, C., KAPPES, S., AKELLA, A., AND NATH, S. Cheap and large CAMs for high performance data-intensive networked systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)* (2010).

[5] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review (SIGMETRICS 12)* (2012).

[6] CARRA, D., AND MICHIARDI, P. Memory partitioning in Memcached: an experimental performance analysis. In *International Conference on Communications (ICC 14)* (2014).

[7] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 09)* (2009).

[8] CHEN, F., LEE, R., AND ZHANG, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *International Symposium on High Performance Computer Architecture (HPCA 11)* (2011).

[9] CHEN, F., LUO, T., AND ZHANG, X. CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *USENIX Conference on File and Storage Technologies (FAST'11)* (2011).

[10] DEBNATH, B., SENGUPTA, S., AND LI, J. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 11)* (2011).

[11] DIRIK, C., AND JACOB, B. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In *International Symposium on Computer Architecture (ISCA 09)* (2009).

[12] FACEBOOK. McDipper: a key-value cache for flash storage. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920.

[13] GAL, E., AND TOLEDO, S. Algorithms and data structures for flash memories. In *ACM Computing Survey (CSUR)* (2005), vol. 37:2.

[14] GOKHALE, S., AGRAWAL, N., NOONAN, S., AND UNGUREANU, C. KVZone and the search for a write-optimized key-value store. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 10)* (2010).

[15] GONZÁLEZ, J., BJØRLING, M., LEE, S., DONG, C., AND HUANG, Y. R. Application-driven flash translation layers on Open-Channel SSDs.

[16] 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 *International Symposium on Microarchitecture (Micro 09)* (2009).

[17] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 09)* (2009).

[18] HU, X., WANG, X., LI, Y., ZHOU, L., LUO, Y., DING, C., JIANG, S., AND WANG, Z. LAMA: optimized locality-aware memory allocation for key-value cache. In *USENIX Annual Technical Conference (ATC 15)* (2015).

[19] JOHN BUCY, JIRI SCHINDLER, S. S. G. G. DiskSim 4.0. http://www.pdl.cmu.edu/DiskSim/.

[20] KLIMOVIC, A., KOZYRAKIS, C., THEREKSA, E., JOHN, B., AND KUMAR, S. Flash storage disaggregation. In *The Eleventh European Conference on Computer Systems (EuroSys 16)* (2016).

[21] LEE, S., LIU, M., JUN, S., XU, S., KIM, J., ET AL. Application-managed flash. In *USENIX Conference on File and Storage Technologies (FAST 16)* (2016).

[22] LEVENTHAL, A. Flash storage memory. In *Communications of the ACM* (2008), vol. 51(7), pp. 47–51.

[23] LILLY, P. Facebook ditches DRAM, flaunts flash-based McDipper. http://www.maximumpc.com/facebook-ditches-dram-flaunts-flash-based-mcdipper.

[24] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: a memory-efficient, high-performance key-value store. In *ACM Symposium on Operating Systems Principles (SOSP 11)* (2011).

[25] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: separating keys from values in SSD-conscious storage. In *USENIX Conference on File and Storage Technologies (FAST 16)* (2016).

[26] MARGAGLIA, F., YADGAR, G., YAAKOBI, E., LI, Y., SCHUSTER, A., AND BRINKMANN, A. The devil is in the details: implementing flash page reuse with WOM codes. In *USENIX Conference on File and Storage Technologies (FAST 16)* (2016).

[27] MÁRMOL, L., SUNDARARAMAN, S., TALAGALA, N., AND RANGASWAMI, R. NVMKV: a scalable and lightweight, FTL-aware key-value store. In *USENIX Annual Technical Conference (ATC 15)* (2015).

[28] MÁRMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. NVMKV: a scalable and lightweight flash aware key-value store. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)* (2015).

[29] MARSH, B., DOUGLIS, F., AND KRISHNAN, P. Flash memory file caching for mobile computers. In *Hawaii Conference on Systems Science* (1994).

[30] MEMBLAZE. Memblaze. http://www.memblaze.com/en/.

[31] MEMCACHED. Memcached: a distributed memory object caching system. http://www.memcached.org.

[32] MESNIER, M. P., AKERS, J., CHEN, F., AND LUO, T. Differentiated storage services. In *ACM Symposium on Operating System Principles (SOSP 11)* (2011).

[33] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013).

[34] OH, Y., CHOI, J., LEE, D., AND NOH, S. H. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *USENIX Conference on File and Storage Technologies (FAST 12)* (2012).

[35] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. SDF: software-defined flash for web-scale internet storage systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 14)* (2014).

[36] OUYANG, X., ISLAM, N. S., RAJACHANDRASEKAR, R., JOSE, J., LUO, M., WANG, H., AND PANDA, D. K. SSD-assisted hybrid memory to accelerate memcached over high performance networks. In *International Conference for Parallel Processing (ICPP 12)* (2012).

[37] REDIS. http://redis.io/.

[38] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems (TC 92)* (1992), vol. 10(1):26-52.

[39] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. Flashtier: a lightweight, consistent and durable storage cache. In *The European Conference on Computer Systems (EuroSys 12)* (2012).

[40] SESHADRI, S., GAHAGAN, M., BHASKARAN, S., BUNKER, T., DE, A., JIN, Y., LIU, Y., AND SWANSON, S. Willow: a user-programmable SSD. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014).

[41] SHAFAEI, M., DESNOYERS, P., AND FITZPATRICK, J. Write amplification reduction in flash-based SSDs through extent-based temperature identification. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (2016).

[42] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending SSD lifetimes with disk-based write caches. In *USENIX Conference on File and Storage Technologies (FAST 10)* (2010).

[43] T13. T13 documents referring to TRIM. http://t13.org/Documents/MinutesDefault.aspx?keyword=trim.

[44] TANG, L., HUANG, Q., LLOYD, W., KUMAR, S., AND LI, K. RIPQ: advanced photo caching on flash for facebook. In *USENIX Conference on File and Storage Technologies (FAST 15)* (2015).

[45] TWITTER. Fatcache. https://github.com/twitter/fatcache.

[46] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of LSM-tree based key-value store on Open-Channel SSD. In *The European Conference on Computer Systems (EuroSys 15)* (2015).

[47] 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 *USENIX Annual Technical Conference (ATC 15)* (2015).

[48] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., AND SUNDARARAMAN, S. Don't stack your log on my log. In *Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)* (2014).

[49] ZHANG, H., DONG, M., AND CHEN, H. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *USENIX Conference on File and Storage Technologies (FAST 16)* (2016).

[50] ZHANG, Y., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. De-indirection for flash-based SSDs with nameless writes. In *USENIX Conference on File and Storage Technologies (FAST 12)* (2012).

[51] ZHANG, Y., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Removing the costs and retaining the benefits of flash-based SSD virtualization with FSDV. In *International Conference on Massive Storage Systems and Technology (MSST 15)* (2015).

[52] ZHANG, Y., SOUNDARARAJAN, G., STORER, M. W., BAIRAVASUNDARAM, L. N., SUBBIAH, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Warming up storage-level caches with bonfire. In *USENIX Conference on File and Storage Technologies (FAST 13)* (2013).

[53] ZHENG, M., TUCEK, J., HUANG, D., QIN, F., LILLIBRIDGE, M., YANG, E. S., ZHAO, B. W., AND SINGH, S. Torturing databases for fun and profit. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014).

[54] ZHENG, M., TUCEK, J., QIN, F., AND LILLIBRIDGE, M. Understanding the robustness of SSDs under power fault. In *USENIX Conference on File and Storage Technologies (FAST 13)* (2013).