conference

proceedings

19th USENIX Conference on File and Storage Technologies (FAST '21)

February 23–25, 2021

Sponsored by

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

In cooperation with ACM SIGOPS

usenix
ASSOCIATION

Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)

February 23–25, 2021

ISBN 978-1-939133-20-5

FAST ’21 Sponsors

Gold Sponsor

NetApp®

Silver Sponsors

amazon

FUTUREWEI Technologies

Google

vmware®

Bronze Sponsors

FACEBOOK

IBM®

SNIA®

Open Access Sponsor

NetApp®

Industry Partners and Media Sponsors

ACM *Queue*
FreeBSD Foundation
No Starch Press

USENIX Supporters

USENIX Patrons

Bloomberg • Facebook • Google
Microsoft • NetApp

USENIX Benefactors

Amazon • Auristor • Discernible
Thinkst Canary • Transcend • Two Sigma

USENIX Partner

Top10VPN

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST ’21)

February 23–25, 2021

ISBN 978-1-939133-20-5

Conference Organizers

Program Co-Chairs
Marcos K. Aguilera, *VMware Research*
Gala Yadgar, *Technion—Israel Institute of Technology*

Program Committee
Nitin Agrawal, *ThoughtSpot*
Marcos K. Aguilera, *VMware Research*
Woongki Baek, *UNIST (Ulsan National Institute of Science and Technology)*
Mahesh Balakrishnan, *Facebook*
Suparna Bhattacharya, *Hewlett Packard Enterprise*
Janki Bhimani, *Florida International University*
Angelos Bilas, *University of Crete and FORTH*
Randal Burns, *Johns Hopkins University*
Feng Chen, *Louisiana State University*
Vijay Chidambaram, *The University of Texas at Austin and VMware Research*
Natacha Crooks, *University of California, Berkeley*
Daniel Ellard, *Raytheon BBN Technologies*
Danny Harnik, *IBM Research—Haifa*
Dean Hildebrand, *Google*
Cheng Huang, *Microsoft*
William Jannen, *Williams College*
Song Jiang, *The University of Texas at Arlington*
Rob Johnson, *VMware Research*
Kimberly Keeton
Patrick P. C. Lee, *The Chinese University of Hong Kong*
Xiaosong Ma, *Qatar Computing Research Institute, HBKU*
Peter Macko, *NetApp*
Ethan L. Miller, *University of California, Santa Cruz, and Pure Storage*
Dalit Naor, *The Academic College of Tel Aviv–Yaffo*
Don Porter, *The University of North Carolina at Chapel Hill*
Rob Ross, *Argonne National Laboratory*
Ken Salem, *University of Waterloo and Amazon*
Jiri Schindler, *Tranquil Data*
Russell Sears, *Apple*
Mehul A. Shah, *Amazon AWS*
Keith A. Smith, *MongoDB*
Amy Tai, *VMware Research*
Vasily Tarasov, *IBM Research*
Carl Waldspurger, *Carl Waldspurger Consulting*
Youjip Won, *Korea Advanced Institute of Science and Technology (KAIST)*
Gala Yadgar, *Technion—Israel Institute of Technology*

Test of Time Awards Committee
Jiri Schindler, *Tranquil Data*
Bianca Schroeder, *University of Toronto*

Work-in-Progress Reports (WiPs) Co-Chairs
Peter Macko, *NetApp*
Amy Tai, *VMware Research*

Tutorial Coordinators
Andy Klosterman, *NetApp*
John Strunk, *Red Hat*

Steering Committee
Nitin Agrawal, *ThoughtSpot*
Angela Demke Brown, *University of Toronto*
Casey Henderson, *USENIX Association*
Kimberly Keeton
Geoff Kuenning, *Harvey Mudd College*
Arif Merchant, *Google*
Sam H. Noh, *UNIST (Ulsan National Institute of Science and Technology)*
Raju Rangaswami, *Florida International University*
Erik Riedel
Jiri Schindler, *Tranquil Data*
Bianca Schroeder, *University of Toronto*
Keith A. Smith, *MongoDB*
Eno Thereska, *Amazon*
Carl Waldspurger, *Carl Waldspurger Consulting*
Hakim Weatherspoon, *Cornell University*
Brent Welch, *Google*
Ric Wheeler, *Facebook*
Erez Zadok, *Stony Brook University*

External Reviewers

Amogh Akshintala
Deniz Altinbüken
Irina Calciu
Yuval Cassuto
Moshe Gabel
Aishwarya Ganesan
Myungsuk Kim
Jean-Sebastien Legare
Beomseok Nam
Aalap Tripathy
Cong Xu
Eitan Yaakobi
Jun Yuan

Message from the
FAST ’21 Program Co-Chairs

Welcome to the 19th USENIX Conference on File and Storage Technologies (FAST ’21). This year’s conference continues the tradition of bringing together researchers and practitioners from both industry and academia for a program of innovative and rigorous storage-related research. It is, however, the first time that FAST is held online due to the current worldwide travel restrictions. We are pleased to present a diverse set of papers on topics such as cloud storage, key-value stores, consistency, reliability, caching, HPC systems, SSD, and traditional file systems. Submissions to the conference came from authors representing academia, industry, and the open-source community.

FAST ’21 received 130 submissions. Of these, we accepted 28 papers, for an acceptance rate of 21%. The Program Committee used a two-round online review process and then held a two-day virtual PC meeting to select the final program. In the first round, each paper was assigned three reviewers. In the second round, 80 papers were assigned at least two more reviews. The Program Committee discussed 35 papers in the PC meeting on December 7–8, 2020, spanning 17 time-zones. We used Eddie Kohler’s excellent HotCRP service to manage all stages of the review process, from submission to author notification.

As in the previous years, we included a category of deployed-systems papers, which address experience with the practical design, implementation, analysis, or deployment of large-scale, operational systems. We received 8 deployed-systems submissions and we accepted 3. Unlike previous years, there was no special category for short papers.

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 ’21. We would also like to thank the attendees of FAST ’21 and the future readers of these papers. Together with the authors, you form the FAST community and make storage research vibrant and exciting. We extend our thanks to the entire USENIX staff, especially Casey Henderson, Jasmine Murcia, and Arnold Gatilao, who have provided outstanding support throughout the planning and organizing of this conference with the highest degree of professionalism and friendliness. Most importantly, their behind-the-scenes work makes this conference actually happen. We would like to thank the Work-in-Progress Session Chairs, Peter Macko and Amy Tai. Our thanks go also to the members of the FAST Steering Committee who provided invaluable advice and feedback, and to our Steering Committee Liaison, Keith Smith, for his guidance and encouragement on many issues, large and small, over the past year.

Finally, we wish to thank our Program Committee for their many hours of hard work reviewing, discussing, and shepherding the submissions. In total, the PC wrote 557 thoughtful and meticulous reviews and 1491 online comments. HotCRP recorded approximately 375,000 words in reviews and comments (excluding HotCRP boilerplate language). The reviewers’ evaluations, and their thorough and conscientious deliberations at the PC meeting, contributed significantly to the quality of our decisions. Each paper had a shepherd that reviewed the final submission and provided additional feedback. In many cases, this led to significant improvements in the final quality of the submissions. We look forward to an interesting and enjoyable conference!

Marcos K. Aguilera, *VMware Research*
Gala Yadgar, *Technion—Israel Institute of Technology*
FAST ’21 Program Co-Chairs

19th USENIX Conference on File and Storage Technologies (FAST '21)

February 23–25, 2021

Tuesday, February 23

Indexing and Key-Value Store

Advanced File Systems

Wednesday, February 24

Transactions, Deduplication, and More

Cloud and Distributed Systems

Thursday, February 25

Caching Everywhere

The SSD Revolution Is Not Over

ROART: Range-query Optimized Persistent ART

Shaonan Ma[1], Kang Chen[1]*, Shimin Chen[2,3], Mengxing Liu[1],
Jianglang Zhu[1], Hongbo Kang[1], and Yongwei Wu[1]
[1]*Tsinghua University*,† [2]*SKL of Computer Architecture, ICT, CAS*
[3]*University of Chinese Academy of Sciences*

Abstract

With the availability of commercial NVM devices such as Intel Optane DC PMM, it is time to start thinking about applying the existing persistent data structures in practice. This paper considers three practical aspects, which have significant influences on the design of persistent indexes, including **functionality, performance** and **correctness**.

We design a new persistent index, ROART, based on adaptive radix tree (ART), taking all these practical aspects into account. ROART (i) proposes a *leaf compaction* method to reduce pointer chasing for range queries, (ii) minimizes persistence overhead with three optimizations, i.e., *entry compression*, *selective metadata persistence* and *minimally ordered split*, and (iii) designs a fast memory management to prevent memory leaks, and eliminates the long recovery time by proposing an *instant restart* strategy. Evaluations show that ROART outperforms the state-of-the-art radix tree by up to 1.65× and B$^+$-Trees by 1.17∼8.27× respectively.

1 Introduction

Emerging Non-Volatile Memory (NVM) is attractive because of its byte-addressability, low latency and durability. Many researchers have focused on how to design fast persistent data structures [1–30]. With the announcement of the first generation products (Intel Optane DC PMM [31]), it is time to investigate how to apply the achieved results in practice.

We point out that there are three significant aspects affecting the design of persistent indexes, i.e., functionality, performance, and correctness. Any persistent index designed for practical uses needs to consider these aspects carefully.

1. Functionality: Variable-sized Keys and Range Queries. Variable-sized keys are important in real world systems, such as RDBMS [32–41] and Key-Value Stores [25, 42–47]. Whether an index supports variable-sized keys will have a great impact on the direction of its optimization. Moreover, range queries are used to support inequality comparisons in many real-world applications [32–41]. Therefore, it is desirable that the index structure supports range queries efficiently in addition to point read and write operations. In this paper, we focus on index structures that support both variable-sized keys and range queries.

2. Performance: Persistence Overhead. Persistence overhead plays an essential role in the performance of indexes targeting NVM. To guarantee crash consistency of indexes, once an operation is completed, the modification must be persisted to NVM by cache line flush and memory fence instructions. Due to the design of hardware, NVM writes have lower throughput than reads and poor scalability of bandwidth (because the writes issued by more threads exceed the capability of underlying buffers [48]). Moreover, persistence operations incur much larger (e.g., at least by 2.4×) overhead than normal writes.

3. Correctness: Anomaly Resolution and Memory Safety. First, persistent indexes may suffer from anomalies [26], such as lost update and dirty read, if they provide no protection to concurrent operations. These anomalies will cause the effect of successful operations to disappear after system crash and restart. Second, memory allocations in NVM need to deal with crash consistency, which is not a problem in DRAM. Memory leaks may happen after a crash due to (i) inconsistent memory allocation metadata, and/or (ii) lazy GC (garbage collection) used in the design of non-blocking data structures.

In order to support range queries, our work mainly focuses on tree-based indexes rather than hash tables. According to our experiments (§2.1), B$^+$-Trees may not be the best performing index for variable-sized keys. Therefore, we choose the radix tree as our basis, which naturally supports variable-sized keys. To the best of our knowledge, no recent persistent radix tree has fully taken the above aspects into account. WORT [19] is write-optimized but only runs in a single thread. P-ART [24] is a concurrent persistent radix tree converted

*Corresponding author: Kang Chen (chenkang@tsinghua.edu.cn)

†Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China

from the volatile ART [49, 50] index based on the principles of RECIPE. There is little optimization on range queries and persistence overhead. Moreover, neither of the two radix trees considers memory safety, which may lead to memory leaks.

We propose a new index structure called ROART (**R**ange-query **O**ptimized **A**daptive **R**adix **T**ree), considering all the above factors. To improve range queries, we propose *leaf compaction* (LC) that delays the leaf split and compacts multiple leaf nodes into a leaf array. The benefits of this technique are threefold. First, it reduces pointer chasing during range queries. Second, it can decrease the number of complex split operations. Finally, it tends to lower the height of the tree, which is beneficial to all index operations.

To reduce persistence overhead, we propose three optimizations in ROART: (i) *Entry compression* (EC) that combines the key and the child pointer in an 8-byte entry; (ii) *Selective Metadata Persistence* (SMP) to reduce the amount of metadata to persist; and (iii) *minimally ordered split* (MO) that relaxes the order of steps in a split operation to reduce the number of `sfence` instructions. Previously mentioned LC also helps here because it delays leaf node split and reduces its persistence overhead.

For correctness, we protect ROART against anomalies by using `non-temporal store` [24, 48] techniques. For memory safety, there have been several previous proposals. Logging-based allocators [51–53] suffer from heavy persistence overhead for allocation/deallocation operations. Post-crash garbage collection techniques [22, 54–56] reduce the persistence overhead, but introduce long recovery time. We propose a new technique, called *instant restart*, with concurrent post-crash garbage collection. While performing the background GC during recovery, indexes can handle foreground requests concurrently, i.e., restarting services instantly. We integrate this technique in our new memory allocator, called DCMM (**D**elayed **C**heck **M**emory **M**anagement).

In summary, this paper makes the following contributions:

(1) We present an in-depth analysis on the three practical aspects of persistent indexes to understand the impact of different design choices (§2).

(2) We propose ROART that addresses the three design aspects (§3). For functionality, we choose ART as basis to naturally support variable-sized keys and propose a *leaf compaction* method to optimize range queries. For performance, we propose three techniques to reduce persistence overhead, i.e., *entry compression*, *selective metadata persistence* and *minimally ordered split*. For correctness, we carefully protect ROART against anomalies, and design DCMM with *instant restart* to support memory safety without long recovery time.

(3) We perform extensive experiments to compare ROART with state-of-the-art tree-based indexes (§4), including P-ART [24], PMwCAS-ART (implemented with PMwCAS [23]), FAST&FAIR [5], SkipList [22] and BzTree [6]. ROART outperforms the existing solutions by 1.15∼8.27× under YCSB workloads.

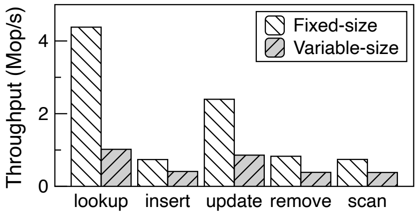


Figure 1: Performance degradation of storing variable-size data in FAST&FAIR.

2 Practical Considerations

We discuss the three aspects, i.e. functionality, persistence overhead, and correctness, in detail in this section.

2.1 Functionality

2.1.1 Variable-sized Keys

Indexes supporting variable-sized keys can be applicable to a wider range of applications, including database systems [32–41]. However, such support may come with a price.

A number of persistent B^{+}-Tree indexes maintain fixed-sized (8-byte key and 8-byte value) entries in the array of nodes such as NVTree [3], wB^{+}-Tree [2], FPTree [4], RN-Tree [7] and LB^{+}-Tree [9], and entries are appended to the arrays. FAST&FAIR [5], which also supports only fixed 8-byte keys, reduces the number of `clwbs` if multiple keys and values are in the same cache line. These B^{+}-Trees have excellent cache locality and high traversal performance based on optimizing 8-byte keys only.

A straightforward way to adapt the indexes with 8-byte keys to support variable-sized keys is to allocate extra data areas and store the addresses of the keys in the indexes. However, this incurs pointer chasing overhead. We use FAST&FAIR, a state-of-the-art persistent B^{+}-Tree, as an example to reveal the performance degradation using such a method. In Figure 1, we evaluate the performance of five operations (lookup/insert/update/remove/scan) with four threads using modified FAST&FAIR. We use the above method to support variable-sized keys, and use an appropriate NVM allocator (with post-crash GC (§2.3.2)) to eliminate the persistence overhead during allocation, so that we can focus on the performance difference between fixed-sized and variable-sized keys. We find the degradations of the five operations are about 3.9/1.8/2.79/2.15/1.94× respectively. The main performance difference comes from pointer chasing and string comparison during traversal. To persist extra data areas, operations like insert/update introduce more persistence.

BzTree [6] employs a different approach, slotted pages, for variable-sized keys. Based on this approach, fixed-sized metadata grows downward into the node, and variable-sized keys and values grow upward. Such an approach can reduce pointer chasing during traversal, but introduce the cost of additional metadata [8]. The performance of BzTree is in Figure 14, its lookups are fast but writes are slow.

Indexes based on radix tree [49] have better performance (Figure 14) in supporting variable-sized keys than B^+-Tree due to less comparison in traversal. However, they also have their own shortcomings, such as inefficiency on range queries.

2.1.2 Range Queries

Range query is an important feature in RDBMS [32–41] and Key-Value Stores [25, 42–47]. This paper focuses on tree-based indexes which naturally support range queries. B^+-Trees support efficient range queries because multiple keys are stored in one leaf node, and scan in leaf nodes causes no pointer chasing. In other tree structures, such as radix trees and binary search trees, one node can only store one key, and keys can be stored in leaf nodes but also in non-leaf nodes. Range queries on these trees have to traverse different levels of the trees, and chase more pointers. As the performance gap between sequential read and random read is larger in NVM than that in DRAM [48], more random accesses deteriorate the range query performance in NVM.

However, B^+-Trees may not be the best choice for indexes when both variable-sized keys and range queries are required. In Figure 1, the scan throughput of B^+-Trees decreases by 48.5% when variable-sized keys are used. In §3, we optimize the adaptive radix tree (ART) for range queries.

2.2 Persistence Overhead

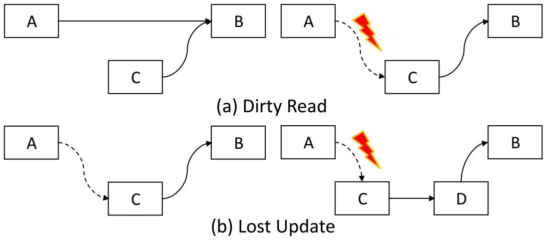
Explicit persistence is required to guarantee crash consistency for indexes in NVM. However, cache line flush and memory fence are costly compared to other instructions. Due to the poor scalability of NVM writes [48], reducing the persistence overhead is crucial for improving performance.

There are several general methods to convert a volatile index to its non-volatile counterpart in NVM. PMwCAS [23] records the metadata of each CAS (`Compare-and-Swap`) into a descriptor to ensure multiple CASs can execute atomically. RECIPE [24] adds a persistent instruction (flush and fence) after each store to ensure persistence. Unfortunately, without any optimization to reduce persistence operations, such generality comes with high overhead [8].

Other works propose special optimizations to eschew heavy persistence. wB^+-Tree [2], NVTree [3], FPTree [4], and LB^+-Tree [9] abandon the order of keys in leaf nodes to reduce writes for insertions and deletions. FAST&FAIR [5] and LB^+-Tree [9] reduce persistence operations by making data share the same cache line as much as possible. RNTree [7] uses HTM [57] to increase the granularity of atomic writes to reduce the number of persistence operations. From the above, we see that it is important to exploit the properties of target data structures. In this paper, we optimize persistence in ROART by exploiting inherent properties of ART (§3).

Table 1: States of three steps in an insert operation.

next pointer	Step (i)	Step (ii)	Step (iii)
volatile state	old	new	new
persistent state	old	old	new



(a) Dirty Read

(b) Lost Update

Figure 2: Two anomalies in unmodified lock-free linked list. The dotted line indicates that it has not been persisted.

2.3 Correctness

2.3.1 Anomaly Resolution

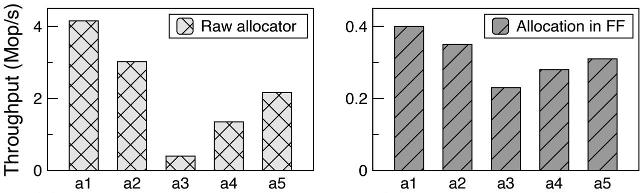
When designing lock-free non-volatile data structures, two anomalies (`Dirty Read` and `Lost Update`), are prone to occur [26]. The main reason is that threads may access data yet to be persisted, which are lost after crash and restart.

We use the lock-free linked list [58] as an example to describe the two anomalies. The insert operation has three steps. Step (i) creates a new node, sets its next pointer to point to the successor, and then persists the new node. Step (ii) updates the next pointer of the predecessor to point to the new node. Step (iii) persists the next pointer of the predecessor. The states of the predecessor's next pointer are shown in Table 1. Note that there is an inconsistency between volatile and persistent states in step (ii). Without extra protection mechanisms, two anomalies can happen, as illustrated in Figure 2.

Dirty Read. In Figure 2(a), an insert operation inserts a new node C between A and B. Suppose, the operation finishes step (ii) but has not executed step (iii) yet. At this moment, a concurrent read operation visits C. If the system crashes at this point, C is lost after restart and the read operation has read the uncommitted dirty data.

Lost Update. In Figure 2(b), two insert operations want to insert two adjacent nodes C and D between A and B. The successful result should be $A \rightarrow C \rightarrow D \rightarrow B$. Suppose that a thread completes step (i) and (ii) for inserting C. Then another thread inserts D between C and B, and completes all three steps. If the system crashes before the insert of C completes step (iii), we can only recover $A \rightarrow B$ from NVM, but lose the completed insert operation of D.

These anomalies can be fixed in various ways. For lock-free designs, `link-and-persist` [22] and `PMwCAS` [23] can be used, e.g., BzTree stays away from anomalies by `PMwCAS`, SkipList can ensure correctness by `link-and-persist`. For lock-based designs, implementations can replace `temporal store` with `non-temporal store` [24], such as P-ART [24].



(a) Performance of raw allocators (b) Allocators in FAST&FAIR

Figure 3: Comparison of different allocators.

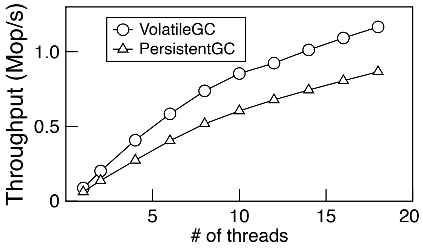


Figure 4: Volatile GC vs. Persistent GC.

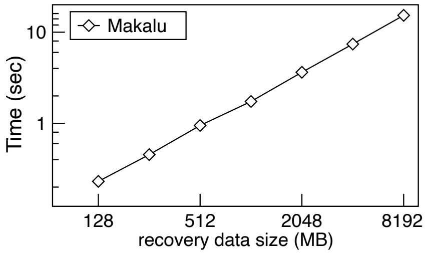


Figure 5: Recovery time of Makalu.

2.3.2 Memory Safety

It is important to guarantee crash consistency for NVM memory management. Inconsistent metadata for allocation/deallocation operations and/or lazy garbage collection can lead to memory leaks.

NVM Allocation. Many existing studies employ the simplistic solution that uses volatile allocators, such as `malloc`, `libvmmalloc` [59], for persistent memory. An allocation typically consists of three steps: (i) allocate a free NVM block, (ii) modify the allocator's metadata, and (iii) return the allocated block to the application. However, these volatile allocators pay no attention to crash consistency of allocators' metadata. Once the system crashes during the allocation process, the metadata is likely to be inconsistent, making part of the NVM memory unreachable. Moreover, volatile allocators can result in inaccurate performance evaluation for applications based on them [48], because they overlook the expensive persistent instructions. Therefore, using volatile allocators directly for NVM is not appropriate.

Correct NVM allocators are divided into two categories, logging-based allocator [51–53] and post-crash GC [22, 54–56]. The former uses logging to ensure the atomicity of operations, introducing additional persistence overhead. The latter reclaims garbage data by scanning all memory space during recovery. It reduces persistence overhead during allocation/deallocation operations, but suffers long recovery time as the amount of data increases.

Table 2: Analysis of recent works[1].

Name	Functionality		Correctness		
	Variable	Range Query	Anomaly	Allocation	GC
NVTree	●	✓	▼	▲	▼
wB$^+$-Tree	●	✓	▼	▲	▼
FPTree	●	✓	✓	▲	▼
FAST&FAIR	●	✓	✓	✖	✖
RNTree	●	✓	✓	✖	✖
WORT	✓	●	▼	✖	▼
BzTree	✓	✓	✓	✓	✓
P-ART	✓	●	✓	✖	✖
LB$^+$-Tree	●	✓	✓	▲	▼

NOTE: ▲: not open-sourced ▼: no need ●: not optimized
✓: support ✖: poor support

Allocation Performance. We evaluate five commonly used (volatile and persistent) NVM allocators as follows:

a1: malloc, the standard volatile allocator for DRAM.

a2: libvmmalloc, volatile allocator based on `jemalloc`.

a3: PMDK [51], logging-based persistent allocator.

a4: nvm_malloc [52], logging-based persistent allocator.

a5: Makalu [54], persistent allocator with post-crash GC.

Figure 3(a) shows the raw performance of the allocators. The test continuously allocates 64-byte chunks, then writes and persists them in a single thread. Makalu is 50% and 28% slower than malloc and libvmmalloc, respectively. PMDK and nvm_malloc are 81% and 38% slower than Makalu, respectively. Performance of FAST&FAIR using different allocators is shown in Figure 3(b). We see that the gaps between different cases are narrowed due to the tree's traversal overhead. Makalu is 22% and 12% slower than malloc and libvmmalloc, respectively. PMDK and nvm_malloc are 25% and 10% slower than Makalu, respectively.

Garbage Collection. Many indexes support non-blocking lookups [60–62] to improve the read performance. Lazy GC mechanism is necessary in such implementations. An item to delete is firstly labeled as *logically* deleted before it is *physically* deleted for avoiding dangerous concurrent accesses from other threads. After a grace period, GC threads sweep and collect the *logically* deleted items.

For example, epoch-based GC [63] is a commonly used strategy for lazy GC [60, 61, 64]. However, a volatile epoch-based GC implementation may incur memory leaks in NVM because it does not persist the metadata of labeled garbage data. After restart, these garbage data will be unreachable.

A naïve way to address this problem is to persist the metadata of labeled garbage data for every metadata modification. We compare the volatile and naïve persistent epoch-based GC in FAST&FAIR, which supports lock-free lookup operations and needs a GC mechanism. As shown in Figure 4, we see that the performance of persistent epoch-based GC is 25.7% worse than the volatile GC.

[1] *not open-sourced* means that we are not sure what it uses. *no need* means that it does not need to consider this factor.

Table 3: Optimizations on practical aspects in ROART.

	ROART Design
Functionality	*Leaf Compaction* (§3.2) to optimize range queries.
Performance	*Entry Compression* (§3.3) to use only one persistent instruction for structural information.
	Selective Metadata Persistence (§3.3.1) to reduce the amount of persisted metadata.
	Minimally Ordered Split (§3.3.2) to reduce `sfence` instructions in internal node split.
Correctness	`non-temproal store` to resolve anomaly (§3.4.1).
	DCMM (§3.4.2) to prevent memory leaks with minimal persistence during allocation. *Instant Restart* to eliminate the long recovery time for DCMM.

Post-crash GC [22, 54–56] reduces the persistence overhead during normal operations. We test the recovery time of Makalu [54] with post-crash GC in Figure 5, and find that the recovery time increases linearly as the amount of data increases.

We would like to design a GC solution that neither incurs the persistence overhead in the naïve epoch-based GC nor suffers long recovery time of the post-crash GC.

In summary, Table 2 compares the functionality and correctness features of recent studies. We find that most studies do not consider all these aspects (BzTree takes these into account, but it suffers heavy persistence overhead [8]). Thus, there are still a lot of opportunities for improvement.

3 Design of ROART

Based on the discussion and analysis above, we propose a persistent index, ROART, which takes the three aspects into account. Table 3 summarizes the distinct features of ROART.

3.1 Radix Tree and its Persistent Variants

A radix tree is a search tree in which each node represents a chunk of bits in the key. The key in a leaf node equals to the string constructed along the path, starting from the root to the corresponding leaf node. Suppose one node stores s bits of a key. The node has at most $r = 2^s$ children. r is called the *radix* of the tree. ART [49, 50] is a space-efficient radix tree. Its *radix* is 256 and each node represents a 1-byte character (8-bit, $r = 256 = 2^s$, $s = 8$) of the key. We will discuss its node types in §3.3.

Path Compression. The height of the radix tree can be reduced by path compression [49], as illustrated in Figure 6. A node with only one child is merged into its child, and the character it represents is merged into the prefix of its child.

Node Split. With path compression, node splits may happen during insertions. Node splits are divided into two categories, i.e., internal node split and leaf node split, as shown in Figure 7. An internal node split occurs when a new insertion (e.g., $L3$) mismatches the prefix of an internal node (*old*). The node *new* is created and inserted into the tree, which points to $L3$ and *old* as its children. The prefix of *old* will be updated. A leaf node split occurs when a new insertion ($L3$) mismatches the key in a leaf node ($L2$). A new node pointing to both $L2$ and $L3$ will be created and inserted into the tree.

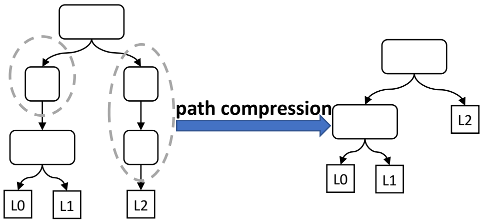


Figure 6: Path compression in radix tree.

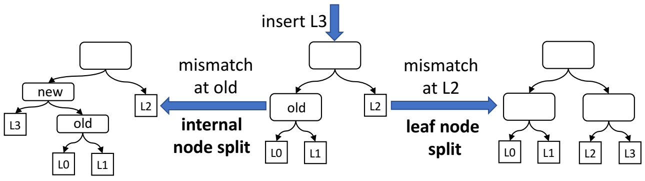


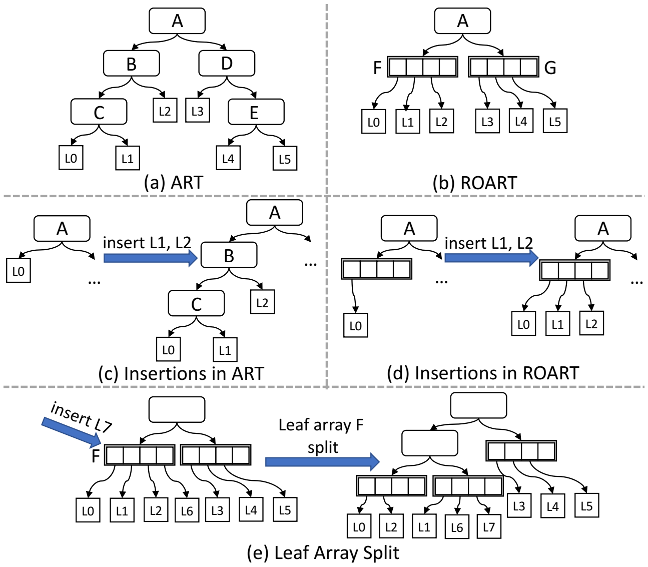
Figure 7: Two ways of node split.

Persistent Variants. Because of ART's efficiency, most persistent radix trees are based on ART. The ART implementation in WORT [19] supports only single thread, and uses a separate slot array and a bitmap in its node to help locate entries. However, this incurs extra persistence overhead. RECIPE [24] proposes a method to convert any volatile data structure to its persistent counterpart. Based on this method, P-ART is directly converted from the volatile ART-ROWEX [50]. However, this leaves many opportunities for persistence optimizations. Neither WORT nor P-ART has optimized range queries.

3.2 Our Solution: ROART Structure

Compared to B$^+$-Trees, the radix tree performs poorly in range queries because each leaf node stores only a pair of key and value, and leaf nodes can be on many different levels in the tree. A range scan has to visit a large number of non-leaf nodes on different levels in addition to leaf nodes, incurring significant overhead. We propose *leaf compaction* (LC) to compact the pointers of leaf nodes into a *leaf array* in the radix tree. A leaf array can contain up to m leaf pointers. If a subtree of the radix tree has less than or equal to m leaf nodes, the subtree is compacted into a leaf array. (We set $m = 64$ in our implementation.) For simplicity of presentation, the figures in this subsection use $m = 4$.

Figure 8(a) and 8(b) show the structural differences between ART and ROART with the same leaf nodes. We see that the subtrees rooted at B and D are compacted into leaf array F and G, respectively. For range queries, *leaf compaction* can effectively reduce the number of pointer chasing in the different levels of the tree. For instance, to run a range query covering $L0-L5$, ART dereferences 15 pointers $(A-B-C-L0-L1-C-B-L2-B-A-D-L3-E-$



(a) ART

(b) ROART

(c) Insertions in ART

(d) Insertions in ROART

(e) Leaf Array Split

Figure 8: Leaf compaction in ROART.

$L4 - L5$), while ROART requires only 11 pointer dereferences ($A - F - L0 - L1 - L2 - F - A - G - L3 - L4 - L5$).

We modify the index operations to support leaf compaction. For lookup, a reader searches the tree as before until it reaches a leaf array. It has to check each leaf node that the leaf array points to, which can be costly. To minimize this effect, we embed a 16-bit fingerprint (hash value) [17, 18] of each leaf key into the pointer in the leaf array (current architectures support only 48-bit addresses), represented as fingerprint: 16-bit | address: 48-bit. In this way, the reader compares the fingerprint of the search key with the fingerprints in the leaf array to filter out most unnecessary cases. The probability of false positives is low (e.g., < 0.001 for $m = 64$).

For insert, when it reaches a leaf array, a writer checks to see if the key already exists. If not, the writer chooses an empty slot to insert, as shown in Figure 8(d). The complex case is when the leaf array is full and the leaf array splits, as shown in Figure 8(e). Here, the writer wants to insert a new leaf $L7$ into leaf array F, which is already full. Note that all keys in F correspond to a subtree in the original radix tree, and therefore share a common prefix. To split, we need to find the first byte position, denoted as P, where the keys diverge. We call the P-th byte as the identifying byte. We divide the keys into subsets based on their identifying bytes. We build a leaf array for each subset, and create a new internal node, which contains each identifying byte and the pointer to the associated leaf array. For example, in Figure 9, a leaf array with four leaf pointers is to split into three new leaf arrays. The keys diverge at the 5th byte, i.e. $P = 5$. The keys are divided, and then new leaf arrays are created. Note that the cost of a leaf array split is high, but fortunately, this is a rare operation.

For update and delete, the procedure is similar to lookup. After the matching key is found, the corresponding modification or deletion is performed.

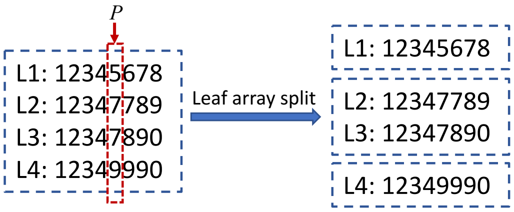


Figure 9: An example of leaf array split.

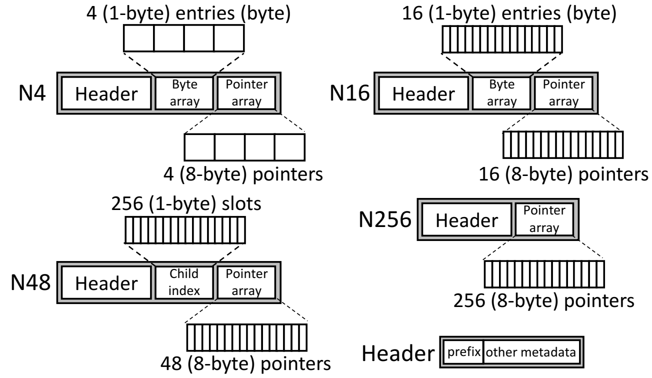


Figure 10: Node types in ART.

For range query, the only difference with ART is that keys in a leaf array are not sorted, but keys are ordered between leaf arrays. Thus, we only need to check/sort the begin and the last leaf arrays to ensure that the return values are within the requested range. If values need to be fully sorted before returned, it will bring about 8.9% performance degradation with some optimizations, e.g., by skipping the prefix of keys and comparing only the different parts of keys.

Interestingly, *leaf compaction* can improve not only range queries but also traversals and insertions. Traversal is an essential step in all operations (lookup/insert/update/delete/scan). *Leaf compaction* tends to reduce the root-to-leaf path lengths, as can be clearly seen in Figure 8(a) and (b). Shorter path lengths are beneficial to traversals. Moreover, when a new insertion mismatches the key in a leaf node, ART incurs a leaf node split (Figure 7) while ROART simply inserts the new leaf in the leaf array, reducing the number of persistent instructions. (Please see Table 4 for detailed counts.)

In summary, *leaf compaction* has several benefits: (i) decreasing the number of pointer chasing for range queries, (ii) reducing root-to-leaf path lengths and traversal overhead, (iii) reducing persistence overhead of insertions. We will evaluate the benefits of this structure in §4.

3.3 Reducing Persistence Overhead

Figure 11 shows the node structures of ROART, which are based on the design of ART. Figure 10 shows the four types of nodes in ART that can store up to 4, 16, 48, and 256 entries, respectively. Each entry contains a `byte` and a child pointer. The `byte` is equal to the character represented by the corre-

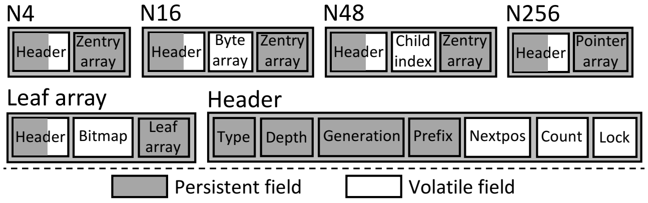


Figure 11: Node structures in ROART.

sponding child node. In N4 and N16, `bytes` and pointers are stored in `Byte array` and `Pointer array`, respectively. In N48, `Child index` has 256 slots so that `bytes` can be used as an index to search in `Child index` to find the location of the corresponding child pointer, because the range of a `byte` is 0-255. N256 directly has an array with 256 pointers. A node expands to a larger node type when it is full and a new entry is to be inserted, and shrinks to a smaller node type when the number of entries is below a threshold.

In ROART, we propose *entry compression* (EC) to pack the key byte into the pointer (where 48-bit are used) in N4, N16, and N48. The resulting entry | empty: 8-bit | key: 8-bit | pointer: 48-bit | is called `Zentry`, as shown in Figure 11. An invalid `Zentry` (for a deleted or unused slot) is set to 0. Since a `Zentry` is 8-byte and can be updated atomically, compared to ART, EC reduces one persistent instruction for persisting each entry in ROART.

3.3.1 Selective Metadata Persistence

We observe that not all metadata need to be persisted for correctness. For example, `Nextpos` and `Count` in the header indicate the next empty entry slot and the number of used slots. The `Bitmap` in the leaf array shows which leaf array entry is in use. They can all be computed by scanning the Zentry or the pointer array, where empty slots are set to 0. Moreover, the `Byte array` in N16 is used to accelerate search with SIMD instructions. It can be rebuilt by retrieving the embedded key byte from each Zentry. The `Child index` in N48 can be restored in the same way. Finally, `Lock` is used for concurrency control and can be cleared upon crash recovery.

Based on this observation, we propose *selective metadata persistence* (SMP) to selectively persist a subset of the metadata and recompute the rest of the metadata after recovery. As shown in Figure 11, volatile metadata are highlighted with white background. Fields with grey background are persisted.

Traditional recovery needs to suspend processing requests and scan the whole indexes, which incurs long recovery time as the amount of data increases. Inspired by the generation lock in NV-Heaps [65], we implement *selective metadata persistence* using generation numbers to hide the recovery overhead. ROART maintains a global generation number (GGN) in NVM. GGN is increased upon each restart. Each node in ROART has its own persistent node generation number (NGN). When accessed, if NGN equals to GGN, the metadata in the node is up-to-date. Otherwise, the metadata in the node is restored, then GGN is assigned to NGN. Per-node latch for recovery (implemented by using flags in memory and CAS instructions) protects the concurrent access on the same node from multiple threads. In this way, after restart, ROART does not suspend normal operations for recovering the whole lost metadata. Instead, it restores the metadata on demand and at the same time executes normal operations.

Table 4: Persistence analysis. Two values are the numbers of `clwb` and `sfence` respectively (lower is better).

Name	Insert				Split	
	N4	N16	N48	N256	Leaf	Internal
ROART	2, 2	2, 2	2, 2	2, 2	2, 2	4, 2
P-ART	2, 2	3, 3	3, 3	2, 2	3, 3	4, 4
WORT	3, 3	4, 4	3, 3	2, 2	3, 3	4, 4

3.3.2 Minimally Ordered Split

Internal node split is costly as shown in Figure 7. It has four steps: (i) allocating a new leaf $L3$, (ii) allocating an internal node, marked as *new*, with two children ($L3$ and node *old*), (iii) changing the pointer (from *old* to *new*) of parent node, (iv) updating the prefix of *old* (not shown in the figure). Without optimizations, the four steps need four `sfence` instructions.

We observe that the order of these four steps can be relaxed. Step (i) and step (ii) are not visible to other threads, we can use only one `sfence` after initializing the two nodes. Step (iii) and (iv) cannot execute atomically. Under concurrent execution, readers may see the incomplete split with inconsistent prefixes. Note that the depth of a node (including the prefix) stays constant. This property can be exploited to detect inconsistent prefixes, as is also used in other work [19, 50]. Once such inconsistency is detected, it is easy to repair the inconsistency by recomputing the prefix. Consequently, the order of step (iii) and step (iv) is not important.

ROART performs the internal split as follows. It performs step (i), (ii) and (iv), flushes the modified cache lines, then calls a single `sfence`. After that, it performs step (iii), flushes the modified cache line, and calls a second `sfence`. In this way, ROART reduces the numbrer of `sfence` instructions of an internal split from four to two.

3.3.3 Persistence Analysis

Table 4 compares the number of persistence instructions, `clwb` and `sfence`, for insert and split operations in ROART, P-ART [24], and WORT [19]. ROART incurs the smallest number of persistence instructions among the three indexes.

3.4 Making ROART Correct

3.4.1 Anomaly Resolution in ROART

ROART employs a concurrency control strategy similar to ART-ROWEX [50], i.e., lock-free read and lock-based write.

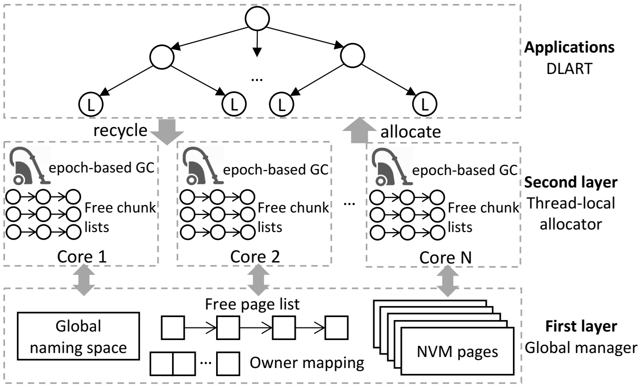


Figure 12: DCMM architecture

Readers may see incomplete write operations, resulting from inconsistency between volatile and persistent states. So extra protection is required. §2.3.1 discusses several methods to address this problem. We adopt `non-temporal store` [24] to fix the potential anomalies in ROART.

3.4.2 Delayed Check Memory Management

NVM allocation and GC have a large performance impact on practical indexes (§2.3.2). We propose a new memory management method, called DCMM, which uses post-crash GC to minimize the persistence during allocation/deallocation, and supports *instant restart* to eliminate the waiting time after restart. To reduce the contention in memory allocation for multiple threads, DCMM uses a two-layer architecture, as in Figure 12.

First Layer. This layer is a global memory manager that manages the entire NVM area at the granularity of pages. The page size is adjustable, and defaults to 128MB [53, 56]. The global memory manager maintains a global naming space. It contains the roots of indexes and an `offset` field indicating the offset of the last allocated page. These are persistent fields. There are also two volatile fields, i.e., `owner_mapping` and `free_page_list`. The former keeps a map between each page and its owner thread, and the latter implements a volatile lock-free linked list for recycled free pages. A thread requests a new page by searching `free_page_list`. If `free_page_list` is empty, it uses an atomic `fetch-and-add` to obtain the `offset` and increase the `offset` by the page size, then persists the `offset`.

Second Layer. For each application thread, a thread-local allocator performs allocation using multiple block classes with different sizes. Each block class is maintained by a volatile linked-list, called `free_chunk_list`. A thread requests a page from the first layer and divides it into `free_chunk_lists`, like in the buddy system [66].

Garbage Collection. DCMM implements post-crash epoch-based GC for supporting lock-free reads and lazy deletions. We implement a decentralized version [60] of epoch-based GC for better scalability.

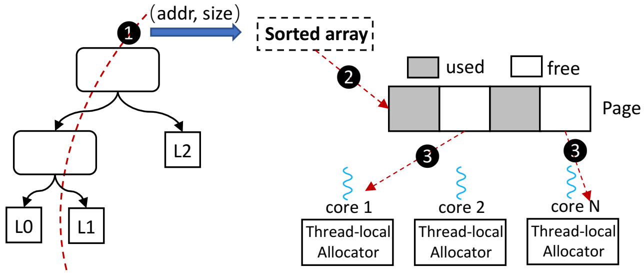


Figure 13: The procedure of recovery.

Persistence Overhead. As discussed in §2.3.2, the allocator based on post-crash GC does not need to persist any metadata during normal operations. Therefore, persistence overhead only occurs in the first layer, for persisting `offset`. This overhead is amortized by multiple memory allocations in the second layer. Most allocations do not invoke the first layer.

Recovery Processing. Upon recovery, `owner_mapping` can be simply reset by mapping each page in the range [0, `offset`) to application threads in the round-robin fashion. Other volatile information can be restored by the recovery process in three steps (Figure 13) : (i) Recovery threads traverse all NVM areas used by the application (i.e., starting from the persistent index root pointers and traversing the trees), and collect all used chunks with their description tuples (address, size). (ii) Free chunks can be calculated based on the used chunks in each page. (iii) Free chunks are put in the `free_chunk_lists` and free pages are put in the `free_page_list`.

Instant Restart. The recovery process as described in the above can take a long time as the amount of data increases (Figure 5). In order to reduce the waiting time after restart, we propose *instant restart* for DCMM. Note that `offset` in the first layer is persisted. Therefore, after restart, we can immediately allocate new pages after `offset` without waiting for other metadata recovery to complete. (If `offset` exceeds the current NVM file limit, a new NVM file will be created and opened for allocation [53, 56].) Hence, we can immediately allow front-end operations and provide memory allocation service instantly after restart, while the background recovery threads run in parallel. In this way, DCMM avoids the front-end from waiting long time for the recovery to complete.

Multi-threading Optimization. The background recovery process can be accelerated by multi-threading. Consider the three steps in recovery processing. Step (i) can be parallelized based on the data structures. For example, multiple threads can be used to traverse different subtrees of ROART. The (address, size) pairs produced during traversal can be put into different sorted arrays based on address ranges. Then, Step (ii) and (iii) can examine different sorted arrays and collect free chunks in parallel.

4 Evaluation

Our evaluations consist of four parts to reflect the performance improvements of each proposed design.

1. Overall Performance Comparison. We choose several tree-based data structures in experiments. For a fair comparison, some modifications are necessary such as adding DCMM to some indexes, and implement missing functions.

2. Detailed Test of Each Design. Several aspects are evaluated: (i) performance improvement by each optimization, (ii) range query (scan) performance with different numbers of keys, (iii) fixed-sized (8-byte) keys performance, (iv) skew tests, (v) latency tests, (vi) space consumption, and (vii) recovery and instant restart.

3. NVM Allocators. We evaluate DCMM with several open-sourced persistent allocators, e.g., PMDK [51], nvm_malloc [52] and Makalu [54].

4. Real-world System Evaluation. We incorporate ROART into a real-world system: Memcached [67]. The core index in Memcached is a volatile hash index and our modification enables Memcached to support persistent storage.

4.1 Evaluation Setup

All evaluations use a Dell PowerEdge R740 server with four Intel(R) Xeon(R) Gold 5220 processors supporting `clwb`, 6×128GB Optane DC PMM per socket. The processor has 32KB L1-cache, 1MB L2-cache, and 25MB L3-cache. The persistent memory is managed by a DAX file system [68] and mapped to a pre-defined address. We choose five other tree-based indexes to compare the performance with ROART.

P-ART. P-ART [24] is a persistent counterpart of ART-ROWEX [50]. For a fair comparison, we use DCMM in P-ART, and implement its missing functions (e.g., update operations, and *selective metadata persistence* for metadata).

PMwCAS-ART. PMwCAS-ART is a baseline we implement by using PMwCAS [23]. PMwCAS allows atomically modifying multiple 8-byte words in NVM and uses PMDK to guarantee the memory safety. We leverage the persistent primitives it provides to modify ART-ROWEX.

FAST&FAIR-DCMM. FAST&FAIR [5] is a state-of-the-art persistent B$^+$-Tree (§2). We modify it to support variable-sized keys and use DCMM to manage NVM.

SkipList-DCMM. We modify the open-source lock-free SkipList [22] to support variable-sized keys and use DCMM.

BzTree. BzTree [6] is a lock-free persistent B$^+$-Tree based on PMwCAS. It can naturally support variable-sized keys by using slotted pages.

4.2 Overall Performance

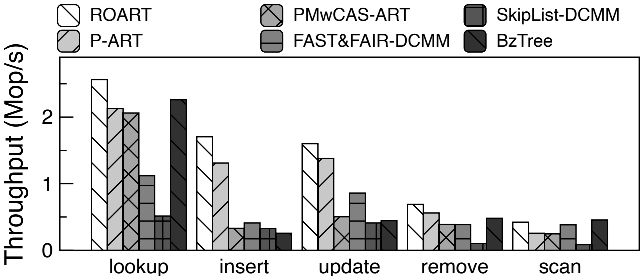


Figure 14: Microbench of six indexes under 4 threads.

To evaluate the overall performance, we test micro-benchmarks with 4 threads and YCSB benchmark. For the micro-benchmarks, keys are randomly generated with sizes between 4 to 128 bytes, and values are fixed as 8 bytes. An 8-byte value can represent an indirect pointer, commonly used in DBMS [32,33]. Each test firstly warms up using 30 million KVs [5,7,24], which exceeds the size of the L3-cache and reflects the performance of NVM. After warming up, each test runs 20 seconds for different workloads and reports the average throughput.

Micro-benchmarks contain the operations of lookup, insert, update, remove and scan. The results are in Figure 14. The lookup performance of ROART is 2.562 Mop/s which is faster than P-ART (2.129 Mop/s) and PMwCAS-ART (2.06 Mop/s). The main improvement comes from *leaf compaction* which lowers the height of the tree and benefits traversal. BzTree is fast because its slotted-page node layout has good cache locality for supporting variable-sized KVs and binary search. ROART is 2.29× and 4.98× faster than FAST&FAIR and SkipList respectively, because FAST&FAIR cannot use binary search inside its nodes and SkipList suffers from poor cache locality.

The insert performance of ROART is significantly better (1.704 Mop/s) than all other five indexes (1.3/5.16/4.15/5.24/6.65×). There are several reasons for the improvement. (i) DCMM has higher allocation performance than PMwCAS (PMDK). (ii) Less persistent related instructions (`clwb` and `fence`) in ROART (§3). PMwCAS suffers from more persistent instructions, P-ART makes no optimization, and FAST&FAIR also causes multiple persistent instructions once the entry moving exceeds one cache line. (iii) Compared with B$^+$-Tree (FAST&FAIR, BzTree), no rebalance operations are required in ROART.

For the update operation, ROART can achieve 1.6 Mop/s throughput and outperforms the others up to 1.16/3.18/1.86/3.9/3.61×. The major performance differences are similar to lookup operation. For the remove operation, SkipList performs very poor, and others are similar. The main reason is that SkipList has a complicated remove operation and suffers from many retries. For the scan operation, with *leaf compaction*, the performance of ROART can outperform P-ART up to 1.65× and is close to FAST&FAIR/BzTree.

We use YCSB [69] benchmark to generate five workloads, which are (a) write-intensive (50% lookup and 50% insert), (b) read-intensive (95% lookup and 5% insert), (c) read-only, (d) insert-only, and (e) scan-insert (95% scan and 5% insert).

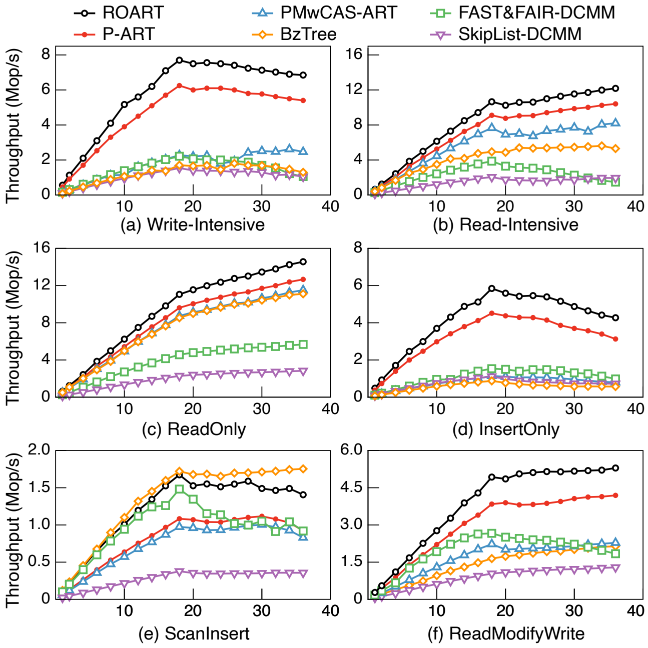


Figure 15: Performance of YCSB.

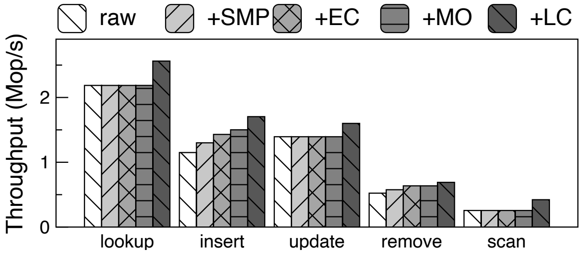


Figure 16: Performance improvement of each optimization. (SMP: Selective Metadata Persistence, EC: Entry Compression, MO: Minimally Ordered split, LC: Leaf Compaction)

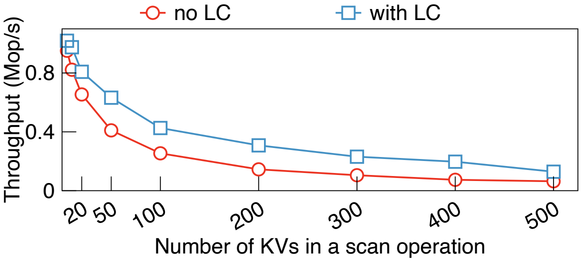


Figure 17: Range queries with different key numbers.

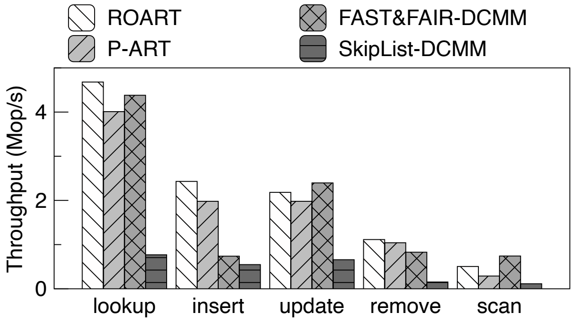


Figure 18: Performance with fixed-sized keys.

The results of five workloads are shown in Figure 15.

In workload (a), ROART outperforms P-ART up to 1.27× and other four indexes up to 2.78∼6.57× using 36 threads. The main performance gain is from its less traversal and persistence. In workload (b), ROART outperforms the other five indexes by 1.17∼8.27× using 36 threads. In workload (c), the performance of all indexes is scaled, but ROART can still outperform others by 1.15∼5.13×. In workload (d), due to the influence of NUMA, performance of all indexes begins to decline after more than 18 threads. ROART decreases 27% and P-ART decreases 30% from 18 threads to 36 threads. In workload (e), ROART can perform 1.52× and 1.53× better than P-ART and FAST&FAIR. It is only 3% and 20% slower than lock-free BzTree in the cases of 18 and 36 threads, because BzTree stores variable-sized keys and values in the nodes, instead of extra data areas, located by metadata in the head of nodes.

4.3 Effects of Each Design

1. Improvement of Each Optimization in ROART. We test the performance improvement of each optimization (§3) in Figure 16. The raw version is the implementation of ROART without the four optimizations (SMP/EC/MO/LC). *Selective metadata persistence* can improve by 13% and 10.1% for insertion and deletion. *Entry compression* can bring about 9.7% and 10.8% improvement for insert and remove operations. *Minimally ordered split* reduces the `fence` instructions for internal node split so that it can improve the insert performance by 4.8%. *Leaf compaction* can lower the height of radix tree and benifit every operation, especially scan. It can bring 17.2%/13.5%/14.7%/8.4%/64.8% improvement for the five operations respectively.

2. Range Queries with Different Key Numbers. In Figure 17, we evaluate the range query performance by scan operations with different key numbers, and the necessary parameters of scan are the maximum and minimum keys as well as the number of required keys. The result shows that ROART with LC can outperform the version without LC by 1.07∼2.01×. When the number of keys is less than 50, the improvement brought by LC is not very much, and when the number of keys is more than 100, the performance is improved at least by 1.65×.

3. Performance with Fixed-sized Keys. Many indexes are optimized for fixed-sized KV, such as FAST&FAIR. We test the performance of ROART (without any optimization for fixed-sized keys) while processing 8-byte fixed-sized keys, compared to P-ART, FAST&FAIR and SkipList. The results are shown in Figure 18. SkipList runs slowest because of its poor cache locality. FAST&FAIR outperforms P-ART by up to 1.09/1.21× in lookup and update because fixed-sized

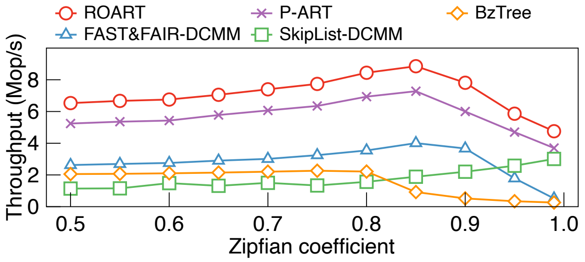


Figure 19: Performance with different zipfian.

Table 5: Latency tests under write-intensive workload (50% lookup and 50% insert) with 16 threads (lower is better).

latency (us)	ROART	P-ART	PMwCAS-ART
avg.	1.2	1.5	3.4
p99	3.5	4.4	8.8
latency (us)	FAST&FAIR	SkipList	BzTree
avg.	3.5	4.4	4.2
p99	11.8	8.9	9.5

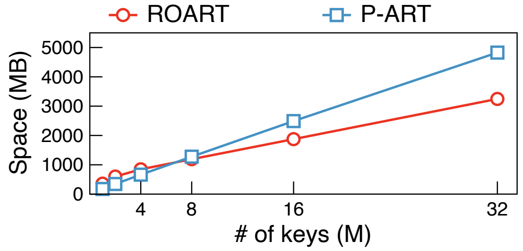


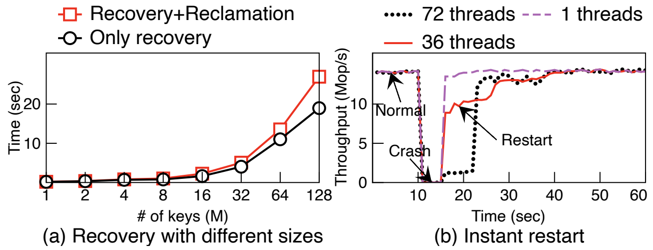
Figure 20: Space consumption of ROART and P-ART

optimized FAST&FAIR has better cache locality. But with the optimization LC, ROART can outperform FAST&FAIR in lookup even with fixed-sized keys, and it is only sightly slower than FAST&FAIR in update. For scan with fixed-sized keys, B$^+$-Tree is still the better index than radix tree.

4. Skew Tests. Figure 19 shows the experiment under a skewed workload (50% lookup and 50% update with 16 threads). The cache brings more benefits when the coefficient is smaller than 0.85. When larger than 0.85, ROART, P-ART and FAST&FAIR all suffer from the lock contention. The performance of ROART and P-ART drops about 28% and 29.5% from 0.5 to 0.99, while FAST&FAIR drops about 80.3%. Performance of SkipList improves because of its lock-free manner. BzTree drops about 87% because it has a complex structure and heavy persistence overhead [8] although it also has a non-blocking design.

5. Latency Test. Latency numbers of each index are shown in Table 5 under a write-intensive workload (50% lookup and 50% insert) with 16 threads. In average latency, ROART can outperform all other indexes by 20% ∼ 73% because of its faster traversal. In p99 latency, ROART can outperform all other indexes by 21% ∼ 71%. SkipList and BzTree is lock-free design so that their p99 latencies increase less than other four lock-based indexes. In ROART, we make no extra optimization on tail latency, which is orthogonal to our work.

6. Space Consumption. We introduce leaf arrays in ROART, which does not exist in ART. In our implementation, the size of leaf array is predefined, which may cause the waste of space. So we evaluate the space consumption of ROART and P-ART to show the impact of leaf arrays. In Figure 20, when the amount of keys is smaller than 8M, the space consumption of ROART is larger than P-ART because most of the entries in leaf arrays are empty and much space is wasted. As the amount of keys increases and empty entries are filled, ROART consumes less space than P-ART. Under extreme cases, if each leaf array only has one valid entry, the space waste of ROART will become serious. We think this case is rare because it is hard to construct. To solve this problem, we can use the approach of ART to provide leaf arrays with various sizes.



(a) Recovery with different sizes (b) Instant restart

Figure 21: Data structure recovery and instant restart.

7. Recovery and Instant Restart. In Figure 21(a), we test normal recovery time with different key numbers. With 128M keys (about 25 GB in total of tree size), data structure recovery takes 19 seconds. With reclamation of free memory (recovery for DCMM), it takes 27 seconds. In this case, free memory chunks in 195 pages (25 GB in total) are reclaimed.

All metadata of data structure and allocator will be restored after restart. So we introduce *selective metadata persistence* (§3.3.1) and *instant restart* (§3.4.2) to hide the recovery overhead of data structure and allocator respectively. The effects of the two techniques are illustrated in Figure 21(b). The test uses 32M keys and 1/36/72 background reclamation threads respectively. The simulation injects a crash at the 11th second, halting for 5 seconds. After restart, ROART can process requests immediately. The recovery of data structure can be delayed until nodes are accessed. Background threads perform the reclamation of free memory chunks in parallel with foreground threads. Experiments show that more threads accelerate recovery process, but causing a greater impact on the foreground performance.

4.4 Performance of NVM Allocators

We make a comparison in Figure 22 between DCMM and other open-sourced persistent allocators, e.g., PMDK [51], nvm_malloc [52] and Makalu [54]. The test workload is to continuously allocate 64-byte chunks, write and persist them. The results show the performance of each thread. PMDK is slow but scalable, the scalability of nvm_malloc and Makalu is poor. DCMM has better performance and scalability than

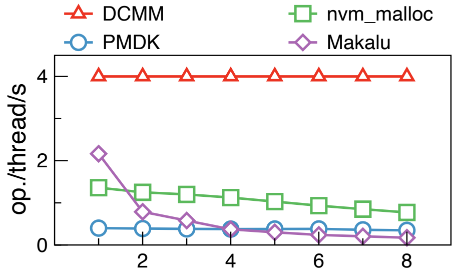


Figure 22: Performance with different allocators.

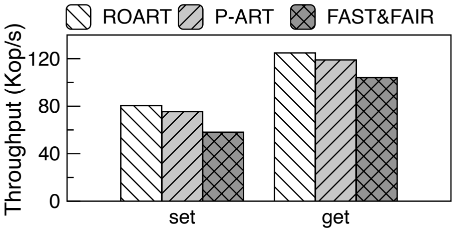


Figure 23: Evaluations in Memcached.

Makalu because DCMM allocates larger pages in a lock-free manner in the first layer, while the page size is only 4K in Makalu and the design in its first layer is lock-based.

There are some other persistent allocators. PAllocator [53] is logging-based with good scalability, but it also suffers extra persistence during allocation/deallocation. PMDK [70] also has a post-crash GC technique, but it is still logging-based. NV-Epochs [22] provides post-crash GC but only supports fixed-sized allocation, and suffers long recovery time. NVM-Reconstruction [55] is a Clang/LLVM extension and runtime library that provides the reconstruction of persistent heaps. Ralloc [56] improves the performance of allocators with post-crash GC, but it still needs a blocked recovery process.

4.5 Real-World System Evaluation

We modify Memcached 1.4.17 to replace its hash index to three persistent indexes, e.g., ROART, P-ART and FAST&FAIR, for persistent storage. We use `memtier_benchmark` to test the performance of set and get operations with single thread. In Figure 23, ROART can outperform P-ART and FAST&FAIR by up to 1.07× and 1.38× in set operations, 1.06× and 1.19× in get operations. The evaluation confirms our previous experiments.

5 Related Works

ROART well resolves the three practical aspects mentioned in §2. Many other related works not mentioned before have also made a lot of efforts.

Persistent Tree-based Indexes. CDDS Tree [1] firstly proposes a multi-version persistent B+Tree design, using copy-on-write techniques without overwriting the original entry, but suffers heavy persistence overhead. DPTree [12] proposes a method to batch modifications in DRAM buffer to reduce persistent overhead, but it needs a background merging process which may stall foreground requests and consume extra bandwidth of NVM. μtree [13] focuses on tail latency in persistent indexes, which is an orthogonal work.

Persistent Hash Indexes. Persistent hash indexes can support fast point access, but has difficulties for range queries. Level hashing [15] proposes a novel two-level hash table structure, reducing the overhead of resizing. Clevel hashing [18] is the multi-thread version of level hashing, based on a lock-free manner and concurrent resize operation in the background. CCEH [16] proposes a three-layer structure based on extendible hashing to reduce NVM writes. Dash [17] uses optimistic concurrent control to improve the parallelism of CCEH, and proposes bucket load balancing strategy to improve load factor of hash table.

Universal Conversion. The general method usually gives a simpler solution to achieve persistent indexes, but less optimization for performance. Izraelevitz et al. [21] design an approach transforming any non-blocking transient data structure to a non-blocking durable one, by adding flush and fence instructions after read or store instructions, but suffering heavy persistence overhead. David et al. [22] propose a `link-and-persist` method to implement log-free concurrent data structures and guarantee durable linearizability. But it can only be applied to 8-byte store/CAS instructions.

6 Conclusion

This paper firstly analyzes three practical aspects, including functionality, performance and correctness. Then ROART is proposed with several optimizations, i.e., (i) *leaf compaction*, (ii) *entry compression*, (iii) *selective metadata persistence*, (iv) *minimally ordered split*, and (v) *instant restart*. Finally, evaluations indicate that ROART can outperform other state-of-the-art indexes by 1.17∼8.27× under various workloads.

Acknowledgments

We are grateful to our shepherd, Vijay Chidambaram, and the anonymous reviewers for their constructive comments and suggestions. This work is supported by National Key Research & Development Program of China (2016YFB1000504), Natural Science Foundation of China (61877035, 61433008, 61373145, 61572280).

References

[1] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *FAST*, volume 11, pages 61–75, 2011.

[2] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[3] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: reducing consistency cost for NVM-based single level systems. In *13th USENIX Conference on File and Storage Technologies FAST 15)*, pages 167–181, 2015.

[4] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386. ACM, 2016.

[5] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, 2018.

[6] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. BzTree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

[7] Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. Building Scalable NVM-based B+ tree with HTM. In *Proceedings of the 48th International Conference on Parallel Processing*, page 101. ACM, 2019.

[8] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment*, 13(4):574–587, 2019.

[9] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+ trees: optimizing persistent index performance on 3dx-point memory. *Proceedings of the VLDB Endowment*, 13(7):1078–1090, 2020.

[10] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. clfB-tree: Cacheline Friendly Persistent B-tree for NVRAM. *ACM Transactions on Storage (TOS)*, 14(1):5, 2018.

[11] Ping Chi, Wang-Chien Lee, and Yuan Xie. Making B+-tree efficient in PCM-based main memory. In *Proceedings of the 2014 international symposium on Low power electronics and design*, pages 69–74. ACM, 2014.

[12] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. Dptree: differential indexing for persistent memory. *Proceedings of the VLDB Endowment*, 13(4):421–434, 2019.

[13] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: a persistent b+-tree with low tail latency. *Proceedings of the VLDB Endowment*, 13(11).

[14] Pengfei Zuo and Yu Hua. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST)*, pages 1–10, 2017.

[15] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.

[16] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, 2019.

[17] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*, 2020.

[18] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812, 2020.

[19] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, 2017.

[20] Wen Pan, Tao Xie, and Xiaojia Song. Hart: A concurrent hash-assisted radix tree for dram-pm hybrid memory systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 921–931. IEEE, 2019.

[21] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*, pages 313–327. Springer, 2016.

[22] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 373–386, 2018.

[23] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472. IEEE, 2018.

[24] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 462–477. ACM, 2019.

[25] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, 2017.

[26] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, page 4. ACM, 2015.

[27] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *ACM SIGPLAN Notices*, volume 53, pages 28–40. ACM, 2018.

[28] Hyungjun Oh, Bongki Cho, Changdae Kim, Heejin Park, and Jiwon Seo. Anifilter: parallel and failure-atomic cuckoo filter for non-volatile memories. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[29] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 329–343. ACM, 2017.

[30] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric nvm architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 757–773, 2020.

[31] Intel Optane DC Persistent Memory Module. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[32] MySQL. https://www.mysql.com/.

[33] PostgreSQL. https://www.postgresql.org/.

[34] Peloton. https://db.cs.cmu.edu/projects/peloton/.

[35] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013.

[36] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

[37] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 677–689, 2015.

[38] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1675–1687, 2016.

[39] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment*, 10(2):49–60, 2016.

[40] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35, 2017.

[41] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642, 2016.

[42] RocksDB. https://rocksdb.org/.

[43] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1077–1091, 2020.

[44] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[45] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling low tail latency on multicore key-value stores. *Proceedings of the VLDB Endowment*, 13(7):1091–1104, 2020.

[46] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13, 2011.

[47] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[48] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, 2020.

[49] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, volume 13, pages 38–49, 2013.

[50] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, page 3. ACM, 2016.

[51] PMDK. https://pmem.io/.

[52] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory Allocation for NVRAM. *ADMS@ VLDB*, 15:61–72, 2015.

[53] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory management techniques for large-scale persistent-main-memory systems. *Proceedings of the VLDB Endowment*, 10(11):1166–1177, 2017.

[54] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *ACM SIGPLAN Notices*, volume 51, pages 677–694. ACM, 2016.

[55] Nachshon Cohen, David T Aksun, and James R Larus. Object-oriented recovery for non-volatile memory. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–22, 2018.

[56] Wentao Cai, Haosen Wen, H Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L Scott. Understanding and optimizing persistent memory allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, pages 60–73, 2020.

[57] Zhaoguo Wang, Hao Qian, Haibo Chen, and Jinyang Li. Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, pages 1–7, 2013.

[58] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[59] libvmmalloc. https://pmem.io/vmem/libvmmalloc/.

[60] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, pages 473–488. ACM, 2018.

[61] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. HOT: a height optimized Trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 521–534. ACM, 2018.

[62] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196. ACM, 2012.

[63] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[64] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

[65] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 47(4):105–118, 2012.

[66] James L Peterson and Theodore A Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[67] Memcached. https://http://memcached.org/.

[68] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42:34–40, 2017.

[69] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[70] pmemobj. https://pmem.io/pmdk/manpages/linux/master/libpmemobj/pmemobj_first.3.

SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage

Hao Chen[†‡] Chaoyi Ruan[†] Cheng Li[†*] Xiaosong Ma[‡] Yinlong Xu[†ζ]

[†]*University of Science and Technology of China*
[‡]*Qatar Computing Research Institute, HBKU*
[ζ]*Anhui Province Key Laboratory of High Performance Computing*

Abstract

Key-Value (KV) stores support many crucial applications and services. They perform fast in-memory processing, but are still often limited by I/O performance. The recent emergence of high-speed commodity NVMe SSDs has propelled new KV system designs that take advantage of their ultra-low latency and high bandwidth. Meanwhile, to switch to entirely new data layouts and scale up entire databases to high-end SSDs requires considerable investment.

As a compromise, we propose SpanDB, an LSM-tree-based KV store that adapts the popular RocksDB system to utilize *selective deployment of high-speed SSDs*. SpanDB allows users to host the bulk of their data on cheaper and larger SSDs, while relocating write-ahead logs (WAL) and the top levels of the LSM-tree to a much smaller and faster NVMe SSD. To better utilize this fast disk, SpanDB provides high-speed, parallel WAL writes via SPDK, and enables asynchronous request processing to mitigate inter-thread synchronization overhead and work efficiently with polling-based I/O. Our evaluation shows that SpanDB simultaneously improves RocksDB's throughput by up to 8.8× and reduces its latency by 9.5-58.3%. Compared with KVell, a system designed for high-end SSDs, SpanDB achieves 96-140% of its throughput, with a 2.3-21.6× lower latency, at a cheaper storage configuration.

1 Introduction

Persistent key-value (KV) stores are widely used today to store data in various formats/sizes for a wide range of applications, such as online shopping [32], social networks [12], metadata management [7], etc. The write-friendly log-structured merge tree (LSM-tree) is widely adopted as the underlying storage engine by mainstream KV stores, such as RocksDB [1], LevelDB [28], Cassandra [23], and X-Engine [32]. It remains appealing as production KV environments are often found write-intensive [9, 14, 25, 32, 46], especially due to aggressive memory caching [11, 50, 53].

The recent availability of fast, commodity NVMe SSDs can bring dramatic KV performance boosts, as demonstrated by recent systems, such as KVell [46] and KVSSD [40]. By either discarding the LSM-tree data structures designed for hard disks or offloading KV data management to specialized hardware, these systems provide high throughput and scalability, with the entire dataset hosted on high-end devices.

This work, instead, aims at *adapting mainstream LSM-tree based KV design* to fast NVMe SSDs and I/O interfaces, with a special focus on *cost-effective deployment in production environments*. This is motivated by our study (Sec 2) showing that current LSM-tree based KV stores fail to exploit the full potential of NVMe SSDs. For example, deploying RocksDB atop Optane P4800X only improves throughput by 23.58% compared with a SATA SSD for a 50%-write workload. In particular, the I/O path of common KV store designs severely under-utilizes ultra-low latency NVMe SSDs, especially for small writes. For instance, going through ext4 brings a latency 6.8-12.4× higher than via the Intel SPDK interfaces [37].

This hurts particularly write-ahead-logging (WAL) [52], crucial for data durability and transaction atomicity, which sits on the critical path of writes and is bottleneck-prone [31]. Second, existing KV request processing assumes slow devices, with workflow designs embedding high software overhead or wasting CPU cycles if switched to fast, polling-based I/O.

In addition, new NVMe interfaces come with access constraints (such as requiring binding the entire device for SPDK access, or recommending pinning threads to cores). This complicates KV design to utilize high-end SSDs for different types of KV I/O, and renders current common practices, such as synchronous request processing less efficient.

Finally, top-of-the-line SSDs like the Optane are costly for large-scale deployment. As large, write-intensive KV stores inevitably possess large fractions of cold data, to host all data on these relatively small and expensive devices is likely beyond the budget of users or cloud database service providers.

Targeting these challenges, we propose SpanDB, an LSM-tree based KV system that adopts *partial deployment of high-end NVMe SSDs*. It is based on a comprehensive analysis of

[*] {cighao, rcy, chengli7, ylxu}@ustc.edu.cn, xma@hbku.edu.qa. Cheng Li is the corresponding author.

bottlenecks/challenges in porting a popular KV store to use SPDK I/O (Sec 2), and contains the following innovations:

- It scales up the processing of all writes and reads of more recent data by incorporating a relatively small yet fast *speed disk* (*SD*), while scaling out data storage on one or more larger and cheaper *capacity disks* (*CD*).
- It enables fast, parallel accesses via SPDK to better utilize the SD, bypassing the Linux I/O stack and allowing high-speed WAL writing in particular. (To our best knowledge, this is the first work studying SPDK support for KV stores.)
- It devises an asynchronous request processing pipeline suitable for polling-based I/O, which removes unnecessary synchronization, aggressively overlaps I/O wait with in-memory processing, and adaptively coordinates foreground/background I/O.
- It strategically and adaptively partitions data according to the actual KV workload, actively involving the CD for its I/O resources, especially bandwidth, to help share the write amplification common in contemporary KV systems.

We implement SpanDB as an extension to Facebook's RocksDB [1], a leading KV store deployed in many production systems [2, 5]. SpanDB re-designs RocksDB's KV request processing, storage management, and group WAL writing to utilize fast SPDK interfaces, and retains RocksDB's data structures and algorithms, such as LSM-tree organization, background I/O mechanism, and transaction support features. Therefore its design stays complementary to many other RocksDB optimizations [9, 10, 17, 48, 57]. Existing RocksDB databases can be migrated to SpanDB when an SD is added.

Our evaluation using YCSB and LinkBench shows that SpanDB significantly outperforms RocksDB in all categories (throughput, average latency, and tail latency) in all test cases, especially write-intensive ones. Against KVell, a recent system designed to leverage high-end SSDs, SpanDB delivers higher throughput in most cases (at a fraction of KVell's latency), without sacrificing transaction support.

2 Background and Motivation

2.1 LSM-tree based KV Stores

Overall architecture. LSM-tree based KV stores organize on-disk data in *levels*, denoted as L_0, L_1, ..., L_k, with capacity generally growing by 10× between adjacent levels except L_0. KV pairs are stored in *Static Sorted Tables (SSTs)*, each an immutable file. To avoid data loss/inconsistency, a sequential write-ahead-log (WAL) file, often sized around tens of GBs, is maintained on persistent storage. Updates are logged there before being made visible, upon the completion of a write operation/transaction. In-memory updates are made in *MemTables*, one active while the rest are immutable. The active MemTable accommodates updates and becomes immutable when full, whereupon one or more immutable MemTables need to be flushed to make space for a new active one.

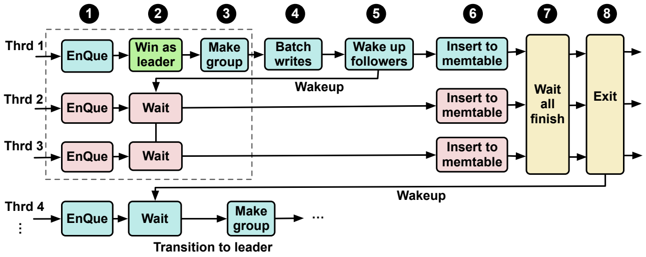


Figure 1: RocksDB group WAL write workflow

Foreground write/read. Upon the arrival of a write operation/transaction, to avoid data loss/inconsistency, its updates (along with associated metadata) must be first appended to a WAL file on persistent storage. Then, the corresponding changes made to the database can be applied to the active MemTable for subsequent visits. Given that failures are common on typical KV platforms today [18, 20, 26, 62], WAL [52] remains an integral part of customer facing databases and sits on the critical path of processing write requests.

User reads may generate random accesses at multiple tree levels, until the target key hits at a certain level or misses all the way. Though production KV systems today greatly improve average read performance through aggressive in-memory caching [11, 25, 50, 53, 60], disk I/O cannot be avoided, especially with larger databases or lower access locality. The inevitable accesses to slow storage contribute heavily to tail latency and may affect the overall performance.

Background flush and compaction. These include (1) *flush*, where an immutable MemTable is written to an L_0 SST file (often making L_0 temporarily larger than L_1), and (2) *compaction*, where SST files selected from a level L_i are read and merged with SSTs of overlapping key ranges at level L_{i+1}, with invalid KV pairs removed. The former is triggered by the number of immutable MemTables reaching a limit, and the latter by a level becoming full. Both operations create large, sequential I/O, whose impact on foreground request processing manifests in I/O contention and write stalls (when user writes need to wait for flushes to empty MemTable space).

Foreground-background coordination. RocksDB and LevelDB control the rate of background I/O through a user-configurable number of flush/compaction threads. They are activated when there are background I/O tasks, sleeping otherwise. Researchers have noted the performance impact of background thread settings and proposed related optimizations [9]. However, existing solutions still retain the background thread design, assuming slow I/O and interrupt-based synchronization, which does not work well with new, polling-based I/O interfaces (to be discussed below).

2.2 Group WAL Writes

The current common practice in writing WAL is *group logging*, which batches multiple write requests for one log data write [27, 30, 54, 76]. This technique is widely adopted by mainstream databases today, including MySQL [4], MariaDB [3], RocksDB [1], LevelDB [28], and Cassandra [44]. Beside fault tolerance, group logging also offers better write performance on slow storage devices (where random accesses tend to be even slower), by promoting sequential writes.

Fig 1 illustrates the RocksDB/LevelDB group logging workflow. The WAL write process is sequential: at any time, at most one group is writing to the log. When there is an ongoing write, worker threads handling write requests form a new group by joining a shared queue, with the first en-queued thread designated the group's *leader* (❶ - ❸). The leader (Thread 1 in this case) collects log entries from peers, until notified to proceed by the leader of the previous group, who just finished writing. This closes the door for the current group and subsequently arriving threads will start a new one.

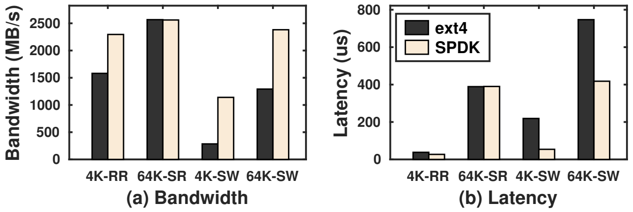
The leader writes log entries to persistent storage in a single synchronous I/O step (using `fsync`/`fdatasync`, ❹). The leader then wakes up group members to actuate updates in MemTables, making such writes visible to subsequent requests (❺ - ❻). It finalizes the group commit by advancing the *last visible sequence* to the latest sequence number among its entries (❼), disbanding the group (❽), and passing the green light to the next leader (Thread 4).

With high-end NVMe SSDs and faster I/O interfaces (details in Section 2.3), the group write time (❹) is dramatically reduced. Meanwhile, batching writes still helps by consolidating small requests. Consequently, the software overhead caused by the synchronous group logging rises to render most of the threads wasting their time on different types of wait (steps ❶-❸ and ❼). For example, we measured that RocksDB spends, on average, 68.1% of write request processing time on these 4 steps on a SATA SSD accessed via ext4, which grows to 81.0% on Optane via SPDK.

2.3 High-Performance SSDs Interfaces

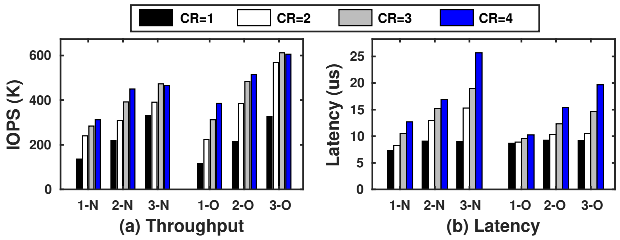
Recent high-end commodity SSDs, such as Intel Optane [36], Toshiba XL-Flash [63], and Samsung Z-SSD [59], offer low latency and high throughput [66]. Recognizing that the Linux kernel I/O stack overhead is no longer negligible in total I/O latency [45, 69], Intel developed SPDK (Storage Performance Development Kit) [37, 69], a set of user-space libraries/tools for accessing high-speed NVMe devices. SPDK moves drivers into user space, avoiding system calls and enabling zero-copy access. It polls hardware for completion instead of using interrupts and avoids locks in the I/O path. Here we summarize SPDK performance behavior and policy restrictions found relevant to KV stores in this work.

SPDK overall performance. We benchmarked two modern NVMe SSDs, Intel Optane P4800X and P4610. Fig 2 gives Optane results for request type/size combinations, simulating typical LSM-tree based KV I/O as described earlier (P4610 results show similar trends). We use `write` calls for ext4 (each followed with `fdatasync`), and the SPDK build-in perf tool (`spdk_nvme_perf`) for SPDK.



(a) Bandwidth (b) Latency

Figure 2: Optane P4800X performance via ext4 and SPDK at different request sizes by 16 threads. "RR", "SR", and "SW" stand for random read, sequential read, and sequential write, respectively.



(a) Throughput (b) Latency

Figure 3: Concurrency evaluation w. 4KB sequential writes

For large sequential reads, going through a file system (as done by current KV stores) actually matches SPDK results. 4KB sequential writes (WAL-style) via ext4, meanwhile, achieve a small fraction of the hardware potential, with latency 4.05× higher than SPDK (IOPS accordingly lower). The 4KB random read and 64KB sequential write tests see ext4-SPDK gaps between these extremes. Such results highlight that SPDK may bring significant improvement to KV I/O, especially for logging and write-intensive workloads.

SPDK concurrency. To assess SPDK's capability of serving concurrent sequential writes, we profile individual SPDK requests, and find the bulk of the 7-8μs single-thread latency indeed occupied by busy-wait, which grows with more threads concurrently writing, due to slower I/O under contention.

We then devise a pipeline scheme, where each thread manages multiple concurrent SPDK requests. It allows to "steal" I/O wait time to issue new requests and check the completion status of outstanding ones (each taking under 1μs).

Fig 3 gives latency and throughput results on the Intel P4610 (N) and Intel Optane (O) SSDs. We vary the number of threads ("3-N" having 3 threads writing to SSD N) and the upper limit for concurrent requests per thread ("CR=2" having each thread issuing up to 2 requests). NVMe SSDs do offer parallelism beyond utilized by the current RocksDB/LevelDB single-WAL-writer design. In particular, Optane (O) shows higher concurrency than P4610 (N), with slower latency and faster throughput growth with more writers. However, even with O, going beyond 3 concurrent writers does not provide higher SPDK IOPS: Using 3 loggers each with CR=3 appears to offer peak WAL speed, which we denote as 3L3R. N, on the other hand, saturates at 2L4R.

SPDK access restrictions. The performance benefit of fast

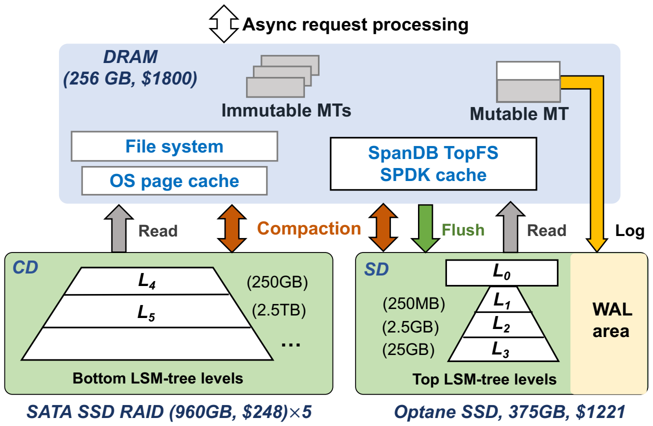


Figure 4: SpanDB storage overview. The dimmed (grey) components reuse RocksDB implementation

SPDK-enabled access to high-end NVMe SSDs comes with strings attached: once an SSD is bound to SPDK by one process, it cannot be accessed by others, either via SPDK or via the Linux I/O stack. This simplifies inter-workload isolation associated with user-level accesses, but also disables partial deployment of file systems to an SSD accessed via SPDK. In addition, users are recommended to bind SPDK-accessing threads to specific cores [22]. We verified that not doing so brings significant I/O performance loss. This, plus the polling-based I/O mode, renders the common practice of using background flush/compaction threads unsuitable for SPDK accesses: unbound threads suffer slow I/O, while bound threads cannot easily yield core resources when idle.

3 SpanDB Overview

Design rationale. We propose SpanDB, a high performance, cost-effective LSM-tree based KV store using heterogeneous storage devices. SpanDB advocates the use of a small, fast, and often more expensive NVMe SSD as a *speed disk (SD)*, while deploying larger, slower, and cheaper SSD (or arrays of such devices) as the *capacity disk (CD)*. SpanDB uses the SD for two purposes: (1) WAL writes and (2) storing the top levels of the RocksDB LSM-tree.

As WAL processing cost is user-visible and directly impacts latency, we reserve enough resources (cores and concurrent SPDK requests, plus sufficient SPDK queues), to maximize its performance. Meanwhile, WAL data only needs to be maintained till the corresponding flush operation and typically require GBs of space, while today's "small" high-end SSDs, such as Optane, offer over 300GB. This motivates SpanDB to move the top levels of the RocksDB LSM-tree to the SD. This also offloads a significant amount of flush/compaction traffic from the CD, where the bulk of colder data resides.

SpanDB architecture. Fig 4 gives a high-level view of SpanDB storage structure. Within DRAM, it retains the RocksDB MemTable design, with one mutable and multiple immutable MemTables. Note that SpanDB introduces no modifications to RocksDB's KV data structures, algorithms, or operation semantics. The major difference here lies in its asynchronous processing model (Sec 4.1), to reduce synchronization overhead and adaptively schedule tasks.

On-disk data are distributed across the CD and SD, two physical storage partitions. The SD is further partitioned, with a small *WAL area* and the rest of its space used as a *data area*. SpanDB manages the SD as a raw device via SPDK and redesigns the RocksDB group WAL writes (Sec 4.2), for fast, parallel logging, improving logging bandwidth by 10×. The data area manages raw SSD pages to host the top levels of the LSM-tree (Sec 4.3). To minimize changes to RocksDB, here SpanDB implements *TopFS*, a lightweight file system (including its own cache), which allows easy and dynamic level relocation. The CD partition, meanwhile, stores the "tree stump", often containing the colder majority of data. Its management remains unchanged from RocksDB, accessed via a file system and assisted by the OS page cache.

Fig 4 also depicts the different types of SpanDB I/O traffic. While the SD WAL area is dedicated to logging, its data area receives all flush operations, which write entire MemTables to L_0 SST files. In addition, both SD data area and CD accommodate user reads and compaction reads/writes, where SpanDB performs additional optimization to enable simultaneous compaction on both partitions and automatically coordinate foreground/background tasks. Finally, SpanDB is capable of dynamic tree level placement based on real-time bandwidth monitoring of both partitions.

Sources of performance benefits. SpanDB improves LSM-tree based KV store design in multiple ways:

- By adopting a small yet fast SD accessed via SPDK, it speeds up WAL by fast, parallel WAL writes.
- By using the SD also for data storage, it optimizes the bandwidth utilization of such fast SSDs.
- By enabling workload-adaptive SD-CD data distribution, it actively aggregates I/O resources available across devices (rather than using CD only as an "overflow layer").
- Though mainly optimizing writes, by offloading I/O to the SD, it reduces tail latency with read-intensive workloads.
- By trimming synchronization and actively balancing foreground/background I/O demands, it exploits fast polling I/O while saving CPU resources.

Limitations. We recognize two limitations with SpanDB's approach: (1) due to the aforementioned SPDK access constraint, the SD needs to be bound to one process, making it hard to share this resource; (2) for all-read workloads, SpanDB produces little speedup and introduces slight overhead in asynchronous processing.

4 Design and Implementation

4.1 Asynchronous Request Processing

KV stores like RocksDB and LevelDB (plus many new systems based on them [8–10, 13, 17, 48, 51, 57, 73]) use embed-

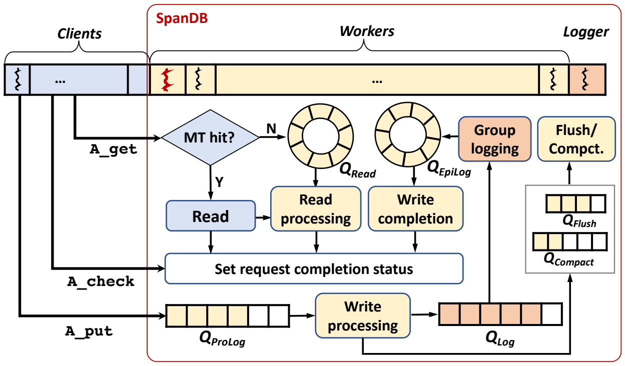


Figure 5: Asynchronous request processing workflow

```
Request *req = null;
while(true){
    if(req == null)
        req = GenerateRequest();
    LogsDB->A_put(req->key, req->value, req->status);// issue async req

    if(!(req->status->IsBusy())){
        pending_queue->enqueue (req);
        req = null; // ready to generate next req
    }
    for(Request* r in pending_queue){
        if (A_check(r->status)==completed) { // check outstanding reqs
            pending_queue.remove(r);
            custom_process(r);
        }
    } // end for
} // end while
```

Figure 6: SpanDB API example

ded DB processing, where all foreground threads assume the “client” role, each synchronously processing one KV request at a time. With such processing often being I/O-bound (especially with WAL writes), users typically obtain higher overall throughput (requests per second) by *thread-overprovisioning*, having more client threads than cores. With fast NVMe SSDs and interfaces such as SPDK, as discussed in Section 2.3, thread synchronization (such as sleep and wakeup) could easily take longer than an I/O request. In this case, thread overprovisioning not only trades off latency, but also reduces overall resource utilization and consequently throughput.

In addition, with polling-based SPDK I/O, having threads co-exist on the same cores loses the appeal of improving CPU utilization during I/O waits. This also applies to the common practice of managing LSM-tree flush/compaction tasks using background threads. In particular, as “fsync” with SPDK I/O involves busy-wait, the existing RocksDB design of unleashing potentially many background threads would create huge disruption to other threads and waste CPU cycles.

Recognizing these, SpanDB adopts asynchronous request processing, as illustrated in Fig 5. On an n-core machine, users configure the number of client threads as N_{client}, each occupying one core. The remaining $(n-N_{client})$ cores host SpanDB internal *server* threads, internally partitioned into two roles: *loggers* and *workers*. All these threads spin on their assigned cores. Loggers are dedicated to WAL writes, while workers handle both background processing (flush/compaction) and non-I/O tasks such as MemTable reads and updates, WAL entry preparation, and transaction related locking/synchronization. Based on the write intensity observed, a *head-server* thread automatically and adaptively decides the number of loggers, who are bound to cores with SPDK queue allocation that protect WAL write bandwidth.

Asynchronous APIs. SpanDB provides simple, intuitive asynchronous APIs. For existing RocksDB synchronous `get` and `put` operations, it adds their asynchronous counterparts `A_get` and `A_put`, plus `A_check` to examine request status. Similar API expansion applies to `scan` and `delete`. Accordingly, SpanDB expands RocksDB’s `status` enumeration.

Fig 6 gives a sample client code segment. Here the client adopts the inherent spirit of asynchronous processing: to overlap wait with active work. It issues `A_put` requests in a loop, moving on to check the status of outstanding requests (and perform proper processing upon their completion), followed by issuing another request. A new request may be temporarily rejected by SpanDB, via the *IsBusy* status set within the `A_put` call, in which case the client will resubmit later.

SpanDB request processing. SpanDB manages the stages of foreground request processing, as well as background flush/compaction tasks in a number of queues. These queues pass sub-tasks among threads and also provide feedback on a certain system component’s stress level. Based on such feedback, SpanDB could regulate the client request issuing rate (via the aforementioned `IsBusy` interface) or dynamically adjust its internal task allocation among workers.

Fig 5 illustrates the relevant SpanDB task queues. The flush and compaction queues (Q_{Flush} and $Q_{compact}$) are from RocksDB’s existing design, though SpanDB modifies the actual operations to use SPDK I/O. In addition, SpanDB adds four queues: one for reads (Q_{Read}), and three to break up writes (Q_{ProLog}, Q_{Log}, and Q_{EpiLog}).

For asynchronous reads, SpanDB retains the RocksDB synchronous model when a request requires no I/O. With typical locality in KV applications, many reads are served from the MemTable, especially with larger MemTables enabled by spacious DRAM today. Given a key, the client quickly checks whether it is a MemTable hit and if so, completes the read operation itself. Such a “lucky read” takes only 4-6μs end to end, as opposed to 30μs on average even when reading from Optane under contention. Otherwise, the client inserts the request into Q_{Read} and returns. A worker will later pick it up, completing the rest of the RocksDB read routine and setting its completion status.

For asynchronous writes, SpanDB breaks its processing into three parts. The client simply dumps a request into Q_{ProLog}, to be processed by a worker. The latter generates a WAL log entry, which in turn is passed into Q_{Log}. Both queues are designed to promote batched logging (described in Sec 2.2): a worker/logger would grab all the items in these queues. Beyond batching, the loggers pipeline log writes, maximizing SPDK write concurrency (see Sec 4.2). After writing a batch to the SD, a logger adds the appropriate requests to Q_{EpiLog}, for workers to complete their final processing, including the actual MemTable updates. Like reads, tasks here require individual attention and no speedup can be achieved from their batching. As seen in Fig 5, Q_{ProLog} and Q_{Log} are flat lock-free queues, which allow easy "grab all" dequeuing. The other two, Q_{Read} and Q_{EpiLog}, are circular queues and only require locks in dequeue operations.

Task scheduling. The above SpanDB queues provide natural feedback for adjusting internal resource allocation. Our SPDK benchmarking results (Fig 3) shows that high-end NVMe SSDs offer parallelism but can be saturated by a few cores each issuing several concurrent requests. Hence SpanDB starts with one logger, growing and shrinking this allocation between 1 and 3 according to the current write intensity. The workers, however, are flexible to work on all the other queues, both foreground and background. Among the 3 foreground queues, SpanDB performs load balancing based on their queue length weighted by their average per-task processing time. Between the foreground and background queues, SpanDB prioritizes foreground, with an adaptive threshold to monitor background queue length, to proactively perform cleaning up, especially with write-intensive workloads.

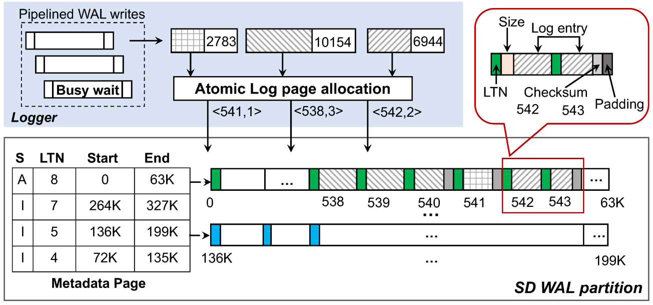
Transaction support. SpanDB fully supports transactions and provides an asynchronous commit interface `A_commit` by making a few minor changes to RocksDB. Note that in RocksDB's transaction mode, writes will generate WAL entries in an internal buffer, which is only written by the commit call. The difference here is that `A_commit` inserts corresponding write tasks into Q_{ProLog}.

4.2 High-speed Logging via SPDK

Enabling parallelism and pipelining. SpanDB uses SPDK to flush log entries to raw NVMe SSD devices, bypassing the file system and Linux I/O stack. It retains the group logging mechanism described in Sec 2.3, but enables multiple concurrent WAL write streams. Rather than having one client as leader (and forcing followers to wait), it employs dedicated loggers, who issue simultaneous batch writes. Each logger grabs all requests it sees in Q_{Log} and aggregates these WAL entries into as few 4KB blocks as possible. It performs pipelining by stealing the SPDK busy-wait time for one request to prepare/check others, as introduced in Sec 2.3. For instance, with 2L4R, there are up to 8 outstanding WAL write groups.

Log data management. Parallel WAL writes complicate log data management, especially on a raw device without a file system. Luckily, with atomic 4KB SPDK writes, coordinating concurrent WAL streams adds little overhead.

SpanDB allocates a configurable number of logical pages on the SD to its WAL area (10GB in our evaluation), each with a unique *log page number (LPN)*. One of them is set aside as a *metadata page*. At any time, there is only one mutable MemTable, whose log "file" grows. We allocate a fixed number of *log page groups*, each containing consecutive pages and large enough to hold logs for one MemTable. Occupied log pages are organized by their corresponding MemTables: SpanDB conveniently reuses the RocksDB MemTable's "log file number" field as a *log tag number (LTN)*, embedded at the beginning of all log pages for recovery.



S	LTN	Start	End
A	8	0	63K
I	7	264K	327K
I	5	136K	199K
I	4	72K	135K

Figure 7: SpanDB's parallel WAL logging mechanism

Fig 7 gives an example of having four MemTables, one mutable (active) and three immutable, with different status ("A" for "active" and "I" for "inactive") in the metadata page. The long stripes in the bottom show two of the log page groups allocated. After a MemTable is flushed, its entire stripe of log pages is recycled, guaranteeing a MemTable's contiguous log storage. For each immutable MemTable, the metadata page records the start and end LPN of its log pages. Given that typical KV stores use a small number of MemTables, one page is more than enough to hold such metadata.

With loggers issuing concurrent requests, each supplying a WAL data buffer and size, the only synchronization point is log page allocation. We implement lightweight *atomic page allocation* with compare-and-swap (CAS) operations. Fig 7 shows 3 requests allocated 1, 3, and 2 pages, respectively, who can then proceed in parallel. These WAL writes do not modify the metadata page, where the per-MemTable end LPN is only appended when that MemTable becomes immutable.

Within a log page, the logger first records the current LTN, followed by a set of log entries, each annotated with its size. The zoom-in part in Fig 7 portraits such layout, including the per-entry checksum (already calculated in RocksDB).

Correctness. SpanDB's parallel WAL write design preserves the RocksDB consistency semantics. It does not change the concurrency control mechanism used to coordinate and order client requests. Therefore, transactions with happened-before restrictions never appear out of order in the log pages, as briefly explained below. RocksDB's default isolation guarantee is `READ COMMITTED`. It also checks *write-write conflicts* and serializes two concurrent transactions that simultaneously update common KV items. With these two isolation guarantees, for any two update transactions T_1 and T_2, `READ COMMITTED` implies that if T_1 happens before T_2 (*i.e.*, T_2 sees the effects of T_1), then T_1 must commit before T_2 started. By the design of the RocksDB group WAL write protocol, the above implies that the log entries of T_1 and T_2 should appear in two batches, where the batch commit of T_1 arrive earlier than and complete before the one of T_2. While log batches are written in parallel with SpanDB, they pass a serialization point for atomic page allocation. Therefore T_1's batch is still guaranteed to obtain a lower sequence number than the one of T_2, for the latter to see the updates of the former. Similarly, When recovering from WAL data, SpanDB always performs redo in ascending order of sequence numbers.

Log recovery. Recovery is rather straightforward. When rebooting from a crash, the recovery process first reads the metadata page, to retrieve the number of log page groups and their corresponding page address ranges. The actual recovery from a log page group is highly similar to RocksDB's from a log file. Again the LTN number in each page helps identify the "end" of the active log page group.

However, the one complication we find is that as SpanDB recycles log page groups, which contain old log pages, during recovery SpanDB needs to know which pages of the current log group have been overwritten. RocksDB relies on the file system during recovery: it reads whatever data is contained in the active log file. Without the file system, SpanDB could persist a separate metadata update or wipe out old log pages (*e.g.*, by writing 0s) before recycling them. Both approaches double the WAL I/O volume and cut the SD's effective WAL write bandwidth in half. Instead, we reuse the per-MemTable LTN as a log page "color". Since the SSDs can guarantee 4K atomic writes to the device and the LTN is always written at the beginning of a page, the pages themselves reveal the location of the last successful writes. Recall the metadata page maintains the current/active LTN (the one with status "A") – a page within this group but with an obsolete LTN has not yet been overwritten from the current MemTable.

4.3 Offloading LSM-tree Levels to SD

For sustained, balanced execution of KV servers, SpanDB migrates the top levels of the RocksDB LSM-tree to the SD, offering users more return from their hardware investment. Below we discuss the major challenges and solutions involved.

Data area storage organization. One constraint in using SPDK on an NVMe SSD is that the whole device has to be unbound from the native kernel drivers, and cannot be accessed through the conventional I/O stack. Therefore one cannot partition the SD, to use SPDK only for writing WAL to one area and install a file system on the other.

To minimize modifications to RocksDB I/O, SpanDB implements TopFS, a stripped-down file system, providing familiar file interface wrappers on top of SPDK I/O. The SST files' append-only and thereafter immutable nature, plus their single-writer access pattern, simplifies the TopFS design. For example, file sizes are known at creation (for flush, with an immutable MemTable's size fixed) or have a known limit (for compaction). Also, each SST file is written in entirety once, by a single thread, from either flush or compaction. In both cases, the input data are not deleted till the SST file write successfully completes. In addition, TopFS does guarantee data persistence upon file close. These enable the allocation of per-file contiguous LPN ranges, similar to the aforementioned log page groups. Metadata management is then simple: a hash table, indexed by file name, stores the files' start and end LPNs. TopFS manages space allocation using an LPN free list, where contiguous LPN ranges are merged.

Ensuring WAL write priority. While flush/compaction could eventually block foreground writes if neglected long enough, in most cases, their latency remains hidden from users. Therefore the SD should ideally utilize the *residual bandwidth* available, but yield to WAL writes, whose latency is fully visible to users. SPDK provides enough NVMe *queue pairs* (each composed of one submission and one completion queue): 31 on Intel Optane P4800X and 128 on Intel P4610. This enables separate management of different request types. Unfortunately, none of the existing commodity SSDs implement priority management over these queues [29]. Also, these queues offer very limited operations: users could only issue requests and check completion status.

Therefore, besides the foreground-background coordination done at its queues (Section 4.1), SpanDB needs to prioritize WAL requests. We found their priority could be effectively protected by (1) allocating dedicated queues to each logger request slot (*i.e.*, 8 queues for L2R4), (2) reducing the flush/compaction I/O request size from the RocksDB default of 1MB to 64KB to minimize their I/O contention with WAL, and (3) limiting the number of *worker* threads assigned to perform flush/compaction.

SpanDB SPDK cache. Another challenge SpanDB faces is that SPDK bypasses the OS page cache. If unattended, this brings excellent raw I/O but disastrous application I/O performance. To overcome this, we implement SpanDB's own cache on TopFS. Note that with SPDK I/O, all data buffers passed must be allocated in pinned memory via `spdk_dma_malloc()`. SpanDB reuses such buffers as a cache, hereby saving additional memory copying.

Upon SpanDB initialization, it allocates a large memory cache (size configurable) in hugepage. Upon an SST file's creation, SpanDB reserves the appropriate number of contiguous 64KB buffers in the cache (recall that the file size or size limit is known). SpanDB manages this cache using another hash table, again with the RocksDB SST file name as the key. The value field is an array storing the cache entry for each file block, storing the appropriate memory address if the block is cached, otherwise `NULL`. The block size configuration clearly involves a tradeoff between cache data granularity and metadata overhead. Our evaluation uses the SpanDB default block size of 64KB, producing a <500KB metadata overhead for a 100GB database.

Dynamic level placement. With all the above mechanisms, we can dynamically adjust the number of tree levels residing on SD. Initially, we pursued an analytical model to directly compute an optimal SD-CD level partitioning that maximizes overall system throughput. However, we could not find accurate LSM-tree write amplification models that agree with our measurement. In particular, state-of-the-art work on this front [49] seems to not take into account write speed and its variation. Our tests show that these factors could heavily impact the transient "tree shape" (with the top levels bulging out at different degrees beyond their size limit) and consequently the write/read amplification level.

Therefore, SpanDB settles for ad-hoc, dynamic partitioning, by observing the sustained resource utilization level imbalance between the SD and CD. Its head-server thread monitors the SD bandwidth usage, and triggers the SST file relocation when it is below BW_L, till either it reaches BW_H or the SD is full, where BW_L and BW_H are two configurable thresholds.

Rather than migrating data between SD and CD, as the SST files constantly go through merging, SpanDB gradually "promotes" or "demotes" a whole level by redirecting their file creation to a different destination. It has a pointer that indicates currently which levels should go to the fast NVMe device. For example, a pointer of 3 covers all top 3 levels. However, this pointer only determines the destination of *new* SST files. Therefore it is possible to have a new L3 file on SD and an older L2 file on CD, though such "inversions" are rare as the top levels are smaller and their files are updated more often.

5 Evaluation

5.1 Experimental Setup

We implemented SpanDB[1] on top of RocksDB, with around 6000 lines of C++ code for its core functionality, plus 300 lines for integration with RocksDB.

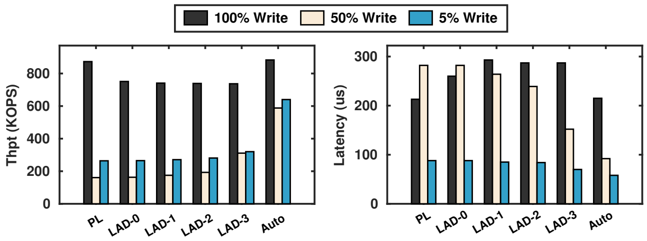
Platform. We use a server with 2 20-core Xeon Gold 6248 processors and 256GB DRAM, running CentOS 7.7. The storage setting, denoted in "CD-SD" pairs, involves four data center device types. Among them, SATA SSDs (Intel S4510, "S") are used to form an 4+1 RAID5 group. As SPDK does not apply to SATA devices, S is used as CD only.

Beside Optane P4800X ("O"), we test two more Intel DC NVMe SSDs as CD and SD respectively: P4510 ("N1") and P4610 ("N2"), the former being larger, cheaper, and with higher bandwidth. The device details are in Table 1. Finally, we access CD via ext4, widely adopted in KV stores studies [13, 17, 51, 57].

Baseline and system configurations. Our natural baseline is vanilla RocksDB (v6.5.1), the base of SpanDB's development. Unless otherwise stated, all tests with RocksDB and SpanDB share the following configurations. Considering the current trend of larger DRAM space in servers, we use four 1GB MemTables, and set the maximum WAL size to 1GB. RocksDB is set to use up to 6 threads for compaction and 2 for flush. We follow common practice in performance evaluation that turns off compression when using synthetically generated requests [9, 48, 51, 57]. The remaining parameters are set to RocksDB default. Additionally, we compare with two recent key value stores designed for high-performance SSDs, namely KVell [46] and RocksDB-BlobFS [61].

Table 1: Enterprise disks tested (pricing from CDW-G on 09/15/2020). DWPD (Drive Writes Per Day) measures the times/day one could overwrite an entire drive for its lifetime. Note that H and S are used in (4+1) RAID5 arrays, while the listed numbers here are single-disk data.

ID	Model	Interface	Capacity	Price	Seq. write bandwidth	Write latency	Endurance (DWPD)
S	Intel SSD DC S4510	SATA	960 GB	\$248 \$0.26/GB	510 MB/s	37 us	1.03
N1	Intel SSD DC P4510	NVMe	4.0 TB	\$978 \$0.25/GB	2900 MB/s	18 us	1.03
N2	Intel SSD DC P4610	NVMe	1.6 TB	\$634 \$0.40/GB	2080 MB/s	18 us	1.03
O	Intel Optane SSD P4800X	NVMe	375 GB	\$1221 \$3.25/GB	2000 MB/s	10 us	30



(a) Throughput

(b) Latency

Figure 8: Impact of data placement in SpanDB (S-O steup, 512GB database)

Workloads and Datasets. We run microbenchmarks and two popular KV workloads, YCSB [16] and Facebook's LinkBench [6]. For most tests with YCSB, we follow common practice [46, 48, 51] and use 1KB KV item size, loading a 512GB database with randomly generated keys as the initial state. The query phase issues 20M requests (preceded by 30% extra requests for warm-up).

5.2 Microbenchmark Results

Adaptive KV data placement. To assess SpanDB's automatic LSM-tree level placement, we use 3 YCSB-like workloads with different write intensity (Fig 8). We tested a 512GB database on the S-O device combination. To compare with SpanDB's adaptive setting ("Auto"), we configured SpanDB with different fixed placement options: "PL" ("pure logging", where the SD is used solely for WAL writes), and "LAD-*n*" (the top *n* levels of the LSM-tree is placed on the SD). The left and right charts show request processing throughput and average latency, respectively.

The results indicate that different workloads see different configuration sweet points. With a write-only workload, PL enables the fastest absorption of write bursts, dedicating the SD to WAL writes. The fixed placement plans (LAD-0 to LAD-3) deliver lower yet almost uniform performance, due to that their total data size (27.3GB) is rather small relative to the total 512GB database. They are slower in writes by

[1] Publicly available at https://github.com/SpanDB/SpanDB

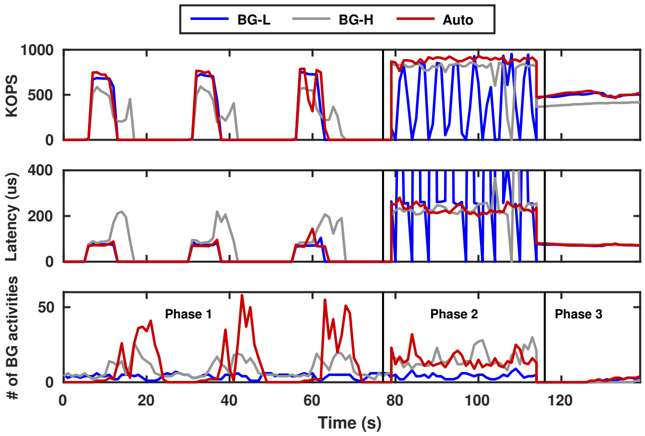
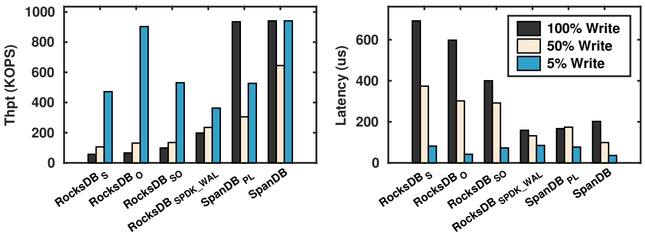


Figure 9: Impact of SpanDB background I/O configurations (S-O setup, 512GB database)



(a) Throughput **(b) Latency**

Figure 10: Performance of different RocksDB and SpanDB configurations (S-O setup, 100GB database)

adding flush/compaction traffic to the SD, while moving one more or fewer (small) layer here has little impact. Please note that we cannot evaluate LAD-4 here, as the total database size (over 300GB), plus the WAL area and the temporary top tree level growth to accommodate fast writes, would run out of the usable space of the Optane disk (around 330GB in our experience). SpanDB's auto policy here matches the PL performance by adopting the same placement.

With more read-intensive workloads, using the SD for data helps by speeding up reads. Again only LAD-3 brings visible improvement as the previous levels are quite small. SpanDB's auto placement, however, roughly doubles throughput and halves latency from LAD-3. Its dynamic strategy does not have to migrate an entire tree level: here it ends up moving about 72% of the L4 data to SD, cutting average read latency significantly. For the rest of the paper, we evaluate SpanDB with its auto data placement.

Adaptive background I/O coordination. Here we use a multi-phase workload to simulate time-varying user behavior common in production environments [32]. It begins with bursty requests, issuing 1.5M requests at the beginning of multiple 25-second episodes with 50% writes and 50% reads (Zipfian key distribution), followed by around 35 seconds of 100% writes, and finally 25 seconds of 95% reads. Fig 9 portraits the request throughput, latency, and background activity level (flush/compaction task counts as reported in RocksDB).

We compare SpanDB's auto adaptation with two fixed configurations: "BG-L" (RocksDB default, one thread each for compaction and flush), and "BG-H" (6 compaction and 2 flush threads). During phase 1 (bursty), BG-H performs the worst, with 2× higher average latency and 39% lower average throughput than BG-L during each burst. After an initial period of write accumulation, the foreground tasks become severely interfered by its aggressive compaction. "Auto" behaves quite similarly to BG-L during the write request bursts, prioritizing foreground tasks. Unlike with the fixed thread allocation in RocksDB, its background I/O is not constrained to a few threads. So after the burst passes, SpanDB Auto loses no time in catching up with background tasks, resulting in "background compaction bursts" (red peaks in the bottom figure) much more intense than both BG-L and BG-H. Overall, this leads to faster completion of backlogged compaction tasks, and better preparation for future write bursts.

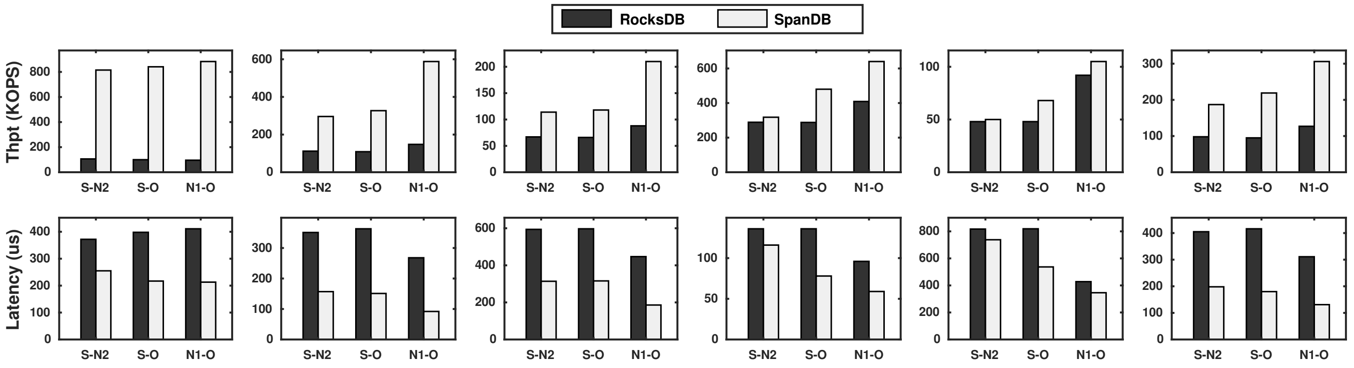
In the second phase (all-write), BG-L regularly stalls foreground processing, producing dramatic latency/throughput fluctuations, which does not happen with the more compaction-conscious BG-H or Auto. With higher background resource allocation, BG-H still performs worse than Auto (due to its less pro-active compaction), obtaining slightly lower throughput and having one write stall. For the last phase (read-intensive), with light flush/compaction load, BG-L and Auto achieve nearly identical performance, while BG-H lags behind in throughput, due to wasting thread allocation (as required by SPDK to be bound to a core) to background tasks.

This confirms that SpanDB's asynchronous workflow, designed mainly to reduce software overhead with polling I/O, also enables adaptive background task scheduling.

Breakdown analysis. Fig 10 breaks down SpanDB's improvement by incrementally enabling its individual techniques, again with workloads at different write intensity. The first 4 bar groups show variants of RocksDB, while the last 2 of SpanDB. To enable *RocksDB*$_O$, RocksDB's execution on a single fast disk (Optane), we use a smaller database (100GB). With *RocksDB*$_{SO}$, RocksDB uses the SD (O) for WAL and CD (S) for all data. *RocksDB*$_{SPDK\_WAL}$ uses the same setting, only with WAL writes via SPDK instead of ext4. *SpanDB*$_{PL}$ adds asynchronous processing and parallel WAL writes, while *SpanDB* enables auto-placement of data.

From the all-write results, one sees clearly how little the hardware upgrade matters with RocksDB (*RocksDB*$_S$ to *RocksDB*$_O$). Separating logging with data I/O helps (*RocksDB*$_{SO}$), and adopting SPDK further doubles write throughput. Still, *RocksDB*$_{SPDK\_WAL}$ only unlocks a small fraction of the Optane disk's concurrent small write performance, as demonstrated by SpanDB (both PL and Auto), who achieves a 4.5× throughput.

When the workload becomes balanced (50% read), the performance growth becomes less dramatic, though still very significant. Here *RocksDB*$_S$ and *RocksDB*$_O$ have almost identical



(a) 100% write (b) YCSB-A, zipfian (c) YCSB-A, uniform (d) YCSB-B (e) YCSB-E (f) YCSB-F

Figure 11: Throughput and latency of various YCSB workloads, 20M requests on 512GB database. (YCSB-A: 50% update and 50% read, YCSB-B: 5% update and 95% read, YCSB-E: 95% scan and 5% insert, YCSB-F: 50% read and 50% read-modify-write)

performance, as adding a CD helps offloading write traffic, but lowers read speed. SpanDB's auto version beats the best RocksDB variant by 2.74×, and nearly doubles the throughput of its PL version, as the fast SD accelerates reads. With 95%-read (blue bar), $RocksDB_O$ stands out among RocksDB variants, showing that with reads, the vanilla RocksDB on ext4 actually quite efficiently utilizes the Optane disk (consistent with benchmarking results in Fig 2). SpanDB's auto version, in this case, also chooses to place its data on the SD and matches the $RocksDB_O$ performance.

5.3 Overall Performance

We use YCSB and LinkBench to evaluate SpanDB's overall performance against RocksDB, on three CD-SD hardware pairs: S-N2, S-O, and N1-O. Note that RocksDB allows easy assignment of logging destination, therefore we set it to also writes WAL to the SD (its LSM-tree levels, however, cannot be relocated without substantial code change). Hence the RocksDB baseline evaluated does use both disks in the CD-SD pair, though via the file system.

YCSB write-intensive tests. As SpanDB primarily targets write optimization, we start with write-intensive workloads.

With all-write (Fig 11(a)), issuing 20M write requests (Zipfian key distribution), RocksDB delivers uniformly low performance across all three device pairs. This reveals how existing systems, logging sequentially via a file system, fail to utilize high-end SSDs well. From this baseline, SpanDB dramatically improves *both* throughput and latency, bringing a throughput increase of 7.6-8.8× across different CD-SD combinations, while reducing average latency by 1.5-2×. This higher improvement on throughput than on latency is attributed to our parallel batch logging (both in Q_{ProLog} and Q_{Log}). For example, on S-O, the RocksDB log batch size averages around 20. SpanDB has an average batch size of around 7, but may have multiple threads process batches in parallel.

Fig 11(b) and Fig 11(c) give results for YCSB-A (50% reads and 50% updates), with Zipfian and uniform key distribution, respectively. Having 50% reads, on such a large database, actually slows down overall request processing, as reads cannot be batched. Here SpanDB's improvement over RocksDB remains significant: improving throughput by 2.6-4.0× while reducing latency by 2.2-3.0× (Zipfian distribution). With more reads, both systems are more sensitive to the underlying storage hardware, and the N1-O combination excels due to N1's lower read latency than S. Meanwhile, compared with RocksDB, SpanDB harvests much more performance gain from this device pair.

With uniform distribution, SpanDB's edge over RocksDB is weakened by having more memory cache read misses. Most data reside on the CD, where SpanDB's reads work similarly as the baseline. Still, SpanDB outperforms RocksDB by 1.7-2.4× in throughput and by 1.9-2.4× in latency.

Other standard YCSB tests. Next, we run the other 3 YCSB workloads: B, E and F. Due to space limit we give Zipfian results only, and omit C (no writes) and D (similar to B).

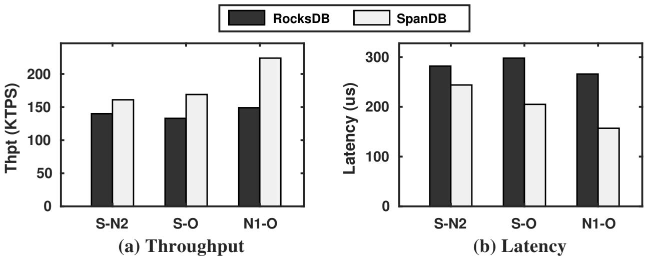
With the 95%-read YCSB-B and YCSB-E (Fig 11(d) and Fig 11(e)), SpanDB still delivers moderate enhancement: throughput growth by 1.03×-1.66×, and latency cut by 9.5%-42%. Between them, it has a smaller gain with YCSB-E, dominated by scan operations and with a higher memory hit ratio (from reading a random number of consecutive keys). YCSB-F (Fig 11(f)) contains 50% reads and 50% read-modify-writes. Though its read ratio (75%) is between YCSB-A and YCSB-B, it behaves more like YCSB-A (with read-modify-write dominated by write cost): SpanDB outperforms RocksDB significantly in both throughput and latency.

Among all tests in Fig 11, except for the most read-intensive ones (B and E), SpanDB on the lowest device setting (S-N2) significantly outperforms RocksDB on the highest one (N1-O), demonstrating its cost-effectiveness. The two 95%-read workloads highlight the benefit of a low-latency CD, while SpanDB further boosts performance across all device pairs.

Finally, we report SpanDB's impact on tail latency. Due to space limit, here we focus on the read-intensive tests (B, E,

Table 2: Tail latency in YCSB read-intensive tests (S-O)

	YCSB-B (Zipf)		YCSB-E (Zipf)		YCSB-F (Zipf)	
	RocksDB	SpanDB	RocksDB	SpanDB	RocksDB	SpanDB
P90 (us)	471.5	277.1	2844.0	1404.1	685.4	261.2
P99 (us)	803.4	507.6	6016.6	4241.6	2801.7	1848.2



(a) Throughput
(b) Latency

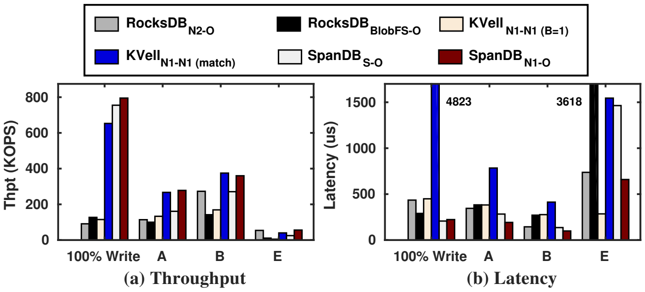
Figure 12: Performance of LinkBench

and F), on S-O, listing the P90 and P99 request latency in Table 2. Though the write-oriented SpanDB produces moderate overall performance improvement for read-intensive workloads as shown earlier, it reduces the P90 and P99 tail by up to 1.40× and 2.62×, respectively. A closer examination reveals that for mixed workloads (F), SpanDB reduces the impact of compaction on tail reads; for read-intensive (B and E), it helps by faster writes.

LinkBench transactional workload. We assess SpanDB's asynchronous transaction processing with Facebook's LinkBench [6] (Fig 12). Our test uses a 206GB database containing 600M vertices and 2622M links, performing 20M requests with LinkBench's default configuration: 56% scan, 11% write, 13% read, and 20% read-modify-write operations. Again, for this overall read-intensive workload (around 70%-read), SpanDB fares well against RocksDB, increasing throughput by up to 50.3% and cuts latency by up to 41%. The results demonstrate SpanDB's effectiveness in handling graph OLTP workloads, where WAL writes cannot be forgone.

Comparison w. NVMe SSD-based systems. Finally, Fig 13 compares SpanDB against two recent systems leveraging fast NVMe SSDs: KVell [46] and RocksDB-BlobFS [61]. Here we test with larger datasets, using a 2TB database (except for RocksDB-BlobFS, which failed to run with larger sizes and we included its 250GB test results for reference.) We assess 4 YCSB workloads: all-write, A, B, and E.[2]

First, RocksDB-BlobFS, accessing a single Optane via BlobFS, delivers worse performance than RocksDB in most cases, even with a much smaller database. Then, we compare with KVell, which benefits from a shared-nothing design that partitions data across multiple disks, aggressive request batching, and elimination of sorting/compaction. Meanwhile, such a shared-nothing design with no logging creates challenges in handling transactions (which is not supported by the current KVell). As the 2TB database runs beyond the O-O capacity,



(a) Throughput
(b) Latency

Figure 13: Additional system comparison, 2TB database (RocksDB-BlobFS w. 250GB only)

here it runs on N1-N1. We test KVell with two batch size settings: "B=1" (lowest latency and throughput) and "match" (the smallest batch size that surpasses SpanDB's throughput).

With all-write, KVell cannot match SpanDB's throughput even at its largest batch size (64), where it suffers huge latency (average at nearly 5000μs). Batch size 1 delivers a throughput at 15.2% of SpanDB's S-O level, and an average latency at 2.17×. YCSB-A sees a similar contrast, though to a lesser degree. With the read-intensive YCSB-B, KVell slightly outperforms SpanDB N2-O in throughput with batch size at 3, but reports latency 4.17× higher, while batch size 1 loses in both throughput and latency. SpanDB also wins in scans (YCSB-E), producing a 1.4× throughput and 57% latency reduction compared against KVell at batch size 64.

CPU utilization. With the 50%-write YCSB workload, SpanDB's CPU utilization is 94.5%, while RocksDB's CPU utilization is only 63.7%. This is a direct consequence of spinning threads on cores, as required by SpanDB's polling-based I/O and asynchronous request processing: All workers are busy with the request processing and never sleep. Overall the system spends more time doing useful work: SpanDB delivers a 3× throughput improvement RocksDB in this experiment. Meanwhile, under light loads SpanDB can easily enable queue wait monitoring, with its head-server thread directing other internal threads to sleep when necessary.

5.4 Recovery

We also tested SpanDB's recovery by inserting system crashes at random time points in our experiments. Specifically, we verified that updates in a MemTable, which were persisted to WAL on SD before a crash, could be correctly recovered upon rebooting. Results show that SpanDB was successfully recovered in all cases. Regarding performance, both SpanDB and RocksDB achieve almost the same recovery speed, *e.g.*, 10.25s and 10.27s to recover a 4GB database, respectively. It is reasonable as our earlier results show SPDK and ext4 deliver similar performance for large, sequential reads.

6 Related Work

Tiered storage. Multiple systems leverage tiering techniques on heterogeneous devices, mainly developing general-purpose

[2] KVell's code base does not include YCSB-F, whose implementation was identical to YCSB-A according to the authors.

file systems, such as NVMFS [55], Strata [43], and Ziggurat [75], transparently operating across NVRAM, SSD, and HDD layers. SpanDB is similar in exploiting the low latency of fast devices and the high bandwidth/capacity of slower ones. Its major novelty, meanwhile, lies in its KV-specific optimizations, many above the block storage layer. Also, its design addresses performance constraints brought by high-end commodity SSDs (as well as the new SPDK interface), rather than NVRAM units often emulated in evaluation.

HiLSM [47] and MatrixKV [70] use hybrid storage devices for KV. However, they both only intend to use a small portion of a fast and expensive NVM device. In addition, they target byte-addressable NVM as the fast device, while SpanDB focuses on the efficient utilization of NVMe SSDs, which currently offer much wider commodity hardware choices and significantly lower cost.

Existing work has deployed LSM-tree based KV stores across multiple devices. For example, Mutant [71] ranks SST files by popularity and places them on different cloud storage devices. PrismDB [56] makes LSM-trees "read-aware" by pinning hot objects to fast devices. SpanDB is similar in placing the top-level SST files to fast devices, but significantly differs from them by focusing more on write processing (often harder to scale [13, 32]). To this end, it encompasses many new, NVMe-oriented optimizations such as leveraging SPDK, parallel logging, and adaptive flush/compaction.

KV stores optimizations for fast, homogeneous storage. Many recent KV systems target low-latency, non-volatile storage, mostly by designing novel data structures, such as UniKV [73], LSM-trie [67], SlimDB [58], FloDB [10], PebblesDB [57], KVell [46], and SplinterDB [15]. As WAL creates a major performance bottleneck, many of them turned off WAL in evaluation, while KVell completely removed the commit log. This may lead to data inconsistency and a lack of transaction support. SpanDB instead retains the data structure and semantics of the mainstream LSM-tree based design. Moreover, the above systems assume homogeneous deployment, while SpanDB promotes heterogeneous storage that supplements older, slower devices with small, high-end ones.

Several systems deploy hardware solutions. X-Engine [32, 74] leverages hardware acceleration such as FPGA-accelerated compaction. KVSSD [40] and PinK [35] further offload KV management to specialized hardware, which are not commercially available yet. SpanDB, on the other hand, does not require special hardware support.

Another group of work optimizes KV stores on persistent memory, including HiKV [68], NoveLSM [41], NVMRocks [38], Bullet [34], SLM-DB [39], and FlatStore [14]. All use emulators in implementation/evaluation except FlatStore, which uses Intel Optane DCPMM. While KV stores directly running on persistent memory have undeniable performance advantages, hardware cost and capacity limit remain practical issues. The 256GB Optane DCPMM cost 3.12× higher (per GB) than the O disk used in our tests, and 40.5× higher than N1. Also, they require more expensive processors. These systems, therefore, fit better read-intensive workloads with moderate dataset sizes. Our work targets large databases with substantial write traffic, and aims to deliver high performance while keeping the overall hardware cost low.

Also, FlashStore [19] uses flash as a cache for KV stores. MyNVM [21] reduces DRAM cache demand in MyRocks [24], building a second-layer cache on Optane SSD. SpanDB's SD, instead, is not designed to be a cache.

Logging optimizations. Wang et al. utilized NVM for enhanced scalability via distributed logging [64]. NV-Logging [33] proposes per-transaction logging to enable concurrent logging for multiple transactions. NVWAL [42] exploits NVM to speed up WAL writes in SQLite. Again the above studies adopt emulation, and though now commodity NVM products are available their cost remains high, as discussed earlier. SpanDB, instead, improves WAL write performance on widely adopted NVMe block devices.

Other related work. The Staged Event-Driven Architecture (SEDA) decomposes request processing into a sequence of stages and use queues to pipeline, parallelize, and coordinate their execution [65]. Similar ideas have been used in many systems, including DeepFlash [72] and ours.

There are many studies optimizing LSM-tree based KV stores, such as SILK [9] (I/O scheduling to reduce the interference between client and background tasks), Monkey [17] and ElasticBF [48] (adopting dynamic bloom filter sizes to minimize lookup cost), TRIAD [8] (exploring workload skewness to reduce flush/compaction overhead), WiscKey [51] (separating keys and values to speedup sequential/random accesses), and HashKV [13] (WiscKey optimization targeting update-intensive workloads). Our work is orthogonal and complementary to the above techniques.

7 Conclusion

In this work, we explored a "poor man's design" that deploys a small and expensive high-speed SSD at the most-needed locations of a KV store, while leaving the majority of data on larger, cheaper, and slower devices. Our results reveal that the mainstream LSM-tree based design can be significantly improved to take advantage of such partial hardware upgrade (while retaining the major data structures and algorithms, as well as many orthogonal optimizations).

Acknowledgment

We sincerely thank all anonymous reviewers for their insightful feedback and especially thank our shepherd Angelos Bilas for his guidance in our camera-ready preparation. We thank Sam H. Noh for helpful discussions during his visit to QCRI. We also thank Sen Zheng of Zhongjia IT, for his valuable technical support during the COVID-19 lockdown. This work was supported in part by National Nature Science Foundation of China through grant No. 61832011, 61772486, and 61802358.

References

[1] A Persistent Key-Value Store for Fast Storage Environments. https://rocksdb.org/. "[accessed-Sept-2020]".

[2] Benchmarking Apache Samza. https://engineering.linkedin.com/performance/benchmarking-apache-samza-12-million-messages-second-single-node. "[accessed-Sept-2020]".

[3] Group Commit for the Binary Log. https://mariadb.com/kb/en/group-commit-for-the-binary-log/. "[accessed-Sept-2020]".

[4] MySQL Reference Manual. https://dev.mysql.com/doc/refman/5.7/en/replication-options-binary-log.html#sysvar_binlog_order_commits. "[accessed-Sept-2020]".

[5] RocksDB on Steroids. https://www.i-programmer.info/news/84-database/8542-rocksdb-on-steroids.html. "[accessed-Sept-2020]".

[6] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: a Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013.

[7] Andrew Audibert. Scalable Metadata Service in Alluxio: Storing Billions of Files. https://www.alluxio.io/blog/scalable-metadata-service-in-alluxio-storing-billions-of-files/. "[accessed-Sept-2020]".

[8] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[9] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[10] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

[11] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, June 2013.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[13] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[14] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 20)*, 2020.

[15] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[16] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010.

[17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[18] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems. *Keynote from LADIS*, 2009.

[19] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. *Proc. VLDB Endow.*, September 2010.

[20] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[21] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[22] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. I/O Is Faster Than the CPU: Let's Partition Resources and Eliminate (Most) OS Abstractions. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, 2019.

[23] Facebook. Cassandra on RocksDB at Instagram. https://developers.facebook.com/videos/f8-2018/cassandra-on-rocksdb-at-instagram. "[accessed-Sept-2020]".

[24] Facebook. MyRocks. http://myrocks.io/. "[accessed-Sept-2020]".

[25] Facebook. Under the Hood: Building and Open-sourcing RocksDB. https://www.facebook.com/notes/facebook-engineering/under-the-hood-building-and-open-sourcing-rocksdb/10151822347683920/. "[accessed-Sept-2020]".

[26] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[27] Dieter Gawlick and David Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Eng. Bull.*, 1985.

[28] Sanjay Ghemawat and Jeff Dean. LevelDB, A Fast and Lightweight Key/Value Database Library by Google. https://github.com/google/leveldb, 2014. "[accessed-Sept-2020]".

[29] Shashank Gugnani, Xiaoyi Lu, and Dhabaleswar K Panda. Analyzing, Modeling, and Provisioning QoS for NVMe SSDs. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2018.

[30] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, 1987.

[31] Kyuhwa Han, Hyukjoong Kim, and Dongkun Shin. WAL-SSD: Address Remapping-Based Write-Ahead-Logging Solid-State Disks. *IEEE Transactions on Computers*, 2019.

[32] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, 2019.

[33] Jian Huang, Karsten Schwan, and Moinuddin K Qureshi. NVRAM-aware Logging in Transaction Systems. *Proceedings of the VLDB Endowment*, 2014.

[34] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[35] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, July 2020.

[36] Intel. Breakthrough Performance for Demanding Storage Workloads. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-905p-product-brief.pdf. "[accessed-Sept-2020]".

[37] Intel. SPDK: Storage Performance Development Kit. https://spdk.io/. "[accessed-Sept-2020]".

[38] Andrew Pavlo Jianhong Li and Siying Dong. NVM-Rocks: RocksDB on Non-Volatile Memory Systems. http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/, 2017. "[accessed-Sept-2020]".

[39] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-Ri Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[40] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel DG Lee. Towards Building a High-Performance, Scale-In Key-Value Storage System. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, 2019.

[41] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, July 2018.

[42] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-ahead Logging. *ACM SIGOPS Operating Systems Review*, 2016.

[43] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[44] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 2010.

[45] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[46] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[47] Wenjie Li, Dejun Jiang, Jin Xiong, and Yungang Bao. HiLSM: an LSM-based Key-Value Store for Hybrid NVM-SSD Storage Systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pages 208–216, 2020.

[48] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[49] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Towards Accurate and Fast Evaluation of Multi-stage Log-structured Designs. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[50] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, February 2019.

[51] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. *ACM Transactions on Storage (TOS)*, 2017.

[52] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Transactions on Database Systems (TODS)*, 1992.

[53] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[54] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage Management in the NVRAM Era. *Proceedings of the VLDB Endowment*, 2013.

[55] S. Qiu and A. L. Narasimha Reddy. NVMFS: A Hybrid File System for Improving Random Write in NAND-flash SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, 2013.

[56] Ashwini Raina, Asaf Cidon, Kyle Jamieson, and Michael J. Freedman. PrismDB: Read-aware Log-structured Merge Trees for Heterogeneous Storage. https://arxiv.org/abs/2008.02352, 2020. arXiv.

[57] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[58] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. SlimDB: A Space-Efficient Key-Value Storage Engine for Semi-Sorted Data. *Proceedings of the VLDB Endowment*, 2017.

[59] Samsung. Ultra-Low Latency with Samsung Z-NAND SSD . https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf. "[accessed-Sept-2020]".

[60] Dong Siying. Workload Diversity with RocksDB. http://www.hpts.ws/papers/2017/hpts2017_rocksdb.pdf, 2017. "[accessed-Sept-2020]".

[61] SPDK. BlobFS (Blobstore Filesystem) - BlobFS Getting Started Guide - RocksDB Integration. https://spdk.io/doc/blobfs.html. "[accessed-Sept-2020]".

[62] 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. *ACM SIGARCH Computer Architecture News*, 2015.

[63] Toshiba. Toshiba Memory Introduces XL-FLASH Storage Class Memory Solution. https://business.kioxia.com/en-us/news/2019/memory-20190805-1.html. "[accessed-Sept-2020]".

[64] Tianzheng Wang and Ryan Johnson. Scalable Logging through Emerging Non-Volatile Memory. *Proceedings of the VLDB Endowment*, 2014.

[65] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, 2001.

[66] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an Unwritten Contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, July 2019.

[67] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015.

[68] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[69] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017.

[70] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31, 2020.

[71] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, 2018.

[72] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable Parallel Flash Firmware for Many-core Architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, February 2020.

[73] Qiang Zhang, Yongkun Li, Patrick P. C. Lee, Yinlong Xu, Qiu Cui, and Liu Tang. UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing. In *Proceedings of the 36th IEEE International Conference on Data Engineering (ICDE 2020)*, 2020.

[74] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *18th USENIX Conference on File and Storage Technologies FAST 20)*, 2020.

[75] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, February 2019.

[76] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience

Siying Dong[†], Andrew Kryczka[†], Yanqin Jin[†] and Michael Stumm[‡]

[†]Facebook Inc., 1 Hacker Way, Menlo Park, CA, U.S.A
[‡]University of Toronto, Toronto, Canada

Abstract

RocksDB is a key-value store targeting large-scale distributed systems and optimized for Solid State Drives (SSDs). This paper describes how our priorities in developing RocksDB have evolved over the last eight years. The evolution is the result both of hardware trends and of extensive experience running RocksDB at scale in production at a number of organizations. We describe how and why RocksDB's resource optimization target migrated from write amplification, to space amplification, to CPU utilization. Lessons from running large-scale applications taught us that resource allocation needs to be managed across different RocksDB instances, that data format needs to remain backward and forward compatible to allow incremental software rollout, and that appropriate support for database replication and backups are needed. Lessons from failure handling taught us that data corruption errors needed to be detected earlier and at every layer of the system.

1 Introduction

RocksDB [19, 54] is a high-performance persistent key-value storage engine created in 2012 by Facebook, based on Google's LevelDB code base [22]. It is optimized for the specific characteristics of Solid State Drives (SSDs), targets large-scale (distributed) applications, and is designed as a library component that is embedded in higher-level applications. As such, each RocksDB instance manages data on storage devices of just a single server node; it does not handle any inter-host operations, such as replication and load balancing, and it does not perform high-level operations, such as checkpoints — it leaves the implementation of these operations to the application, but provides appropriate support so they can do it effectively.

RocksDB and its various components are highly customizable, allowing the storage engine to be tailored to a wide spectrum of requirements and workloads; customizations can include the write-ahead log (WAL) treatment, the compression strategy, and the compaction strategy (a process that removes dead data and optimizes LSM-trees as described in §2). RocksDB may be tuned for high write throughput or high read throughput, for space efficiency, or something in between. Due to its configurability, RocksDB is used by many applications, representing a wide range of use cases. At Facebook alone, RocksDB is used by over 30 different applications, in aggregate storing many hundreds of petabytes of production data. Besides being used as a storage engine for ***databases*** (e.g., MySQL [37], Rocksandra [6], CockroachDB [64], MongoDB [40], and TiDB [27]), RocksDB is also used for the following types of services with highly disparate characteristics (summarized in Table 1):

- ***Stream processing***: RocksDB is used to store staging data in Apache Flink [12], Kafka Stream [31], Samza [43], and Facebook's Stylus [15].
- ***Logging/queuing services***: RocksDB is used by Facebook's LogDevice [5] (that uses both SSDs and HDDs), Uber's Cherami [8], and Iron.io [29].
- ***Index services***: RocksDB is used by Facebook's Dragon [59] and Rockset [58].
- ***Caching on SSD***: In-memory caching services, such as Netflix's EVCache [7], Qihoo's Pika [51] and Redis [46], store data evicted from DRAM on SSDs using RocksDB.

A prior paper presented an analysis of several database applications using RocksDB [11]. Table 2 summarizes some of the key system metrics obtained from production workloads.

Having a storage engine that can support many different use cases offers the advantage that the same storage engine can be used across different applications. Indeed, having each application build its own storage subsystem is problematic, as doing so is challenging. Even simple applications need to protect against media corruption using checksums, guarantee data consistency after crashes, issue the right system calls in the correct order to guarantee durability of writes, and handle errors returned from the file system in a correct manner. A well-established common storage engine can deliver sophistication in all those domains.

Additional benefits are achieved from having a common storage engine when the client applications run within a com-

	Read/Write	Read Types	Special Characteristics
Databases	Mixed	Get + Iterator	Transactions and backups
Stream Processing	Write-Heavy	Get or Iterator	Time window and checkpoints
Logging / Queues	Write-Heavy	Iterator	Support HDD too
Index Services	Read-Heavy	Iterator	Bulk loading
Cache	Write-Heavy	Get	Can drop data

Table 1: RocksDB use cases and their workload characteristics

	CPU	Space Util	Flash Endurance	Read Bandwidth
Stream Processing	11%	48%	16%	1.6%
Logging / Queues	46%	45%	7%	1.0%
Index Services	47%	61%	5%	10.0%
Cache	3%	78%	74%	3.5%

Table 2: System metrics for a typical use case from each application category.

mon infrastructure: the monitoring framework, performance profiling facilities, and debugging tools can all be shared. For example, different application owners within a company can take advantage of the same internal framework that reports statistics to the same dashboard, monitor the system using the same tools, and manage RocksDB using the same embedded admin service. This consolidation not only allows expertise to be easily reused among different teams, but also allows information to be aggregated to common portals and encourages developing tools to manage them.

Given the diverse set of applications that have adopted RocksDB, it is natural that priorities for its development have evolved. This paper describes how our priorities evolved over the last eight years as we learned practical lessons from real-world applications (both within Facebook and other organizations) and observed changes in hardware trends, causing us to revisit some of our early assumptions. We also describe our RocksDB development priorities for the near future.

§2 provides background on SSDs and Log-Structured Merge (LSM) trees [45]. From the beginning, RocksDB chose the LSM tree as its primary data structure to address the asymmetry in read/write performance and the limited endurance of flash-based SSDs. We believe LSM-trees have served RocksDB well and argue they will remain a good fit even with upcoming hardware trends (§3). The LSM-tree data structure is one of the reasons RocksDB can accommodate different types applications with disparate requirements.

§3 describes how our primary optimization target shifted from minimizing write amplification to minimizing space amplification, and from optimizing performance to optimizing efficiency.

§4 describes lessons we learned serving large-scale distributed systems; for example: (*i*) resource allocation must be managed across multiple RocksDB instances, since a single server may host multiple instances; (*ii*) data format must be backward and forward compatible, since RocksDB software updates are deployed/rolled-back incrementally; and (*iii*) proper support for database replication and backups are important.

§5 describes our experiences on failure handling. Large-scale distributed systems typically use replication for fault tolerance and high availability. However, single node failures must be properly handled to achieve that goal. We have found that simply identifying and propagating file system and checksum errors is not sufficient. Rather, faults (such as bitflips) at every layer must be identified as early as possible and applications should be able to specify policies for handling them in an automated way when possible.

§6 presents our thoughts on improving the key-value interface. While the core interface is simple and powerful given its flexibility, it limits the performance for some critical use cases. We describe our support for user-defined timestamps separate from the key and value.

§8 lists several areas where RockDB would benefit from future research.

2 Background

The characteristics of flash have profoundly impacted the design of RocksDB. The asymmetry in read/write performance and limited endurance pose challenges and opportunities in the design of data structures and system architectures. As such, RocksDB employs flash-friendly data structures and optimizes for modern hardware.

2.1 Embedded storage on flash based SSDs

Over the last decade, we have witnessed the proliferation of flash-based SSD for serving online data. The low latency and high throughput device not only challenged software to take advantage of its full capabilities, but also transformed how many stateful services are implemented. An SSD offers hundreds of thousands of Input/Output Operations per Second (IOPS) for both of read and write, which is thousands of times faster than a spinning hard drive. It can also support hundreds of MBs of bandwidth. Yet high write bandwidth cannot be sustained due to a limited number of program/erase cycles. These factors provide an opportunity to rethink the storage engine's data structures to optimize for this hardware.

The high performance of the SSD, in many cases, also shifted the performance bottleneck from device I/O to the network for both of latency and throughput. It became more attractive for applications to design their architecture to store data on local SSDs rather than use a remote data storage service. This increased the demand for a key-value store engines that are embedded in applications.

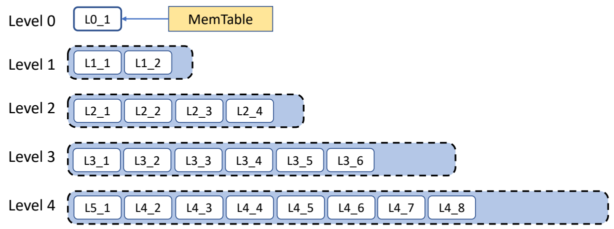


Figure 1: RocksDB LSM-tree using leveled compaction. Each white box is an SSTable.

RocksDB was created to address these requirements. We wanted to create a flexible key-value store to serve a wide range of applications using local SSD drives while optimizing for the characteristics of SSDs. LSM trees played a key role in achieving these goals.

2.2 RocksDB architecture

RocksDB uses Log-Structured Merge (LSM) trees [45] as its primary data structure to store data.

Writes. Whenever data is written to RocksDB, it is added to an in-memory write buffer called *MemTable*, as well as an on-disk Write Ahead Log (WAL). Memtable is implemented as a skiplist so keep the data ordered with O(log n) insert and search overhead. The WAL is used for recovery after a failure, but is not mandatory. Once the size of the MemTable reaches a configured size, then (*i*) the MemTable and WAL become immutable, (*ii*) a new MemTable and WAL are allocated for subsequent writes, (*iii*) the contents of the MemTable are *flushed* to a "Sorted String Table" (SSTable) data file on disk, and (*iv*) the flushed MemTable and associated WAL are discarded. Each SSTable stores data in sorted order, divided into uniformly-sized blocks. Each SSTable also has an index block with one index entry per SSTable block for binary search.

Compaction. The LSM tree has multiple levels of SSTables, as shown in Fig. 1. The newest SSTables are created by MemTable flushes and placed in Level-0. Levels higher than Level-0 are created by a process called *compaction*. The size of SSTables on a given level are limited by configuration parameters. When level-L's size target is exceeded, some SSTables in level-L are selected and merged with the overlapping SSTables in level-(L+1). In doing so, deleted and overwritten data is removed, and the table is optimized for read performance and space efficiency. This process gradually migrates written data from Level-0 to the last level. Compaction I/O is efficient as it can be parallelized and only involves bulk reads and writes of entire files.

Level-0 SSTables have overlapping key ranges, as each SSTable covers a full sorted run. Later levels each contain only one sorted run so the SSTables in these levels contain a partition of their level's sorted run.

Reads. In the read path, a key lookup occurs at each successive level until the key is found or it is determined that the key is not present in the last level. It begins by searching all MemTables, followed by all Level-0 SSTables, and then the SSTables in successively higher levels. At each of these levels, binary search is used. Bloom filters are used to eliminate an unnecessary search within an SSTable file. Scans require that all levels be searched.

Compaction	Write Amplification	Max Space Overhead	Avg Space Overhead	#I/O per Get() with bloom filter	# I/O per Get() without filter	# I/O per iterator seek
Leveled	16.07	9.8%	9.5%	0.99	1.7	1.84
Tiered	4.8	94.4%	45.5%	1.03	3.39	4.80
FIFO	2.14	N/A	N/A	1.16	528	967

Table 3: Write amplification, overhead and read I/O for three major compaction types under RocksDB 5.9. Number of sorted runs is set to 12 for Tiered Compaction, and 20 bloom filter bits per key are used for FIFO Compaction. Direct I/O is used and block cache size is set to be 10% of fully compacted DB size. Write amplification is calculated as total SSTable file writes vs number of Mem-Table bytes flushed. WAL writes are not included.

RocksDB supports multiple different types of compaction. *Leveled Compaction* was adapted from LevelDB and then improved [19]. In this compaction style, levels are assigned exponentially increasing size targets as exemplified by the dashed boxes in Fig. 1. *Tiered Compaction* (called *Universal Compaction* in RocksDB) is similar to what is used by Apache Cassandra or HBase. Multiple sorted runs are lazily compacted together, either when there are too many sorted runs, or the ratio between total DB size over the size of the largest sorted run exceeds a configurable threshold. Finally, *FIFO Compaction* simply discards old files once the DB hits a size limit and only performs lightweight compactions. It targets in-memory caching applications.

Being able to configure the type of compaction allows RocksDB to serve a wide range of use cases. By using different compaction styles, RocksDB can be configured as read friendly, write friendly, or very write friendly for special cache workloads. However, application owners will need to consider trade-offs among the different metrics for their specific use case [2]. A lazier compaction algorithm improves write amplification and write throughput, but read performance suffers, while a more aggressive compaction sacrifices write performance but allows for faster reads. Services like logging or stream processing can use a write heavy setup while database services need a balanced approach. Table 3 depicts this flexibility by way of micro-benchmark results.

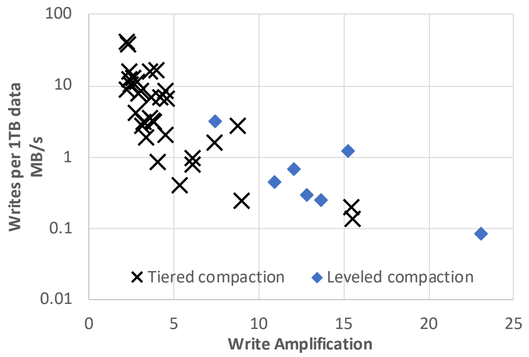


Figure 2: Survey of write amplification and write rate across 42 randomly sampled ZippyDB and MyRocks applications.

3 Evolution of resource optimization targets

Here we describe how our resource optimization target evolved over time: from write amplification to space amplification to CPU utilization.

Write amplification

When we started developing RocksDB, we initially focused on saving flash erase cycles and thus write amplification, following the general view of the community at the time (e.g., [34]). This was rightly an important target for many applications, in particular for those with write-heavy workloads (Table 1) where it continues to be an issue.

Write amplification emerges at two levels. SSDs themselves introduce write amplification: by our observations between 1.1 and 3. Storage and database software also generate write amplification; this can sometimes be as high as 100 (e.g., when an entire 4KB/8KB/16KB page is written out for changes of less than 100 bytes).

Leveled Compaction in RocksDB usually exhibits write amplification between 10 and 30, which is several times better than when using B-trees in many cases. For example, when running LinkBench on MySQL, RocksDB issues only 5% as many writes per transaction as InnoDB, a B-tree based storage engine [37]. Still, write amplification in the 10–30 range is too high for write-heavy applications. For this reason we added Tiered Compaction, which brings write amplification down to the 4–10 range, although with lower read performance; see Table 3. Figure 2 depicts RocksDB's write amplification under different data ingestion rates. RocksDB application owners often pick a compaction method to reduce write amplification when the write rate is high, and compact more aggressively when the write rate is low to achieve space efficiency and read performance goals.

Space amplification

After several years of development, we observed that for most applications, space utilization was far more important than write amplification, given that neither flash write cycles nor write overhead were constraining. In fact the number of IOPS utilized in practice was low compared to what the SSD could provide (yet still high enough to make HDDs unattractive, even when ignoring maintenance overhead). As a result, we shifted our resource optimization target to disk space.

	Dynamic Leveled Compaction			LevelDB-style Compaction		
# keys (millions)	Fully compacted size (GB)	Steady DB size (GB)	Space overhead(%)	Fully compacted size (GB)	Steady DB size (GB)	Space overhead(%)
200	12.0	13.5	12.4	12.0	15.1	25.6
400	24.0	26.9	11.8	24.0	26.9	12.2
600	36.0	40.4	12.2	36.4	42.5	16.9
800	48.0	54.2	12.7	48.3	57.9	19.7
1,000	60.1	67.5	12.4	60.3	73.8	22.4

Table 4: RocksDB space efficiency measured in a microbenchmark: data is prepopulated and each write is to a key chosen randomly from the pre-populated key space. RocksDB 5.9 with all default options. Constant 2MB/s write rate.

Fortunately, LSM-trees also work well when optimizing for disk space due to their non-fragmented data layout. However, we saw an opportunity to improve Leveled Compaction by reducing the amount of dead data (i.e., deleted and overwritten data) in the LSM tree. We developed *Dynamic Leveled Compaction*, where the size of each level in the tree is automatically adjusted based on the actual size of the last level (instead of setting the size of each level statically) [19]. This method achieves better and more stable space efficiency than Leveled Compaction. Table 3 shows space efficiency measured in a random write benchmark: Dynamic Leveled Compaction limits its space overhead to 13%, while Leveled Compaction can add more than 25% . Moreover, space overhead in the worst case under Leveled Compaction can be as high as 90%, while it is stable for dynamic leveling. In fact, for UDB, one of Facebooks main databases, the space footprint was reduced to 50% when InnoDB was replaced by RocksDB [36].

CPU utilization

An issue of concern sometimes raised is that SSDs have become so fast that software is no longer able to take advantage of their full potential. That is, with SSDs, the bottleneck has shifted from the storage device to the CPU, so fundamental improvements to the software are necessary. We do not share this concern based on our experience, and we do not expect it to become an issue with future NAND flash based SSDs for two reasons. First, only a few applications are limited by the IOPS provided by the SSDs; as discussed in §4.2, most applications are limited by space.

Second, we find that any server with a high-end CPU has more than enough compute power to saturate one high-end SSD. RocksDB has never had an issue making full use of SSD performance in our environment. Of course, it is possible to configure a system that results in the CPU becoming a

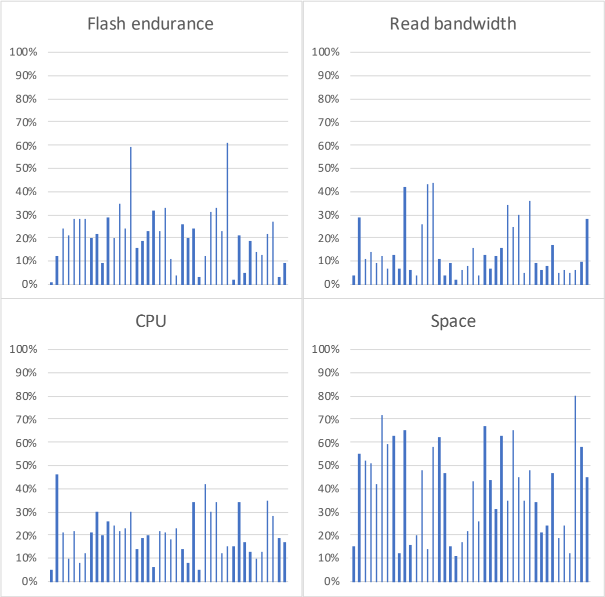


Figure 3: Resource utilization across four metrics. Each line represents a different deployment with a different workload. Measurements were taken over the course of one month. All numbers are the average across all hosts in the deployment. CPU and read bandwidth are for the highest hour during the month. Flash endurance and space utilization are average across the entire month.

bottleneck; e.g., a system with one CPU and multiple SSDs. However, effective systems are typically those configured to be well-balanced, which today's technology allows. Intensive write-dominated workloads may also cause the CPU to become a bottleneck. For some, this can be mitigated by configuring RocksDB to use a more lightweight compression option. For the other cases, the workload may simply not be suitable for SSDs since it would exceed the typical flash endurance budget that allows the SSD to last 2-5 years.

To confirm our view, we surveyed 42 different deployments of ZippyDB [65] and MyRocks in production, each serving a different application. Fig. 3 shows the result. Most of the workloads are space constrained. Some are indeed CPU heavy, but hosts are generally not fully utilized to leave headroom for growth and handling data center or region-level failures (or because of misconfigurations). Most of these deployments include hundreds of hosts, so averages give an idea of the resource needs for these use cases, considering that workloads can be freely (re-)balanced among those hosts (§4).

Nevertheless, reducing CPU overheads has become an important optimization target, given that the low hanging fruit of reducing space amplification has been harvested. Reducing CPU overheads improves the performance of the few applications where the CPU is indeed constraining. More importantly, optimizations that reduce CPU overheads allow for hardware configurations that are more cost-effective — until several years ago, the price of CPUs and memory was reasonably low relative to SSDs, but CPU and memory prices have increased substantially, so decreasing CPU overhead and memory usage has increased in importance. Early efforts to lower CPU overhead included the introduction of prefix bloom filters, applying the bloom filter before index lookups, and other bloom filter improvements. There remains room for further improvement.

Adapting to newer technologies

New architectural improvements related to SSDs could easily disrupt RocksDB's relevancy. For example, open-channel SSDs [50, 66], multi-stream SSD [68] and ZNS [4] promise to improve query latency and save flash erase cycles. However, these new technologies would benefit only a minority of the applications using RocksDB, given that most applications are space constrained, not erase cycle or latency constrained. Further, having RocksDB accommodate these technologies directly would challenge the unified RocksDB experience. One possible path worth exploring would be to delegate the accommodation of these technologies to the underlying file system, perhaps with RocksDB providing additional hints.

In-storage computing potentially might offer significant gains, but it is unclear how many RocksDB applications would actually benefit from this. We suspect it would be challenging for RocksDB to adapt to in-storage computing, likely requiring API changes to the entire software stack to fully exploit. We look forward to future research on how best to do this.

Disaggregated (remote) storage appears to be a much more interesting optimization target and is a current priority. So far, our optimizations have assumed the flash was locally attached, as our system infrastructure is primarily configured this way.However, faster networks currently allow many more I/Os to be served remotely, so the performance of running RocksDB with remote storage has become viable for an increasing number of applications. With remote storage, it is easier to make full use of both CPU and SSD resources at the same time, because they can be separately provisioned on demand (something much more difficult to achieve with locally attached SSDs). As a result, optimizing RocksDB for remote flash storage has become a priority. We are currently addressing the challenge of long I/O latency by trying to consolidate and parallelize I/Os. We have adapted RocksDB to handle transient failures, pass QoS requirements to underlying systems, and report profiling information. However, more work is needed.

Storage Class Memory (SCM) is a promising technology. We are investigating how best to take advantage of it. Several possibilities are worth considering: 1. use SCM as an extension of DRAM — this raises the questions of how best to implement key data structures (e.g., block cache or memtable) with mixed DRAM and SCM, and what overheads are introduced when trying to exploit the offered persistency; 2. use SCM as the main storage of the database, but we note that RocksDB tends to be bottlenecked by space or CPU, rather than I/O; and 3. use SCM for the WALs, but this raises the question of whether this use case alone justifies the costs of SCM, considering that we only need a small staging area before it is moved to SSD.

Main Data Structure Revisited

We continuously revisit the question of whether LSM-trees remain appropriate, but continue to come to the conclusion that they do. The price of SSDs hasn't dropped enough to change the space and flash endurance bottlenecks for most use cases and the alternative of trading SSD usage with CPU or DRAM only makes sense for a few applications. While the main conclusion remains the same, we frequently hear users' demands for write amplification lower than what RocksDB can provide. Nevertheless, we noted that when object sizes are large, write amplification can be reduced by separating key and value (e.g. WiscKey [35] and ForrestDB [1]), so we are adding this to RocksDB (called BlobDB).

4 Lessons on serving large-scale systems

RocksDB is a building block for a wide variety of large-scale distributed systems with disparate requirements. Over time, we learned that improvements were needed with respect to resource management, WAL treatment, batched file deletions, data format compatibility, and configuration management.

Resource management

Large-scale distributed data services typically partition the data into *shards* that are distributed across multiple server nodes for storage. The size of shards is limited, because a shard is the unit for load balancing and replication, and because shards are copied between nodes atomically for this purpose. As a result, each server node will typically host tens or hundreds of shards. In our context, a separate RocksDB instance is used to service each shard, which means that a storage host will have many RocksDB instances running on it. These instances can either all run in one single address space, or each in its own address space.

The fact that a host may run many RocksDB instances has implications on resource management. Given that the instances share the host's resources, the resources need to be managed both globally (per host) and locally (per instance) to ensure they are used fairly and efficiently. When running in single process mode, having global resource limits is important, including for (1) memory for write buffer and block cache, (2) compaction I/O bandwidth, (3) compaction threads, (4) total disk usage and (5) file deletion rate (described below), and such limits are potentially needed on a per-I/O device basis. Local resource limits are also needed, for example to ensure that a single instance cannot utilize an excessive amount of any resource. RocksDB allows applications to create one or more resource controllers (implemented as C++ objects passed to different DB objects) for each type of resource and also do so on a per instance basis. Finally, it is important to support prioritization among RocksDB instances to make sure a resource is prioritized for the instances that need it most.

Another lesson learned when running multiple instances in one process: liberally spawning unpooled threads can be problematic, especially if the threads are long-lived. Having too many threads increases the probability of CPU, causes excessive context switching overhead, and makes debugging extremely difficult, and I/O spikes. If a RocksDB instance needs to perform some work using a thread that may go to sleep or wait on a condition, then it is better to use a thread pool where size and resource usage can be easily capped.

Global (per host) resource management is more challenging when the RocksDB instances run in separate processes, given that each shard only has local information. Two strategies can be applied. First, each instance is configured to use resources conservatively, as opposed to greedily. With compaction, for example, each instance can initiate fewer compactions than "normal," ramping up only when compactions are behind. The downside of this strategy is that the global resources may not be fully exploited, leading to sub-optimal resource usage. The second, operationally more challenging strategy is for the instances to share resource usage information amongst themselves and to adapt accordingly in an attempt to optimize resource usage more globally. More work will be needed to improve host-wide resource management in RocksDB.

WAL treatment

Traditional databases tend to force a write-ahead-log (WAL) write upon every write operation to ensure durability. In contrast, large-scale distributed storage systems typically replicate data for performance and availability, and they do so with various consistency guarantees. For example, if copies of the same data exist in multiple replicas, and one replica becomes corrupted or inaccessible, then the storage system uses valid replica(s) from other unaffected hosts to rebuild the replica of the failed host. For such systems, RocksDB WAL writes are less critical. Further, distributed systems often have their own replication logs (e.g., Paxos logs), in which case RocksDB WAL are not needed at all.

We learned it is helpful to provide options for tuning WAL sync behavior to meet the needs of different applications. Specifically, we introduced differentiated WAL operating modes: (*i*) synchronous WAL writes, (*i*) buffered WAL writes, and (*i*) no WAL writes at all. For buffered WAL treatment, WAL is periodically written out to disk in the background at low priority so as not to impact RocksDB's traffic latencies.

Rate-limited file deletions

RocksDB typically interacts with the underlying storage device via a file system. These file systems are flash-SSD-aware; e.g., XFS, with realtime discard, may issue a TRIM command [28] to the SSD whenever a file is deleted. TRIM commands are commonly believed to improve performance and flash endurance [21], as validated by our production experience. However, it may also cause performance issues. TRIM is more disruptive than we originally thought: in addition to updating the address mapping (most often in the SSD's internal memory), the SSD firmware also needs to write these changes to FTL's[1] journal in flash, which in turn may trigger SSD's internal garbage collection, causing considerable data movement with an attendant negative impact on foreground IO latencies. To avoid TRIM activity spikes and associated increases in I/O latency, we introduced rate limiting for file deletion to prevent multiple files from being deleted simultaneously (which happens after compactions).

Data format compatibility

Large scale distributed applications run their services on many hosts, and they expect zero downtime. As a result, software upgrades are incrementally rolled out across the hosts; and when issues arise, the updates are rolled back. In light of continuous deployment [56], these software upgrades occur frequently; RocksDB issues a new release once a month.For this reason, it is important that the data on disk remain both backward and forward compatible across the different software versions. A newly upgraded (or rolled back) RocksDB instance must be able to make sense of the data stored on disk by the previous instance. Further, RocksDB data files may need to be copied between distributed instances for replica building or load balancing, and these instances may be running different versions. A lack of a forward compatibility guarantee caused operational difficulties in some RocksDB deployments, which led us to add the guarantee.

RocksDB goes to great lengths to ensure data remains both forward and backward compatible (except for new features). This is challenging both technically and process-wise, but we have found the effort pays off. For backwards compatibility, RocksDB must be able to understand all formats previously written to disk; this adds software and maintenance complexities. For forward compatibility, future data formats need to be understood, and we aim to maintain forward compatibility for at least one year. This can be achieved in part, by using generic techniques, such as those used by Protocol Buffer [63] or Thrift [62]. For configuration file entries, RocksDB needs to be able to identify new fields and use best-effort guesses on how to apply the configuration or when to discard. We continuously test different versions of RocksDB with different versions of its data.

Managing configurations

RocksDB is highly configurable so that applications can optimize for their workload. However, we have found configuration management to be a challenge. Initially, RocksDB inherited LevelDB's method of configuring parameters where the parameter options were directly embedded in the code. This caused two problems. First, parameter options were often tied to the data stored on disk, causing potential compatibility issues when data files created using one option could not be opened by a RocksDB instance newly configured with another option. Second, configuration options not explicitly specified by the code were automatically set to RocksDB's default values. When a RocksDB software update included changes to the default configuration parameters (e.g., to increase memory usage or compaction parallelism), applications would sometimes experience unexpected consequences.

Config Area:	Compaction	I/O	Compression	SSTable file	Plug-in fcts
Configurations:	14	4	2	7	6

Table 5: The number of distinct configurations used across 39 ZippyDB deployments

To address these issues, RocksDB first introduced the ability for a RocksDB instance to open a database with a string parameter that included configuration options. Later RocksDB introduced support for optionally storing an options file along with the database. We also introduced two tools: (*i*) a validation tool that validates whether the options for opening a database was compatible with the target database; and (*ii*) a migration tool rewrites a database to be compatible with the desired options (although this tool is limited).

A more serious problem with RocksDB configuration management is the large number of configuration options. In the early years of RocksDB, we made the design choice of supporting a high degree of customization: we introduced many new knobs, and introduced the support of pluggable components, all to allow applications to realize their performance potential. This proved to be a successful strategy for gaining initial traction early on. However, a common complaint now is that there are far too many options and that it is too difficult to understand their effects; i.e., it has become very difficult to specify an "optimal" configuration.

More daunting beyond having many configuration parameters to tune is the fact that the optimal configuration depends not just on the system that has RocksDB embedded, but also on the workload generated by the applications above them. Consider, for example, ZippyDB [65], an in-house developed, large-scale distributed key-value store that uses RocksDB on its nodes. ZippyDB serves numerous different applications, sometimes individually, sometimes in a multi-tenant setup. Although significant efforts go into using uniform configurations across all ZippyDB use cases wherever possible, the workloads are so different for the different use cases, a uniform configuration is not practically feasible when performance is important. Table 5 shows that across the 39 ZippyDB deployments we sampled, over 25 distinct configurations.

Tuning configuration parameters is also particularly challenging for systems with embedded RocksDB that are shipped to third parties. Consider a third party using a database such as MySQL or ZippyDB within one of their applications. The third party will typically know very little about RocksDB and how it is best tuned. And the database owners have little appetite for tuning the systems of their clients.

[1] FTL: Flash Translation Layer.

These real-world lessons triggered changes in our configuration support strategy. We have spent considerable effort on improving out-of-box performance and simplifying configurations. Our current focus is on providing *automatic adaptivity*, while continuing to support extensive explicit configuration, given that RocksDB continues to server specialized applications. We note that pursuing adaptivity while retaining explicit configurability creates significant code maintenance overhead, we believe the benefits of having a consolidated storage engine outweighs the code complexity.

Replication and backup support
RocksDB is a single node library. The applications that use RocksDB are responsible for replication and backups if needed. Each application implements these functions in its own way (for legitimate reasons), so it is important that RocksDB offer appropriate support these functions.

Bootstrapping a new replica by copying all the data from an existing one can be done in two ways. First, all the keys can be read from a source replica and then written to the destination replica (*logical copying*). On the source side, RocksDB supports data scanning operations by offering the ability to minimize the impact on concurrent online queries; e.g., by providing the option to not cache the result of these operations and thus prevent cache trashing. On the destination side, bulk loading is supported and also optimized for this scenario.

Second, bootstrapping a new replica can be done by copying SSTables and other files directly (*physical copying*). RocksDB assists physical copying by identifying existing database files at a current point in time, and preventing them from being deleted or mutated. Supporting physical copying is an important reason RocksDB stores data on an underlying file system, as it allows each application to use its own tools. We believe the potential performance gains of RocksDB directly using a block device interface or heavily integrating with FTL does not outweigh the aforementioned benefit.

Backup is an important feature for most databases and other applications. For backups, applications have the same logical vs. physical choice as with replication. One difference between backups and replication is that applications often need to manage multiple backups. While most applications implement their own backups (to accommodate their own requirements), RocksDB provides a backup engine for applications to use if their backup requirements are simple.

We see two areas for further improvement in this area, but both require changes to the key-value API; they are discussed in §6. The first involves applying updates in a consistent order on different replicas, which introduces performance challenges. The second involves performance issues surrounding write requests that are issued one at a time and the fact that replicas can fall behind and applications may wish these replicas to catch up faster. Various solutions have been implemented by different applications to address these issues, but they all have limitations [20]. The challenge is that applications cannot issue writes out of order and do snapshot reads with their own sequence numbers because RocksDB does not currently support multi-versioning with user defined timestamps.

5 Lessons on failure handling

Through production experience, we have learned three major lessons about failure handling. First, data corruption needs to be detected early to minimize the risk of data unavailability or loss, and in doing so to pinpoint where the error originated. Second, integrity protection must cover the entire system to prevent silent corruptions from being exposed to RocksDB clients or spreading to other replicas (see Fig. 4). Third, errors need to be treated in a differentiated manner.

Frequency of silent corruptions
RocksDB users do not usually use data protection by SSD (e.g. DIF/DIX) for performance reason, and storage media corruptions are detected by RocksDB block checksums, which is a routine feature for all mature databases so we skip the analysis here. CPU/memory corruption does happen rarely and it is difficult to accurately quantify. Applications that use RocksDB often run data consistency checks that compare replicas for integrity. This catches errors, but those could have been introduced either by RocksDB or by the client application (e.g., when replicating, backing up, or restoring data).

We found that the frequency of corruptions introduced at the RocksDB level can be estimated by comparing primary and secondary indexes in MyRocks database tables that have both; any inconsistencies whould have been introduced at the RocksDB level, including CPU or memory corruptions. Based on our measurements, corruptions are introduced at the RocksDB level roughly once every three months for each 100PB of data. Worse, in 40% of those cases, the corruption had already propagated to other replicas.

Data corruptions also occur when transferring data, often because of software bugs. For example, a bug in the underlying storage system when handling network failures, caused us to see, over a period of time, roughly 17 checksum mismatches for every petabyte of physical data transferred.

Multi-layer protection
Data corruption needs to be detected as early as possible to minimize downtime and data loss. Most RocksDB applications have their data replicated on multiple hosts; when a checksum mismatch is detected, the corrupt replica is discarded and replaced with a correct one. However, this is a viable option only as long as a correct replica still exists.

Today, RocksDB checksums file data at multiple levels to identify corruption in the layers beneath it. These, as well as the planned application layer checksum, are shown in Fig. 4. Multiple levels of checksums are important, primarily because they help detect corruptions early and because they protect against different types of threats. Block checksums, inherited from LevelDB, prevent data corrupted at or below the file system from being exposed to the client. File checksums, added in 2020, protect against corruption caused by the underlying storage system from being propagated to other replicas and against corruption caused when transferring SSTable files over the wire. For WAL files, handoff checksums enable efficient early detection of corruptions at write time.

Block integrity. Each SSTable block or WAL fragment has a checksum attached to it, generated when the data is created. Unlike the file checksum that is verified only when the file is moved, this checksum is verified every time the data is read, due to its smaller scope. Doing so prevents data corrupted by the storage layer from being exposed to RocksDB clients.

File integrity. File contents are particularly at risk of being corrupted during transfer operations; e.g., for backups or when distributing SSTable files. To address this, SSTables are protected by their own checksum, generated when the table is created. An SSTable's checksum is recorded in the metadata's SSTable file entry, and is validated with the SSTable file wherever it is transferred. However, we note that other files, such as WAL files, are still not protected this way.

Handoff integrity. An established technique for detecting write corruptions early is to generate a handoff checksum on the data to be written to the underlying file system, and pass it down along with the data, where it is verified by the lower layers [48, 70]. We wish to protect WAL writes using such a write API, since unlike SSTables, WALs benefit from incremental validation on each append. Unfortunately, local file systems rarely support this — some, specialized stacks, such as Oracle ASM [49] do, however.

On the other hand, when running on remote storage, the write API can be changed to accept a checksum, hooking into the storage service's internal ECC. RocksDB can use checksum combining techniques on the existing WAL fragment checksums to efficiently compute a write handoff checksum. Since our storage service performs write-time verification, we expect it to be extremely infrequent for corruption detection to be delayed until read time.

End-to-end protection
While the layers of protection described above prevent clients from being exposed to corrupt data in many cases, they are not comprehensive. One deficiency of the protections mentioned so far is that data is unprotected above the file I/O layer; e.g., data in MemTable and the block cache. Data corrupted at this level will be undetectable and thus will eventually be exposed to the user. Further, flush or compaction operations can persist corrupted data, making the corruption permanent.

Key-value integrity. To address this problem, we are currently implementing per-key-value checksums to detect corruptions that occur above the file I/O layer. This checksum will be transferred along with the key/value wherever it is copied, although we will elide it from file data where alternative integrity protection already exists.

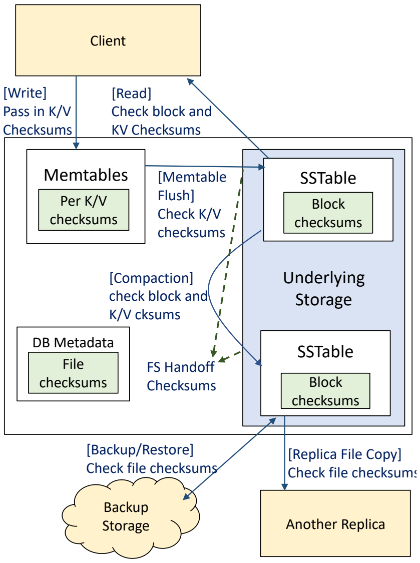


Figure 4: Four Types of Checksums

Severity-based error handling
Most of the faults RocksDB encounters are errors returned by the underlying storage system. These errors can stem from a multitude of issues, from severe problems like a read-only file system, to transient problems like a full disk or a network error accessing remote storage. Early on, RocksDB reacted to such issues either by simply returning error messages to the client or by permanently halting all write operations.

Today, we aim to interrupt RocksDB operations only if the error is not locally recoverable; e.g., transient network errors should not require user intervention to restart the RocksDB instance. To implement this, we improved RocksDB to periodically retry resume operations after encountering an error classified as transient. As a result, we obtain operational benefits as clients do not need to manually mitigate RocksDB for a significant portion of faults that occur.

6 Lessons on the key-value interface

The core key-value (KV) interface is surprisingly versatile. Almost all storage workloads can be served by a datastore with a KV API; we have rarely seen an application not able to implement functionality using this interface. That is perhaps why KV-stores are so popular. The KV interface is generic. Both keys and values are variable-length byte arrays. Applications have great flexibility in determining what information to pack into each key and value, and they can freely choose from a rich set of encoding schemes. Consequently, it is the application that is responsible for parsing and interpreting the keys and values. Another benefit of the KV interface is its portability. It is relatively easy to migrate from one key-value system to another. However, while many use cases achieve optimal performance with this simple interface, we have noticed that it can limit performance for some applications.

For example, building concurrency control outside of RocksDB is possible but hard to make efficient, especially if two-phase-commit needs to be supported where some data persistency is needed before committing the transaction. We added transaction support for this reason, which is used by MyRocks (MySQL+RocksDB). We continue to add features; e.g., gap/next key locking and large transactions support.

In other cases, the limitation is caused by the key-value interface itself. Accordingly, we have started to investigate possible extensions to the basic key-value interface. One such extension is support for user-defined timestamps.

Versions and timestamps

Over the last few years, we have come to understand the importance of data versioning. We have concluded that version information should become a first-class citizen in RocksDB, in order to properly support features, such as multi-version concurrency control (MVCC) and point-in-time reads. To achieve this, RocksDB needs to be capable of accessing different versions efficiently.

So far, RocksDB has internally been using 56-bit sequence numbers to identify different versions of KV-pairs. The sequence number is generated by RocksDB and incremented on every client write (hence, all data is logically arranged in sorted order). The client application cannot affect the sequence number. However RocksDB allows the application to take a *Snapshot* of the DB, after which RocksDB guarantees that all KV pairs that existed at the time of the snapshot will persist until the snapshot is explicitly released by application. As a result, multiple KV-pairs with the same key may co-exist, differentiated by their sequence numbers.

This approach to versioning is inadequate as it does not satisfy the requirements of many applications. To read from a past state, a snapshot must have already been taken at the previous point in time. RocksDB does not support taking a snapshot of the past, since there is no API to specify a time-point. Moreover, it is inefficient to support point-in-time reads. Finally, each RocksDB instance assigns its own sequence numbers and snapshots can be obtained only on a per instance basis. This complicates versioning for applications with multiple, (possibly replicated) shards, each of which is a RocksDB instance. In summary, it is essentially impossible to create versions of data that offer cross-shard consistent reads.

Applications can work around these limitations by encoding timestamps within the key or within the value. However, they will experience performance degradations in either case. Encoding within the key sacrifices performance for point-lookups, while encoding within the value sacrifices performance for out-of-order writes to the same key and complicates the reading of old versions of keys. We believe application-specified timestamps would better address these limitations, where the application can tag its data with timestamps that can be understood globally, and do so outside the key or value.

We have added basic support for application-specified timestamp and evaluated this approach with DB-Bench. The results are shown in Table 6. Each workload has two steps: the first step populates the database, and we measure performance during the second step. For example, in "`fill_seq + read_random`", we populate the initial database by writing a number of keys in ascending order, and in step 2 perform random read operations. Relative to the baseline, where the application encodes a timestamp as part of the key (transparent to RocksDB), the application-specified timestamp API can lead to a 1.2X or better throughput gain. The improvements arise from treating the timestamp as metadata separate from the user key, because then point lookups can be used instead of iterators to get the newest value for a key, and Bloom filters may identify SSTables not containing that key. Additionally, the timestamp range covered by an SSTable can be stored in its properties, which can be leveraged to exclude SSTables that could only contain stale values.

workload	throughput gain
fill_seq + read_random	1.2
fill_seq + read_while_writing	1.9
fill_random + read_random	1.9
fill_random +read_while_writing	2.0

Table 6: DB_bench microbenchmark using the timestamp API sees $\geq$ 1.2X throughput improvement.

We hope this feature will make it easier for users to implement multi-versioning in their systems for single node MVCC, distributed transactions, or resolving conflicts in multi-master replication. The more complicated API, however, is less straightforward to use and perhaps prone to misuse. Further, the database would consume more disk space than storing no timestamp, and would be less portable to other systems.

7 Related Work

Our work on RocksDB has benefited from a broad range of research in a number of areas.

Storage Engine Libraries

Many storage engine have been built as a library to be embedded in applications. RocksDB's KV interface is more primitive than, for example, BerkeleyDB [44], SQLite [47] and Hekaton [18]. Further, RocksDB differs from these systems by focusing on the performance of modern server workloads, which require high throughput and low latency, and typically run on high end SSDs and multicore CPUs. This differs from systems with more general targets, or built for faster storage media [18, 30].

Key-value stores for SSDs

Over the years, much effort has gone into optimizing key-value stores, especially for SSDs. As early as 2011, SILT [34] proposed a key-value store that balanced between memory efficiency, CPU, and performance. ForestDB[45] uses HB+

trees to index on top of logs. TokuDB [32] and other databases use FractalTree/Bε trees. LOCS [67], NoFTL-KV [66] and FlashKV [69] target Open-Channel SSDs for improved performance. While RocksDB benefited from these efforts, our position and strategy for improving performance is different and we continue to depend on LSM trees. Several studies have compared the performance of RocksDB with other databases such as InnoDB [41], TokuDB [19] [37], and WiredTiger [10].

LSM-tree improvements
Several systems also use LSM trees and improved their performance. Write amplifications is often the primary optimization goal; e.g., WiscKey [35], PebblesDB [52], IAM-tree [25] and TRIAD [3]. These systems go further in optimizing for write amplification than RocksDB which focuses more on tradeoffs among different metrics. SlimDB [53] optimized LSM trees for space efficiency; RocksDB also focuses on deleting dead data. Monkey [17] attempts to balance between DRAM and IOPs. bLSM [57], VT-tree [60] and cLSM [24] optimize for the general performance of LSM trees.

Large-scale storage systems
There are numerous distributed storage systems [13, 14, 16, 26, 38, 64]. They usually have complex architectures spanning multiple processes, hosts and data centers. They are not directly comparable to RocksDB, a storage engine library on a single node. Other systems (e.g.,MongoDB, MySQL [42], Microsoft SQL Server [38]) can use modular storage engines; they have addressed similar challenges to what RocksDB faces, including failure handling and using timestamps.

Failure handling. Checksums are frequently used to detect data corruption [9, 23, 42]. Our argument that we need both end-to-end and handoff checksums still mirrors the classic end-to-end argument [55] and is similar to the strategy used by others: [61], ZFS [71], Linux [48] and [70]. Our argument for earlier corruption detection is similar to [33] which argues that domain-specific checking is inadequate.

Timestamp support. Several storage systems provide timestamp support: HBase [26], WiredTiger [39] and BigTable [14]; Cassandra [13] supports a timestamp as an ordinary column. In these systems, timestamps are a count of the number of milliseconds since the UNIX epoch. Hekaton [18] uses a monotonically increasing counter to assign timestamps, which is similar to the RocksDB sequence number. RocksDB's ongoing work on a user timestamp can be complementary to the aforementioned efforts. We hope key-value APIs with a user-defined timestamp extension can make it easier for upper-level systems to support features related to data versioning with low overhead in both performance and efficiency.

8 Future Work and Open Questions

Besides completing the improvements mentioned above, including optimizing for dis-aggregated storage, key-value separation, multi-level checksums and application-specified timestamps, we plan to unify leveled and tiered compaction and improve adaptivity. However, a number of open questions could benefit from further research.

1. How can we use SSD/HDD hybrid storage to improve efficiency?
2. How can we mitigate the performance impact on readers when there are many consecutive deletion markers?
3. How should we improve our write throttling algorithms?
4. Can we develop an efficient way of comparing two replicas to ensure they contain the same data?
5. How can we best exploit SCM? Should we still use LSM tree and how to organize storage hierarchy?
6. Can there be a generic integrity API to handle data handoff between RocksDB and the file system layer?

9 Conclusions

RocksDB has grown from a key-value store serving niche applications to its current position of widespread adoption across numerous industrial large-scale distributed applications. The LSM tree as the main data structure has served RocksDB well, as it exhibits good write and space amplification. Our view on performance has, however, evolved over time. While write and space amplification remain the primary concern, additional focus has shifted to CPU and DRAM efficiency, as well as remote storage.

Lessons from running large-scale applications taught us that resource allocation needs to be managed across different RocksDB instances, that the data format needs to remain backward *and forward* compatible to allow incremental software deployments, that appropriate support for database replication and backups are needed, and that configuration management needs to be straightforward and preferably automated. Lessons from failure handling taught us that data corruption errors need to be detected earlier and at every layer of the system. The key-value interface enjoys great popularity for its simplicity with some limitations in performance. Some simple revisions to the interface might yield a better balance.

Acknowledgments

We attribute the success of RocksDB to all current and past RocksDB team members at Facebook, all those who made contributions in the open-source community, as well as RocksDB users. We especially thank Mark Callaghan, the mentor to the project for years, as well as Dhruba Borthakur, the lead founding member of RocksDB. We also appreciate comments to the paper by Jason Flinn and Mahesh Balakrishnan. Finally, we thank our shepherd, Ethan Miller, and the anonymous reviewers for their valuable feedback.

A RocksDB Feature Timeline

	Performance	Configurability	Features
2012	• Multi-threaded compactions		• Compaction filters • Locking SSTables from deletion
2103	• Tiered compaction • Prefix Bloom filter • Bloom Filter for MemTables • Separate thread pool for MemTable flush	• Pluggable MemTable • Pluggable file format	• Merge Operator
2014	• FIFO compaction • Compaction rate limiter • Cache-friendly Bloom filters	• String-based config options • Dynamic config changes	• Backup engine • Support for multiple key spaces ("column family") • Physical checkpoints
2015	• Dynamic leveled compaction • File deletion rate limiting • Parallel Level 0 and 1 compaction	• Separate config file • Config compatibility checker	• Bulk loading for SSTable file integration • Optimistic and pessimistic transactions
2016	• Different compression for last level • Parallel MemTable inserts	• MemTable total size caps across instances • Compaction migration tools	• DeleteRange()
2017	• Separate thread pool for bottom-most compactions • Two-level file indices • Level 0 to level 0 compactions	• Single memory limit for both block cache and MemTable	
2018	• Dictionary compression • Hash index into data blocks		• Automatic recovery from out-of-space errors • Query trace and replay tools
2019	• Batched MultiGet() with parallel I/O	• Configure plug-in function using object registry	• Secondary instance
2020	• Multithreaded single file compression		• Entire file checksum • Automatically recover from retriable errors • Partial support for user-defined timestamps

B Recap of lessons learned

Some of the lessons we learned include:

1. It's important that a storage engine can be tuned to fit different performance characteristics. (§1)
2. Space efficiency is the bottleneck for most applications using SSDs. (§3, Space amplification)
3. CPU overhead is becoming more important to allow systems to run more efficiently. (§3, CPU utilization)
4. Global, per host, resource management is necessary when many RocksDB instances run on the same host. (§4, Resource management)
5. Having WAL treatment be configurable (synchronous WAL writes, buffered WAL writes or disabled WAL) offers applications performance advantages. (§4, WAL treatment)
6. The SSD TRIM operation is good for performance but file deletions need to be rate limited to prevent occasional performance issues. (§4, Rate-limited file deletions)
7. RocksDB needs to provide both of backward and "forward" compatibility. (§4, Data format compatibility)
8. Automatic configuration adaptivity is helpful in simplifying configuration management. (§4, Managing configurations)
9. Data replication and backups need to be properly supported. (§4, Replication and backup support)
10. It is beneficial to detect data corruptions earlier, rather than eventually. (§5)
11. CPU/memory corruption does happen, though very rarely, and sometimes cannot be handled by data replication. (§5)
12. Integrity protection must cover the entire system in order to prevent corrupted data (e.g., caused by bitflips in CPU/memory) from being exposed to clients or other replicas; detecting corruption only when the data is at rest or being sent over the wire is insufficient. (§5)
13. Users often demand RocksDB to automatically recover from transient I/O errors, e.g. out-of-space or caused by network problems. (§5)
14. Error handling needs to be treated in a differentiated manner, depending on their causes and consequences. (§5)
15. The key/value interface is versatile, but there are some performance limitation; adding a timestamp to key/value can offer a good balance between performance and simplicity. (§6)

C Recap of design choices revisited

Some notable design choices revisited include:

1. Customizability is always good to users. (§4, Managing configurations)
2. RocksDB can be blind to CPU bit flips. (§5)
3. It's OK to panic when seeing any I/O error. (§5)

References

[1] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. ForestDB: A fast key-value storage system for variable-length string keys. *IEEE Trans. on Computers*, 65(3):902–915, 2015.

[2] Manos Athanassoulis, Michael S Kester, Lukas M Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. Designing access methods: The RUM conjecture. In *Proc. Intl. Conf on Extending Database Technology (EDBT)*, volume 2016, pages 461–466, 2016.

[3] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating synergies between memory, disk and log in log-structured key-value stores. In *Proc. USENIX Annual Technical Conference (USENIX-ATC'17)*, pages 363–375, 2017.

[4] Matias Bjørling. Zone Append: A new way of writing to zoned storage. In *Proc. Usenix Linux Storage and Filesystems Conference (VAULT'20)*, 2020.

[5] Facebook Engineering Blog. LogDevice: A distributed data store for logs. `https://engineering.fb.com/core-data/logdevice-a-distributed-data-store-for-logs/`. [Online; retrieved September 2020].

[6] Instagram Engineering Blog. Open-sourcing a 10x reduction in Apache Cassandra tail latency. `https://instagram-engineering.com/open-sourcing-a-10x-reduction-in-apache-cassandra-tail-latency-d64f86b43589`. [Online; retrieved September 2020].

[7] Netflix Technology Blog. Application data caching using SSDs: The Moneta project: Next generation EVCache for better cost optimization. `https://netflixtechblog.com/application-data-caching-\protect\discretionary{\char\hyphenchar\font}{}{}using-ssds-5bf25df851ef`. [Online; retrieved September 2020].

[8] Uber Engineering Blog. Cherami: Uber Engineering's durable and scalable task queue in Go. `https://eng.uber.com/cherami-message-queue-system/`. [Online; retrieved September 2020].

[9] Dhruba Borthakur. HDFS architecture guide. *Hadoop Apache Project*, 53(1-13):2, 2008.

[10] Mark Callaghan. MongoRocks and WiredTiger versus LinkBench on a small server. `http://smalldatum.blogspot.com/2016/10/mongorocks-and-wiredtiger-versus.html`. [Online; retrieved Jan 2021].

[11] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conf. on File and Storage Technologies (FAST'20)*, pages 209–223, February 2020.

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[13] Apache Cassandra. `https://cassandra.apache.org/`. [Online; retrieved September 2020].

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[15] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime data processing at Facebook. In *Proc. Intl. Conf. on Management of Data*, pages 1087–1098, 2016.

[16] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Trans. on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proc. Intl. Conf. on Management of Data (SIGMOD'17)*, pages 79–94, 2017.

[18] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'13)*, pages 1243–1254, 2013.

[19] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Stumm. Optimizing space amplification in RocksDB. In *Proc. Conf. on Innovative Data Systems Research (CIDR'17)*, 2017.

[20] Jose Faleiro. The dangers of logical replication and a practical solution. In *Proc. 18th Intl. Workshop on High Performance Transaction Systems (HPTS'19)*, 2019.

[21] Tasha Frankie, Gordon Hughes, and Ken Kreutz-Delgado. A mathematical model of the trim command in NAND-flash SSDs. In *Proc. 50th Annual Southeast Regional Conference (ACM-SE'12)*, pages 59–64, 2012.

[22] S. Ghemawat and J. Dean. LevelDB. https://github.com/google/leveldb, 2011.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proc. 19th ACM Symp. on Operating systems principles (SOSP'13)*, pages 29–43, 2003.

[24] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proc. European Conf. on Computer Systems (EUROSYS'15)*, pages 1–14, 2015.

[25] Caixin Gong, Shuibing He, Yili Gong, and Yingchun Lei. On integration of appends and merges in log-structured merge trees. In *Proc. 48th Intl. Conf. on Parallel Processing (ICPP'19)*, pages 1–10, 2019.

[26] Apache HBase. https://hbase.apache.org/. [Online; retrieved September 2020].

[27] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-based HTAP database. *Proc. VLDB Endow.*, 13(12):3072–3084, August 2020.

[28] Intel. Trim overview. https://www.intel.com/content/www/us/en/support/articles/000016148/memory-and-storage.html. [Online; retrieved Jan 2021].

[29] Iron.io. Confluent https://www.iron.io. [Online; retrieved September 2020].

[30] Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proc. SIGMOD Intl. Conf. on Management of Data (SIGMOD'15)*, pages 691–706, 2015.

[31] Jay Kreps. Introducing Kafka Streams: Stream processing made simple. Confluent https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/. [Online; retrieved September 2020].

[32] B Kuszmaul. How TokuDB fractal tree indexes work. Technical report, Technical report, TokuTek, 2010.

[33] Chuck Lever. End-to-end data integrity requirements for NFS. Oracle Corp. https://datatracker.ietf.org/meeting/83/materials/slides-83-nfsv4-2. [Online; retrieved September 2020].

[34] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symp. on Operating Systems Principles (SOSP'11)*, pages 1–13, 2011.

[35] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in SSD-conscious storage. *ACM Trans. on Storage (TOS)*, 13(1):1–28, 2017.

[36] Yoshinori Matsunobu. Migrating a database from InnoDB to MyRock. Facebook Engineering Blog https://engineering.fb.com/core-data/migrating-a-database-from-innodb-to-myrocks/, 2017. [Online; retrieved September 2020].

[37] Yoshinori Matsunobu, Siying Dong, and Herman Lee. MyRocks: LSM-tree database storage engine serving Facebook's Social Graph. *Proc. VLDB Endowment*, 13(12):3217–3230, August 2020.

[38] Microsoft. Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server/. [Online; retrieved September 2020].

[39] MongoDB. WiredTiger Storage Engine. https://docs.mongodb.com/manual/core/wiredtiger/. [Online; retrieved September 2020].

[40] MongoRocks. RocksDB storage engine module for MongoDB. https://github.com/mongodb-partners/mongo-rocks. [Online; retrieved September 2020].

[41] MySQL. Introduction to InnodeDB. https://dev.mysql.com/doc/refman/5.6/en/innodb-introduction.html. [Online; retrieved September 2020].

[42] MySQL. MySQL. https://www.mysql.com/. [Online; retrieved September 2020].

[43] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: Stateful scalable stream processing at LinkedIn. *Proc. of the VLDB Endowment*, 10(12):1634–1645, 2017.

[44] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[45] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[46] Keren Ouaknine, Oran Agra, and Zvika Guz. Optimization of RocksDB for Redis on flash. In *Proc. Intl. Conf. on Compute and Data Analysis*, pages 155–161, 2017.

[47] Mike Owens. *The definitive guide to SQLite*. Apress, 2006.

[48] Martin K Petersen. Linux data integrity extensions. In *Linux Symposium*, volume 4, page 5, 2008.

[49] Martin K. Petersen and Sergio Leunissen. Eliminating silent data corruption with Oracle Linux. Oracle Corp. `https://oss.oracle.com/~mkp/docs/data-integrity-webcast.pdf`. [Online; retrieved September 2020].

[50] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. Open-channel SSD (What is it good for). In *Proc. Conf. on Innovative Data Systems Research (CIDR'20)*, 2020.

[51] Qihoo. Confluent `https://github.com/Qihoo360/pika`. [Online; retrieved September 2020].

[52] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proc. 26th Symp. on Operating Systems Principles (SOSP'17)*, pages 497–514, 2017.

[53] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proc. of the VLDB Endowment (VLDB'17)*, 10(13):2037–2048, 2017.

[54] RocksDB.org. A persistent key-value store for fast storage environments. `https://rocksdb.org`. [Online; retrieved September 2020].

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

[56] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at Facebook and OANDA. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30. IEEE, 2016.

[57] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log-structured merge tree. In *Proc. Intl. Conf. on Management of Data (SIGMOD'12)*, pages 217–228, 2012.

[58] Arun Sharma. How we use RocksDB at Rockset. Rockset Blog `https://rockset.com/blog/how-we-use-rocksdb-at-rockset/`. [Online; retrieved September 2020].

[59] Arun Sharma. LogDevice: A distributed data store for logs. Facebook Engineering Blog `https://engineering.fb.com/data-infrastructure/dragon-a-distributed-graph-query-engine/`. [Online; retrieved September 2020].

[60] Pradeep J Shetty, Richard P Spillane, Ravikant R Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with VT-trees. In *Proc. 11th USENIX Conf. on File and Storage Technologies (FAST'13)*, pages 17–30, 2013.

[61] Gopalan Sivathanu, Charles P Wright, and Erez Zadok. Enhancing file system integrity through checksums. Technical report, Citeseer, 2004.

[62] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.

[63] Google Open Source. Protobuf. `https://opensource.google/projects/protobuf`. [Online; retrieved September 2020].

[64] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed SQL database. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'20)*, page 1493–1509, 2020.

[65] Amy Tai, Andrew Kryczka, Shobhit O. Kanaujia, Kyle Jamieson, Michael J. Freedman, and Asaf Cidon. Who's afraid of uncorrectable bit errors? Online recovery of flash errors with distributed redundancy. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 977–992, Renton, WA, July 2019.

[66] Tobias Vinçon, Sergej Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. NoFTL-KV: Tackling write-amplification on KV-stores with native storage management. In *Proc. 21st Intl. Conf. on Extending Database Technology (EDBT'18)*, pages 457–460, 2018.

[67] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proc. 9th European Conf. on Computer Systems (EUROSYS'14*, pages 1–14, 2014.

[68] Fei Yang, K Dou, S Chen, JU Kang, and S Cho. Multi-streaming RocksDB. In *Proc. Non-Volatile Memories Workshop*, 2015.

[69] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. FlashKV: Accelerating KV performance with open-channel SSDs. *ACM Trans on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.

[70] Yupu Zhang, Daniel S Myers, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Zettabyte reliability with flexible end-to-end data integrity. In *Proc. 29th IEEE Symp. on Mass Storage Systems and Technologies (MSST'13)*, pages 1–14, 2013.

[71] 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 *Proc. 8th USENIX Conf. on File and Storage Technologies (FAST'10)*, pages 29–42, 2010.

REMIX: Efficient Range Query for LSM-trees

Wenshao Zhong⋆ Chen Chen⋆ Xingbo Wu⋆ Song Jiang†
⋆University of Illinois at Chicago †University of Texas at Arlington

Abstract

LSM-tree based key-value (KV) stores organize data in a multi-level structure for high-speed writes. Range queries on traditional LSM-trees must seek and sort-merge data from multiple table files on the fly, which is expensive and often leads to mediocre read performance. To improve range query efficiency on LSM-trees, we introduce a space-efficient KV index data structure, named REMIX, that records a globally sorted view of KV data spanning multiple table files. A range query on multiple REMIX-indexed data files can quickly locate the target key using a binary search, and retrieve subsequent keys in sorted order without key comparisons. We build RemixDB, an LSM-tree based KV-store that adopts a write-efficient compaction strategy and employs REMIXes for fast point and range queries. Experimental results show that REMIXes can substantially improve range query performance in a write-optimized LSM-tree based KV-store.

1 Introduction

Key-value stores (KV-stores) are the backbone of many cloud and datacenter services, including social media [1, 2, 8], real-time analytics [7, 10, 25], e-commerce [18], and cryptocurrency [41]. The log-structured merge-tree (LSM-tree) [38] is the core data structure of many KV-stores [9, 18, 20, 26, 34, 42]. In contrast to traditional storage structures (e.g., B+-tree) that require in-place updates on disk, LSM-trees follow an out-of-place update scheme which enables high-speed sequential write I/O. They buffer updates in memory and periodically flush them to persistent storage to generate immutable table files. However, this comes with penalties on search efficiency as keys in a range may reside in different tables, potentially slowing down queries because of high computation and I/O costs. The LSM-tree based designs represent a trade-off between update cost and search cost [17], maintaining a lower update cost but a much higher search cost compared with a B+-tree.

Much effort has been made to improve query performance. To speed up point queries, every table is usually associated with memory-resident Bloom filters [4] so that a query can skip the tables that do not contain the target key. However, Bloom filters cannot handle range queries. Range filters such as SuRF [49] and Rosetta [37] were proposed to accelerate range queries by filtering out tables not containing any keys in the requested range. However, when the keys in the requested range reside in most of the candidate tables, the filtering approach can hardly improve query performance, especially for large range queries. Furthermore, the computation cost of accessing filters can lead to mediocre performance when queries can be answered by cache, which is often the case in real-world workloads [2, 8, 13].

To bound the number of tables that a search request has to access, LSM-trees keep a background compaction thread to constantly sort-merge tables. The table selection is determined by a compaction strategy. The *leveled* compaction strategy has been adopted by a number of KV-stores, including LevelDB [26] and RocksDB [20]. Leveled compaction sort-merges smaller sorted runs into larger ones to keep the number of overlapping tables under a threshold. In practice, leveled compaction provides the best read efficiency but has a high write amplification (WA) due to its aggressive sort-merging policy. Alternatively, the *tiered* compaction strategy waits for multiple sorted runs of a similar size and merges them into a larger run. Tiered compaction provides lower WA and higher update throughput. It has been adopted by many KV-stores, such as Cassandra [34] and ScyllaDB [42]. Since tiered compaction cannot effectively limit the number of overlapping tables, it leads to much higher search cost compared with leveled compaction. Other compaction strategies can better balance the read and write efficiency [16, 17], but none of them can achieve the best read and write efficiency at the same time.

The problem lies in the fact that, to limit the number of sorted runs, a store has to sort-merge and rewrite existing data. Today's storage technologies have shown much improved random access efficiency. For example, random reads on commodity Flash SSDs can exceed 50% of sequential read throughput. New technologies such as 3D-XPoint (e.g., Intel's Optane SSD) offer near-equal performance for random and sequential I/O [45]. As a result, KV-pairs do not have to be *physically* sorted for fast access. Instead, a KV-store could keep its data *logically* sorted for efficient point and range queries while avoiding excessive rewrites.

To this end, we design REMIX, short for **R**ange-query-**E**fficient **M**ulti-table **I**nde**X**. Unlike existing solutions to improve range queries that struggle between physically rewriting data and performing expensive sort-merging on the fly, a REMIX employs a space-efficient data structure to record a globally sorted view of KV data spanning multiple table files. With REMIXes, an LSM-tree based KV-store can take advantage of a write-efficient compaction strategy without sacrificing search performance.

We build RemixDB, a REMIX-indexed LSM-tree based KV-store. Integrated with the write-efficient tiered compaction strategy and a partitioned LSM-tree layout, RemixDB achieves low WA and fast searches at the same time. Experimental results show that REMIXes can effectively improve range query performance when searching on multiple overlapping tables. Performance evaluation demonstrates that RemixDB outperforms the state-of-the-art LSM-tree based KV-stores on both read and write operations simultaneously.

2 Background

The LSM-tree is designed for high write efficiency on persistent storage devices. It achieves high-speed writes by buffering all updates in an in-memory structure, called a MemTable. When the MemTable fills up, the buffered keys will be sorted and flushed to persistent storage as a sorted run by a process called *minor compaction*. Minor compaction is write-efficient because updates are written sequentially in batches without merging with existing data in the store. Since the sorted runs may have overlapping key ranges, a point query has to check all the possible runs, leading to a high search cost. To limit the number of overlapping runs, an LSM-tree uses a *major compaction* process to sort-merge several overlapping runs into fewer ones.

A compaction strategy determines how tables are selected for major compaction. The two most commonly used strategies are leveled compaction and tiered compaction. A store using leveled compaction has a multi-level structure where each level maintains a sorted run consisting of one or more tables. The capacity of a level (L_n) is a multiple (usually 10 [20]) of the previous one (L_{n-1}), which allows a huge KV-store to be organized within a few levels (usually 5 to 7). Leveled compaction makes reads relatively efficient, but it leads to inferior write efficiency. Leveled compaction selects overlapping tables from adjacent levels (L_n and L_{n+1}) for sort-merging and generates new tables in the larger level (L_{n+1}). Because of the exponentially increasing capacity, a table's key range often overlaps several tables in the next level. As a result, the majority of the writes are for rewriting existing data in L_{n+1}, leading to high WA ratios[1] of up to 40 in practice [40]. Figure 1 shows an example of leveled compaction where each table contains two or three keys. If the first table in L_1 (containing keys $(4,\ 21,\ 38)$) is selected for sort-merging with the first two tables in L_2 ($(6,26)$ and $(31,40,46)$), five keys in L_2 will be rewritten.

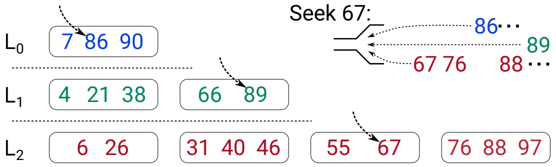


Figure 1: An LSM-tree using leveled compaction

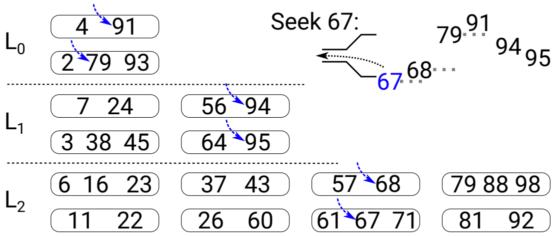


Figure 2: An LSM-tree using tiered compaction

With tiered compaction, multiple overlapping sorted runs can be buffered in a level, as shown in Figure 2. The number of runs in a level is bounded by a threshold denoted by T, where $T > 1$. When the number of sorted runs in a level (L_n) reaches the threshold, all sorted runs in L_n will be sort-merged into a new sorted run in the next level (L_{n+1}), without rewriting any existing data in L_{n+1}. Accordingly, an LSM-tree's WA ratio is $O(L)$ using tiered compaction [15], where L is the number of levels. With a relatively large T, tiered compaction provides much lower WA than leveled compaction does with a similar L. However, since there can be multiple overlapping sorted runs in each level, a point query will need to check up to $T \times L$ tables, leading to a much slower search.

Range query in LevelDB/RocksDB is realized by using an *iterator* structure to navigate across multiple tables as if all the keys are in one sorted run. A range query first initializes an iterator using a *seek* operation with a *seek key*, the lower boundary of the target key range. The seek operation positions the iterator so that it points to the smallest key in the store that is equal to or greater than the seek key (in lexical order for string keys), which is denoted as the *target key* of the range query. The *next* operation advances the iterator such that it points to the next key in the sorted order. A sequence of next operations can be used to retrieve the subsequent keys in the target range until a certain condition is met (e.g., number of keys or end of a range). Since the sorted runs are generated chronologically, a target key can reside in any of the runs. Accordingly, an iterator must keep track of all the sorted runs.

Figure 1 shows an example of seek on an LSM-tree using leveled compaction. To seek key 67, a binary search is used on each run to identify the smallest key satisfying $key \geq seek\_key$. Each identified key is marked by a cursor. Then these keys are sort-merged using a min-heap structure [23], and thus the key 67 in L_2 is selected. Subsequently, each next

[1] WA ratio refers to write amplification ratio, or ratio of the amount of actual data written on the disk to the amount of user-requested data written.

operation will compare the keys under the cursors, return the smallest one, and advance the corresponding cursor. This process presents a globally sorted view of the keys, as shown in the upper right corner of Figure 1. In this example, all three levels must be accessed for the sort-merging. Figure 2 shows a similar example with tiered compaction. Having six overlapping sorted runs, a seek operation is more expensive than the previous example. In practice, the threshold T in tiered compaction is often set to a small value, such as $T = 4$ in ScyllaDB [42], to avoid having too many overlapping sorted runs in a store.

3 REMIX

A range query operation on multiple sorted runs constructs a *sorted view* of the underlying tables on the fly so that the keys can be retrieved in sorted order. In fact, a sorted view inherits the immutability of the table files and remains valid until any of the tables are deleted or replaced. However, existing LSM-tree based KV-stores have not been able to take advantage of this inherited immutability. Instead, sorted views are repeatedly reconstructed at search time and immediately discarded afterward, which leads to poor search performance due to excessive computation and I/O. The motivation of REMIX is to exploit the immutability of table files by retaining the sorted view of the underlying tables and reusing them for future searches.

For I/O efficiency, the LSM-tree based KV-stores employ memory-efficient metadata formats, including sparse indexes and Bloom filters [4]. If we record every key and its location to retain the sorted views in a store, the store's metadata could be significantly inflated, leading to compromised performance for both reads and writes. To avoid this issue, the REMIX data structure must be space-efficient.

3.1 The REMIX Data Structure

The top of Figure 3 shows an example of a sorted view containing three sorted runs, R_0, R_1, and R_2. The sorted view of the three runs is illustrated by the arrows, forming a sequence of 15 keys. To construct a REMIX, we first divide the keys of a sorted view into segments, each containing a fixed number of keys. Each segment is attached with an *anchor key*, a set of *cursor offsets*, and a set of *run selectors*. An anchor key represents the smallest key in the segment. All the anchor keys collectively form a sparse index on the sorted view. Each cursor offset corresponds to a run and records the position of the smallest key in the run that is equal to or greater than the segment's anchor key. Each key in a segment has a corresponding run selector, which indicates the run where the key resides. The run selectors encode the sequential access path of the keys on the sorted view, starting from the anchor key of the segment.

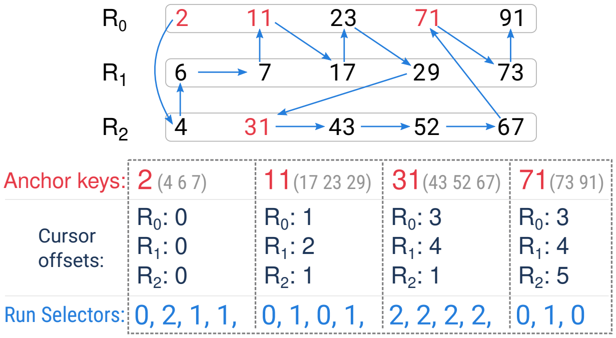


Figure 3: A sorted view of three sorted runs with REMIX

An iterator for a REMIX does not use a min-heap. Instead, an iterator contains a set of cursors and a *current* pointer. Each cursor corresponds to a run and points to the location of a key in the run. The current pointer points to a run selector, which selects a run, and the cursor of the run determines the key currently being reached.

It takes three steps to seek a key using an iterator on a REMIX. First, a binary search is performed on the anchor keys to find the *target segment* whose range covers the seek key, satisfying $anchor\_key \leq seek\_key$. Second, the iterator is initialized to point to the anchor key. Specifically, the cursors are positioned using the cursor offsets of the segment, and the current pointer is set to point to the first run selector of the segment. Finally, the target key can be found by scanning linearly on the sorted view. To advance the iterator, the cursor of the current key is advanced to skip the key. Meanwhile, the current pointer is also advanced to point to the next run selector. After a seek operation, the subsequent keys on the sorted view (within and beyond the target segment) can be retrieved by advancing the iterator in the same manner.

Here is an example of a seek operation. As shown in Figure 3, the four boxes on the bottom represent the REMIX metadata that encodes the sorted view. Note that the keys in parentheses are not part of the metadata. To seek key 17, the second segment, which covers keys $(11,17,23,29)$, is selected with a binary search. Then the cursors are placed on keys 11, 17, and 31 in R_0, R_1, and R_2, respectively, according to the segment's cursor offsets $((1,2,1))$. Meanwhile, the current pointer is set to point to the first run selector of the segment (0, the fifth selector in the figure), indicating that the current key (11) is under the cursor of R_0. Since $11 < 17$, the iterator needs to be advanced to find the smallest key k satisfying $k \geq 17$. To advance the iterator, the cursor on R_0 is first advanced so that it skips key 11 and is now on key 23. The cursor offsets of the iterator now become 2, 2, and 1. Then, the current pointer is advanced to the second run selector of the segment (1, the sixth selector in the figure). The advanced iterator selects R_1, and the current key 17 under the cursor of R_1 is the target key. This concludes the seek operation. The subsequent keys (23, 29, 31, ...) on the sorted view can be retrieved by repeatedly advancing the iterator.

3.2 Efficient Search in a Segment

A seek operation initializes the iterator with a binary search on the anchor keys to find the target segment and scans forward on the sorted view to look for the target key. Increasing the segment size can reduce the number of anchor keys and speed up the binary search. However, it can slow down seek operations because scanning in a large target segment needs to access more keys on average. To address the potential performance issue, we also use binary search within a target segment to minimize the search cost.

Binary Search To perform binary search in a segment, we must be able to randomly access every key in the segment. A key in a segment belongs to a run, as indicated by the corresponding run selector. To access a key, we need to place the cursor of the run in the correct position. This can be done by counting the number of occurrences of the same run selector in the segment prior to the key and advancing the corresponding cursor the same number of times. The number of occurrences can be quickly calculated on the fly using SIMD instructions on modern CPUs. The search range can be quickly reduced with a few random accesses in the segment until the target key is identified. To conclude the seek operation, we initialize all the cursors using the occurrences of each run selector prior to the target key.

Figure 4 shows an example of a segment having 16 run selectors. The number shown below each run selector represents the number of occurrences of the same run selector prior to its position. For example, 41 is the third key in R_3 in this segment, so the corresponding number of occurrences is 2 (under the third "3"). To access key 41, we initialize the cursor of R_3 and advance it twice to skip 5 and 23.

To seek key 41 in the segment in Figure 4, keys 43, 17, 31, and 41 will be accessed successively during the binary search, as shown by the arrows and the circled numbers. Key 43 is the eighth key in the segment and the fourth key of R_3 in the segment. To access key 43, we initialize the cursor of R_3 and advance it three times to skip keys 5, 23, and 41. Then, key 17 can be accessed by reading the first key on R_2 in this segment. Similarly, 31 and 41 are the second and third keys on R_1 and R_3, respectively. In the end, all the cursors of the iterator are initialized to point to the correct keys. In this example, the cursors will finally be at keys 61, 53, 89, and 41, where 41 is the current key.

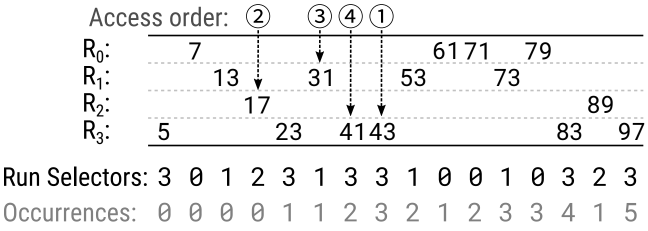

Access order: ② ③ ④ ①
R0: 7 61 71 79
R1: 13 31 53 73
R2: 17 89
R3: 5 23 41 43 83 97
Run Selectors: 3 0 1 2 3 1 3 3 1 0 0 1 0 3 2 3
Occurrences: 0 0 0 0 1 1 2 3 2 1 2 3 3 4 1 5

Figure 4: An example of binary search in a segment. The circled numbers indicate the access order of the keys.

I/O Optimization Performing binary search in a segment can minimize the number of key comparisons. However, the keys on the search path may reside in different runs and must be retrieved with separate I/O requests if the respective data blocks are not cached. For example, the search in Figure 4 only needs four key comparisons but has to access three runs. In fact, it is likely that keys 41, 43, and a few other keys of R_3 belong to the same data block. Accordingly, after a key comparison, the search can leverage the remaining keys in the same data block to further reduce the search range before it has to access a different run. In this way, each of the six keys in R_3 can be found without accessing any other runs. When searching for key 79, for example, accessing R_3 can narrow down the search to the range between key 43 and key 83, where key 79 can be found in R_0 after a key comparison with key 71.

3.3 Search Efficiency

REMIXes improve range queries in three aspects.

REMIXes find the target key using one binary search. A REMIX provides a sorted view of multiple sorted runs. Only one binary search on a REMIX is required to position the cursors on the target keys in multiple runs. Whereas in a traditional LSM-tree based KV-store, a seek operation requires a number of binary searches on each individual run. For example, suppose a store with four equally-sized runs has N keys in each run. A seek operation without a REMIX requires $4 \times \log_2 N$ key comparisons, while it only takes $\log_2 4N$, or $2 + \log_2 N$ key comparisons with a REMIX.

REMIXes move the iterator without key comparisons. An iterator on a REMIX directly switches to the next (or the previous) KV-pair by using the prerecorded run selectors to update the cursors and the current pointer. This process does not require any key comparisons. Reading a KV-pair can also be avoided if the iterator skips the key. In contrast, an iterator in a traditional LSM-tree based KV-store maintains a min-heap to sort-merge the keys from multiple overlapping sorted runs. In this scenario, a next operation requires reading keys from multiple runs for comparisons.

REMIXes skip runs that are not on the search path. A seek operation with a REMIX requires a binary search in the target segment. Only those sorted runs containing the keys on the search path will be accessed at search time. In the best scenario, if a range of target keys reside in one run, such as the segment $(31, 43, 52, 67)$ in Figure 3, only one run (R_2 in the example) will be accessed. However, a merging iterator must access every run in a seek operation.

Furthermore, the substantially reduced seek cost allows for efficient point queries (e.g., GET) on multiple sorted runs indexed by a REMIX without using Bloom filters. We extensively evaluate the point query efficiency in §5.1.

3.4 REMIX Storage Cost

REMIX metadata consists of three components: anchor keys, cursor offsets, and run selectors. We define D to be the maximum number of keys in a segment. A REMIX stores one anchor key for every D keys, requiring $1/D$ of the total key size in a level on average. Assuming the size of a cursor offset is S bytes, a REMIX requires $S \times H$ bytes to store the cursor offsets for every D keys, where H denotes the number of runs indexed by a REMIX. A run selector requires $\lceil \log_2(H) \rceil$ bits. Adding all the three parts together, a REMIX is expected to store $((\bar{L}+SH)/D+\lceil \log_2(H) \rceil/8)$ bytes/key, where $\bar{L}$ is the average anchor key size.

We estimate the storage cost of a REMIX using the average KV sizes publicly reported in Facebook's production KV workloads [2, 8]. In practice, S is implementation-defined, and H depends on the number of tables being indexed. In the estimation, we use cursor offsets of 4 bytes ($S=4$) so that a cursor offset can address 4 GB space for each sorted run. We set the number of sorted runs to 8 ($H=8$). With these practical configurations, a REMIX stores $((\bar{L}+32)/D+3/8)$ bytes/key.

Table 1 shows the REMIX storage costs for each workload with different D (D =16, 32, and 64). For comparison, it also shows the storage cost of the block index (BI) and Bloom filter (BF) of the SSTable format in LevelDB and RocksDB. Note that table files indexed by REMIXes do not use block indexes or Bloom filters. An SSTable stores a key and a block handle for each 4 KB data block. The block index storage cost is estimated by dividing the sum of the average KV size and an approximate block handle size (4 B) by the estimated number of KV-pairs in a 4 KB block. Bloom filters are estimated as 10 bits/key. The REMIX storage costs vary from 1.0 to 5.4 bytes/key for different D and $\bar{L}$ values. For every key size, increasing D can substantially reduce the REMIX storage cost. The last column $\left(\frac{\text{REMIX}}{\text{data}}\right)$ shows the size ratio of a REMIX to its indexed KV data. In the worst case (the USR store), the REMIX's size is still less than 10% of the KV data's size.

Table 1: REMIX storage cost with real-world KV sizes. BI stands for Block Index. BF stands for Bloom Filter. The last column shows the size ratio of REMIX to the KV data.

Work-	Avg.	Avg.	Bytes/Key					REMIX
load	Key	Value	SSTable		REMIX (H=8)			data
[2, 8]	Size	Size	BI	BI+BF	D=16	32	64	(D=32)
UDB	27.1	126.7	1.2	2.4	4.1	2.2	1.3	1.44%
Zippy	47.9	42.9	1.2	2.4	5.4	2.9	1.6	3.16%
UP2X	10.45	46.8	0.2	1.5	3.0	1.7	1.0	2.97%
USR	19	2	0.1	1.4	3.6	2.0	1.2	9.38%
APP	38	245	2.9	4.2	4.8	2.6	1.5	0.91%
ETC	41	358	4.4	5.6	4.9	2.7	1.5	0.67%
VAR	35	115	1.4	2.7	4.6	2.5	1.4	1.65%
SYS	28	396	3.3	4.6	4.1	2.3	1.3	0.53%

4 RemixDB

To evaluate the REMIX performance, we implement an LSM-tree based KV-store named RemixDB. RemixDB employs the tiered compaction strategy to achieve the best write efficiency [16]. Real-world workloads often exhibit high spatial locality [2, 8, 47]. Recent studies have shown that a partitioned store layout can effectively reduce the compaction cost under real-world workloads [24, 31]. RemixDB adopts this approach by dividing the key space into partitions of non-overlapping key ranges. The table files in each partition are indexed by a REMIX, providing a sorted view of the partition. In this way, RemixDB is essentially a single-level LSM-tree using tiered compaction. RemixDB not only inherits the write efficiency of tiered compaction but also achieves efficient reads with the help of REMIXes. The point query operation (GET) of RemixDB performs a seek operation and returns the key under the iterator if it matches the target key. RemixDB does not use Bloom filters.

Figure 5 shows the system components of RemixDB. Similarly to LevelDB and RocksDB, RemixDB buffers updates in a MemTable. Meanwhile, the updates are also appended to a write-ahead log (WAL) for persistence. When the size of the buffered updates reaches a threshold, the MemTable is converted into an immutable MemTable for compaction, and a new MemTable is created to receive updates. A compaction in a partition creates a new version of the partition that includes a mix of new and old table files and a new REMIX file. The old version is garbage-collected after the compaction.

In a multi-level LSM-tree design, the size of a MemTable is often only tens of MBs, close to the default SSTable size. In a partitioned store layout, larger MemTables can accumulate more updates before triggering a compaction [3, 24], which helps to reduce WA. The MemTables and WAL have near-constant space cost, which is modest given the large memory and storage capacity in today's datacenters. In RemixDB, the maximum MemTable size is set to 4 GB. In the following, we introduce the file structures (§4.1), the compaction process (§4.2), and the cost and trade-offs of using REMIXes (§4.3).

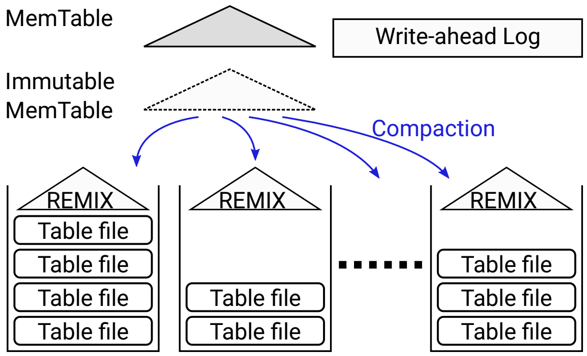


Figure 5: Overview of RemixDB

4.1 The Structures of RemixDB Files

Table Files Figure 6 shows the table file format in RemixDB. A data block is 4 KB by default. A large KV-pair that does not fit in a 4 KB block exclusively occupies a *jumbo* block that is a multiple of 4 KB. Each data block contains a small array of its KV-pairs' block offsets at the beginning of the block for randomly accessing individual KV-pairs.

The metadata block is an array of 8-bit values, each recording the number of keys in a 4 KB block. Accordingly, a block can contain up to 255 KV-pairs. In a jumbo block, except for the first 4 KB, the remaining ones have their corresponding numbers set to 0 so that a non-zero number always corresponds to a block's head. With the offset arrays and the metadata block, a search can quickly reach any adjacent block and skip an arbitrary number of keys without accessing the data blocks. Since the KV-pairs are indexed by a REMIX, table files do not contain indexes or filters.

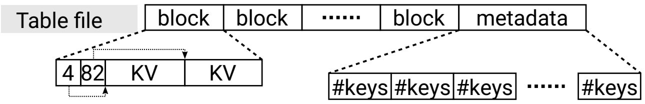


Figure 6: Structure of a table file in RemixDB

REMIX Files Figure 7 shows the REMIX file format in RemixDB. The anchor keys in a REMIX are organized in an immutable B+-tree-like index (similar to LevelDB/RocksDB's block index) that facilitates binary searches on the anchor keys. Each anchor key is associated with a segment ID that identifies the cursor offsets and run selectors of a segment. A cursor offset consists of a 16-bit block index and an 8-bit key index, shown as `blk-id` and `key-id` in Figure 7. The block index can address up to 65,536 4-KB blocks (256 MB). Each block can contain up to 256 KV-pairs with the 8-bit key index.

Multiple versions of a key could exist in different table files of a partition. A range query operation must skip the old versions and return the newest version of each key. To this end, in a REMIX, multiple versions of a key are ordered from the newest to the oldest on the sorted view, and the highest bit of each run selector is reserved to distinguish between old and new versions. A forward scan operation will always encounter the newest version of a key first, and then the old versions can be skipped by checking the reserved bit of each run selector without comparing any keys.

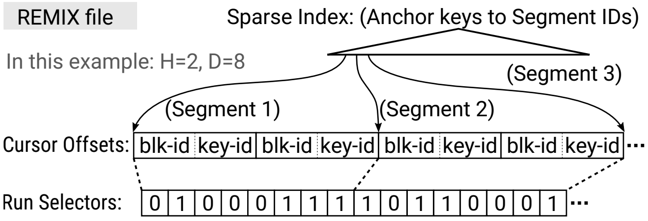


Figure 7: Structure of a REMIX file in RemixDB

If a key has multiple versions, these versions can span two segments. A search may have to check both the segments to retrieve the newest version of the key. To simplify searches in this scenario, we move all the versions of the key forward to the second segment by inserting special run selectors as placeholders in the first segment when constructing a REMIX. We also make sure that the maximum number of keys in a segment is equal to or greater than the number of runs indexed by a REMIX ($D \geq H$) so that every segment is large enough to hold all the versions of a key.

To accommodate the special values mentioned above, each run selector in RemixDB occupies a byte. The eighth and seventh bits (`0x80` and `0x40`) of a run selector indicate an old version and a deleted key (a tombstone), respectively. A special value 63 (`0x3f`) represents a placeholder. In this way, RemixDB can manage up to 63 sorted runs (0 to 62) in each partition, which is sufficient in practice.

4.2 Compaction

In each partition, the compaction process estimates the compaction cost based on the size of new data entering the partition and the layout of existing tables. Based on the estimation, one of the following procedures is executed:

- Abort: cancel the partition's compaction and keep the new data in the MemTables and the WAL.
- Minor Compaction: write the new data to one or multiple new tables without rewriting existing tables.
- Major Compaction: merge the new data with some or all of the existing tables.
- Split Compaction: merge the new data with all the existing data and split the partition into a few new partitions.

Abort After a compaction, a partition that sees any new table file will have its REMIX rebuilt. When a small table file is created in a partition after a minor compaction, rebuilding the REMIX can lead to high I/O cost. For example, the USR workload in Table 1 has the highest size ratio of REMIX to KV data (9.38%). Writing 100 MB of new data to a partition with 1 GB of old table files will create a REMIX that is about 100 MB. To minimize the I/O cost, RemixDB can abort a partition's compaction if the estimated I/O cost is above a threshold. In this scenario, the new KV data should stay in the MemTables and the WAL until the next compaction.

However, in an extreme case, such as having a workload with a uniform access pattern, the compaction process cannot effectively move data into the partitions when most of the partitions have their compactions aborted. To avoid this problem, we further limit the size of new data that can stay in the MemTables and WAL to be no more than 15% of the maximum MemTable size. The compaction process can abort the compactions that have the highest I/O cost until the size limit has been reached.

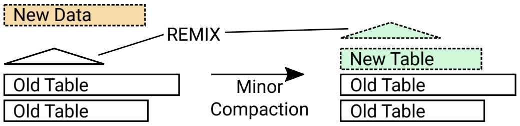


Figure 8: Minor compaction

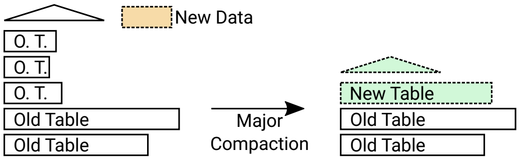


Figure 9: Major compaction

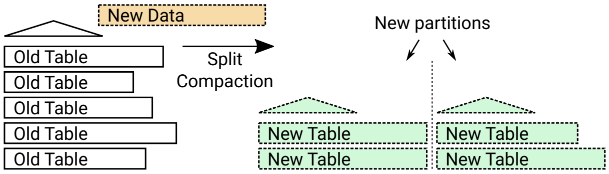


Figure 10: Split compaction

Minor Compaction A minor compaction writes new KV data from the immutable MemTable into a partition without rewriting existing table files and rebuilds the REMIX of the partition. Depending on the new data's size, a minor compaction creates one or a few new table files. Minor compaction is used when the expected number of table files after the compaction (number of existing table files plus the estimated number of new table files) is below a threshold T, which is 10 in our implementation. Figure 8 shows a minor compaction example that creates one new table file.

Major Compaction A major (or split) compaction is required when the expected number of table files in a partition exceeds the threshold T. A major compaction sort-merges existing table files into fewer ones. With a reduced number of table files, minor compactions can be performed in the future. The efficiency of a major compaction can be estimated by the ratio of the number of input table files to the number of output table files. Figure 9 shows a major compaction example. In this example, the new data is merged with three small table files, and only one new table file is created after the compaction (ratio=$3/1$). If the entire partition is sort-merged, the compaction needs to rewrite more data but still produces three tables (ratio=$5/3$) because of the table file's size limit. Accordingly, major compaction chooses the number of input files that can produce the highest ratio.

Split Compaction Major compaction may not effectively reduce the number of tables in a partition filled with large tables, which can be predicted by a low estimated input/output ratio, such as $10/9$. In this case, the partition should be split into multiple partitions so that the number of tables in each partition can be substantially reduced. Split compaction sort-merges new data with all the existing table files in the partition and produces new table files to form several new partitions. Figure 10 shows a split compaction example. To avoid creating many small partitions in a split compaction, the compaction process creates M ($M = 2$ by default) new table files in a partition before switching to the next partition. In this way, a split compaction creates $\lceil E/M \rceil$ new partitions, where E is the number of new table files.

4.3 Rebuilding REMIXes

A partitioned store layout can effectively minimize the compaction cost under real-world workloads with high spatial locality [24, 31]. Specifically, RemixDB can absorb most of the updates in a few partitions, and the compactions in the partitions that receive fewer updates can be avoided (See §4.2). However, if the workload lacks spatial locality, it is inevitable that many partitions have to perform compactions with small amounts of updates. Tiered compaction can minimize writes in these partitions, but rebuilding the REMIX in a partition still needs to read the existing tables. In our implementation, RemixDB leverages the existing REMIX in the partition and employs an efficient merging algorithm to minimize the I/O cost of the rebuilding process.

When rebuilding the REMIX in a partition, the existing tables are already indexed by the REMIX, and those tables can be viewed as one sorted run. Accordingly, the rebuilding process is equivalent to sort-merging two sorted runs, one from the existing data and the other from the new data. When the existing sorted run is significantly larger than the new one, the *generalized binary merging algorithm* proposed by Hwang et al. [30, 33] requires much fewer key comparisons than sort-merging with a min-heap. The algorithm estimates the location of each next merge point based on the size ratio between the two sorted runs and search in the neighboring range. In RemixDB, we approximate the algorithm by using the anchor keys to locate the target segment containing the merge point and finally applying a binary search in the segment. In this process, accessing anchor keys does not incur any I/O since they are stored in the REMIX. A binary search in the target segment reads at most $\log_2 D$ keys to find the merge point. All the run selectors and cursor offsets for the existing tables can be derived from the existing REMIX without any I/O. To create anchor keys for the new segments, we need to access at most one key per segment on the new sorted view.

The read I/O of rebuilding a REMIX is bounded by the size of all the tables in a partition. The rebuilding process incurs read I/O to the existing tables in exchange for minimized WA and improved future read performance. Whether rebuilding a REMIX is cost effective depends on how much write I/O one wants to save and how much future read performance one wants to improve. In practice, writes in SSDs are usually slower than reads and can cause permanent damage to the devices [5, 27, 28, 45]. As a result, reads are more economical than writes, especially for systems having spare I/O bandwidth. In systems that expect intensive writes with weak spatial locality, adopting a multi-level tiered compaction strategy [40, 46] or delaying rebuilding REMIXes in individual partitions can reduce the rebuilding cost at the expense of having more levels of sorted views. Adapting REMIXes with different store layouts is beyond the scope of this paper. We empirically evaluate the rebuilding cost in RemixDB under different workloads in §5.2.

5 Evaluation

In this section, we first evaluate the REMIX performance characteristics (§5.1), and then benchmark RemixDB with a set of micro-benchmarks and Yahoo's YCSB benchmark tool that emulates real-world workloads [13] (§5.2).

The evaluation system runs 64-bit Linux (v5.8.7) on two Intel Xeon Silver 4210 CPUs and 64 GB of DRAM. The experiments run on an Ext4 file system on a 960 GB Intel 905P Optane PCIe SSD.

5.1 Performance of REMIX-indexed Tables

We first evaluate the REMIX performance. We implement a micro-benchmark framework that compares the performance of REMIX-indexed tables with SSTables. The SSTables use Bloom filters to accelerate point queries and employ merging iterators to perform range queries.

Experimental Setup In each experiment, we first create a set of H table files ($1 \leq H \leq 16$), which resemble a partition in a RemixDB or a level in an LSM-tree using tiered compaction. Each table file contains 64 MB of KV-pairs, where the key and value sizes are 16 B and 100 B, respectively. When $H \geq 2$, the KV-pairs can be assigned to the tables using two different patterns:

- **Weak locality**: each key is assigned to a randomly selected table, which provides weak access locality since logically consecutive keys often reside in different tables.
- **Strong locality**: every 64 logically consecutive keys are assigned to a randomly selected table, which provides strong access locality since a range query can retrieve a number of consecutive keys from few tables.

Each SSTable contains Bloom filters of 10 bits/key. A 64 MB user-space block cache[2] is used for accessing the files.

We measure the single-threaded throughput of three range and point query operations, namely Seek, Seek+Next50, and Get, using different sets of tables created with the above configurations. A Seek+Next50 operation performs a seek and retrieves the next 50 KV-pairs. In these experiments, the seek keys are randomly selected following a uniform distribution. For REMIX, we set the segment size to 32 ($D = 32$), and measure the throughput with its in-segment binary search turned on and off, denoted by *full* and *partial* binary search, respectively (see §3.2). For point queries (Get), we measure the throughput of SSTables with Bloom filters turned on and off. We run each experiment until the throughput reading is stable. Figures 11 and 12 show the throughput results for tables with weak and strong access locality, respectively.

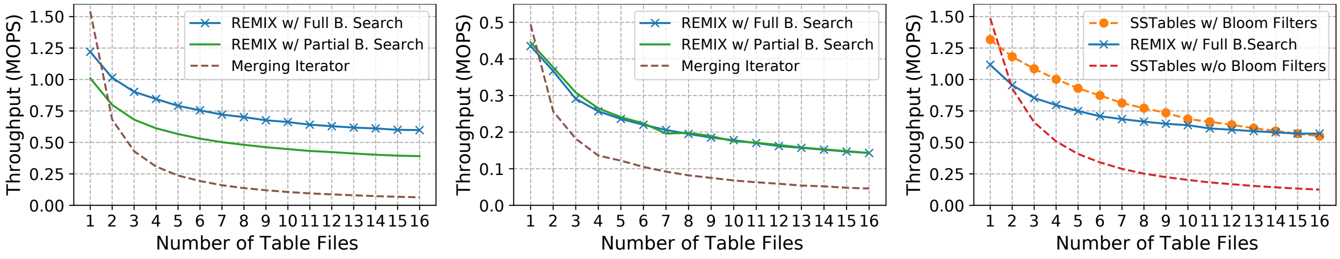
Seek on Tables of Weak Locality Figure 11a shows the throughput of seek operations using a REMIX and a merging iterator. We observe that the throughput with the merging iterator is roughly 20% higher than that of a REMIX with full binary search when there is only one table file. In this scenario, both the mechanisms perform the same number of key comparisons during the binary search. However, when searching in a segment, the REMIX needs to count the number of occurrences on the fly and move the iterator from the beginning of the segment to reach a key for comparison, which is more expensive than a regular iterator.

The throughput of a merging iterator quickly drops as the number of table files increases. Specifically, the throughput of two tables is 50% lower than that of one table; a seek on eight tables is more than 11× slower than a seek on one table. The seek time of a merging iterator is approximately proportional to the number of table files. This is because the merging iterator requires a full binary search on every table file. The REMIX's throughput also decreases with more tables files. The slowdown is mainly due to the growing dataset that requires more key comparisons and memory accesses during a search. However, the REMIX with full binary search achieves increasingly high speedups compared with the merging iterator. Specifically, The speedups are 5.1× and 9.3× with 8 and 16 table files, respectively.

The REMIX throughput decreases by 20% to 33% when the in-segment binary search is turned off (with partial binary search). In this scenario, a seek has to linearly scan the target segment to find the target key. With $D = 32$, the average number of key comparisons in a target segment is 5 ($\log_2 D$) with full binary search and 16 ($D/2$) with partial binary search. However, the search cost is still substantially lower than that of a merging iterator. The REMIX with partial binary search outperforms the merging iterator by 3.5× and 6.1×, with 8 and 16 table files, respectively.

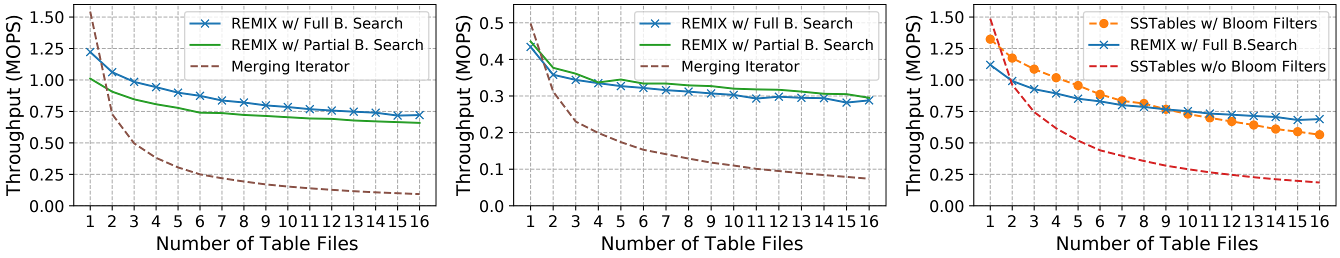
Seek+Next50 Figure 11b shows the throughput of range queries that seek and copy 50 KV-pairs to a user-provided buffer. The overall throughput results are much lower than that in the Seek experiments because the data copying is expensive. However, the REMIX still outperforms the merging iterator when there are two or more tables. The speedup is 1.4×, 2.3×, and 3.1× with 2, 8, and 16 table files, respectively. The suboptimal scan performance of the merging iterator is due to the expensive next operation that requires multiple key comparisons to find the next key on the sorted view. For each KV-pair copied to the buffer, multiple KV-pairs must be read and compared to find the global minimum. In contrast, a REMIX does not require any key comparisons in a next operation.

[2] LevelDB's LRUCache implementation in `util/cache.cc`.



(a) Seek **(b)** Seek+Next50 **(c)** Get

Figure 11: Point and range query performance on tables where keys are randomly assigned (weak locality)



(a) Seek **(b)** Seek+Next50 **(c)** Get

Figure 12: Point and range query performance on tables where every 64 keys are assigned to a table (strong locality)

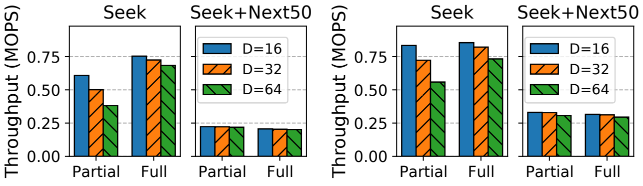
In contrast to the substantial performance gap between the two REMIX curves in Figure 11a, the two curves in Figure 11b are very close to each other. This phenomenon is the result of two effects: (1) the next operations dominate the execution time and (2) the linear scanning of a seek operation in a segment warms up the block cache, which makes the future next operations faster.

Point Query Figure 11c shows the results of the point query experiments. The REMIX's curve is slightly lower than its counterpart in Figure 11a because a get operation needs to copy the KV-pair after a seek using the REMIX. Searching on SSTables with Bloom filters outperforms searching on REMIX-indexed table files when there are fewer than 14 tables. The reasons for the differences are two-fold. First, a search can be effectively narrowed down to one table file at a small cost of checking the Bloom filters. Second, searching in an SSTable is faster than on a REMIX managing many more keys. In the worst case, the REMIX's throughput is 20% lower than that of Bloom filters (with 3 tables). Unsurprisingly, the searches with more than two SSTables are much slower without Bloom filters.

Performance with Tables of Strong Locality Figure 12 shows the range and point query performance on tables with strong access locality. The results in Figures 12a and 12b follow a similar trend of their counterparts in Figure 11. In general, the improved locality allows for faster binary searches since in this scenario the last few key comparisons can often use keys in the same data block. However, the throughput of the merging iterator remains low because of the intensive key comparisons that dominate the search time. The REMIX with partial binary search improves more than that with full binary search. This is because improved locality reduces the penalty on the scanning in a target segment, where fewer cache misses are incurred in each seek operation.

The REMIX point query performance also improves due to the strong locality that speeds up the underlying seek operations, as shown in Figure 12c. Meanwhile, the results of Bloom filters stay unchanged because the search cost is mainly determined by the false-positive rate and the search cost on individual tables. As a result, REMIXes are able to outperform Bloom filters when there are more than 9 tables.

Segment Size (*D*) We further evaluate REMIX range query performance using different segment sizes ($D \in \{16, 32, 64\}$) on eight table files. The other configuration parameters are the same as in the previous experiments. Figure 13 shows the performance results. The throughput of seek-only operations exhibits the largest variations with different Ds when the in-segment binary search is turned off. This is because the linear scanning in a segment adds a significant cost with a large D. On the other hand, the differences become much smaller with full binary search. In the meantime, a larger segment size still leads to higher overhead because of the slower random access speed within a segment. In the Seek+Next50 experiments, the data copying dominates the execution time and there are no significant differences when using different Ds.



(a) Tables of weak locality **(b)** Tables of strong locality

Figure 13: REMIX range query performance with 8 runs

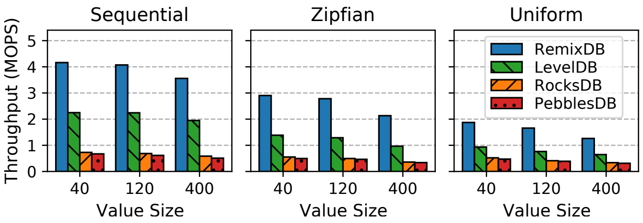


Figure 14: Range query with different value sizes

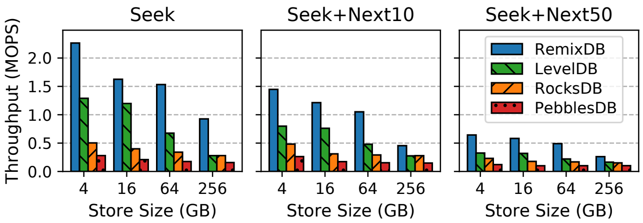


Figure 15: Range query with different store sizes

5.2 Performance of RemixDB

The following evaluates the performance of RemixDB, a REMIX-indexed KV-store based on an LSM-tree.

Experimental Setup We compare RemixDB with state-of-the-art LSM-tree based KV-stores, including Google's LevelDB [26], Facebook's RocksDB [20], and PebblesDB [40]. LevelDB and RocksDB adopt the leveled compaction strategy for balanced read and write efficiency. PebblesDB adopts the tiered compaction strategy with multiple levels for improved write efficiency at the cost of having more overlapping runs.

LevelDB (`v1.22`) supports only one compaction thread. For RocksDB (`v6.10.2`), we use the configurations suggested in its official Tuning Guide[3] [21]. Specifically, RocksDB can have at most three MemTables (one more immutable MemTable than LevelDB). Both RocksDB and RemixDB are configured with four compaction threads. RemixDB, LevelDB, and RocksDB are all configured to use 64 MB table files. For PebblesDB (`#703bd01` [43]), we use the default configurations in its `db_bench` benchmark program. For fair comparisons, we disable compression and use a 4 GB block cache in every KV-store. All the KV-stores are built with optimizations turned on (release build).

In our experiments, we choose three value sizes—40, 120, and 400 bytes. They roughly match the small (ZippyDB, UP2X, USR), medium (UDB, VAR), and large (APP, ETC, SYS) KV sizes in Facebook's production systems [2, 8]. We use 16-byte fixed-length keys, each containing a 64-bit integer using hexadecimal encoding.

Range Query The first set of experiments focuses on how different KV sizes and access patterns affect the search efficiency of the KV-stores. In each experiment, we first sequentially load 100 million KV-pairs into a store using one of the three value sizes. After loading, we measure the throughput of seek operations using four threads with three access patterns, namely sequential, Zipfian ($\alpha = 0.99$), and uniform.

As shown in Figure 14, each set of results shows a similar trend. While RemixDB exhibits the highest throughput, LevelDB is also at least 2× faster than RocksDB and PebblesDB. The sequential loading produces non-overlapping table files in every store, which suggests that a seek operation needs to access only one table file. However, a merging iterator must check every sorted run in the store even though they are non-overlapping, which dominates the execution time of a seek operation if the store has multiple sorted runs. Specifically, each L_0 table in LevelDB and RocksDB is an individual sorted run, but each L_i ($i > 0$) contains only one sorted run; PebblesDB allows multiple sorted runs in every level. That being said, LevelDB outperforms RocksDB by at least 2× even though they both use leveled compaction. We observe that RocksDB keeps several tables (eight in total) at L_0 without moving them into a deeper level during the sequential loading. In contrast, LevelDB directly pushes a table to a deep level (L_2 or L_3) if it does not overlap with other tables, which leaves LevelDB's L_0 always empty. Consequently, a seek operation in RocksDB needs to sort-merge at least 12 sorted runs on the fly, while that number is only 3 or 4 in LevelDB.

The seek performance is sensitive to access locality. A weaker access locality leads to increased CPU and I/O cost on the search path. In each experiment of a particular value size, the throughput with a uniform access pattern is about 50% lower than that of sequential access. Meanwhile, the performance with sequential access is less sensitive to value size because the memory copying cost is insignificant.

The second set of experiments evaluates the range-scan performance with different store sizes and query lengths. Each experiment loads a fixed-size KV dataset with 120 B value size into a store in a random order, then performs range-scans with four threads using the Zipfian access pattern. As shown in Figure 15, RemixDB outperforms the other stores in every experiment. However, the performance differences among the stores become smaller with longer scans. The reason is that a long range-scan exhibits sequential access pattern on each sorted run, where more data have been prefetched during the scan. In the meantime, the memory-copying adds a constant overhead to every store.

As the store size increases to 256 GB, the throughput of LevelDB quickly drops to the same level as RocksDB. Since the stores in the experiments are configured with a 4 GB block cache, the cache misses lead to intensive I/Os that dominate the query time. While RocksDB exhibits high computation cost for having too many L_0 tables with a small store size, the cost is overshadowed by the excessive I/Os in large stores. Meanwhile, RemixDB maintains the best access locality because it incurs a minimal amount of random accesses and cache misses by searching on a REMIX-indexed sorted run.

[3] The configuration for *Total ordered database, flash storage*.

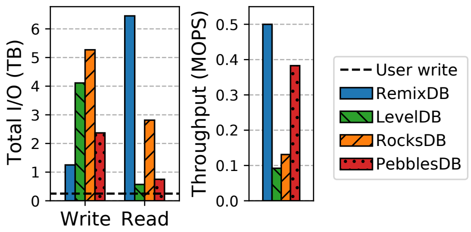


Figure 16: Loading a 256 GB dataset in random order

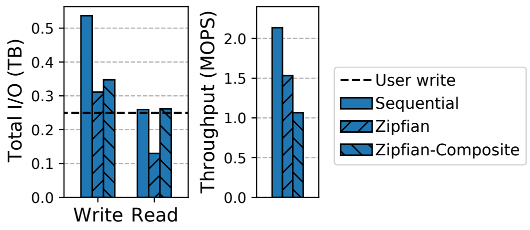


Figure 17: Sequential and skewed write with RemixDB

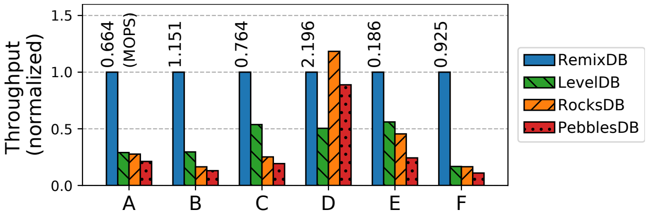


Figure 18: YCSB benchmark results

Write We first evaluate the write performance of each store by inserting a 256 GB KV dataset to an empty store in a random order using one thread. The dataset has 2 billion KV-pairs, and the value size is 120 B. The workload has a uniform access pattern, representing the worst-case scenario of the stores. We measure the throughput and the total I/O on the SSD.

As shown in Figure 16, Both RemixDB and PebblesDB show relatively high throughput because they employ the write-efficient tiered compaction strategy. Their total write I/O on the SSD are 1.25 TB and 2.37 TB, corresponding to WA ratios of 4.88 and 9.26, respectively. LevelDB and RocksDB adopt the leveled compaction strategy, which leads to high WA ratios of 16.1 and 25.6, respectively. RocksDB and RemixDB have much more read I/O than LevelDB and PebblesDB. RocksDB employs four compaction threads to exploit the SSD's I/O bandwidth, resulting in more read I/O than LevelDB due to less efficient block and page cache usage. LevelDB only supports one compaction thread, and it shows a much lower throughput than RocksDB. Although RemixDB has more read I/O than RocksDB, the total I/O of RemixDB is less than that of RocksDB. All told, RemixDB achieves low WA and high write throughput at the cost of increased read I/O.

We further evaluate the write performance of RemixDB under workloads with varying spatial locality. We use three access patterns, namely sequential, Zipfian ($\alpha = 0.99$), and Zipfian-Composite [24]. The Zipfian-Composite distribution represents an agglomerate of attributes in real-world stores [1, 6, 8]. With Zipfian-Composite, the prefix of a key (the first 12 bytes) is drawn from the Zipfian distribution, and the remainder of the key is drawn uniformly at random. For each access pattern, the experiment starts with a 256 GB store constructed as in the random write experiment then performs 2 billion updates (with 128 B values) to the store using the respective access pattern. We measure the throughput and the total I/O during the update phase.

As Figure 17 shows, the sequential workload exhibits the highest throughput because each round of the compaction only affects a few consecutive partitions in the store. The write I/O mainly includes logging and creating new table files, which is about 2× of the user writes. The read I/O for rebuilding REMIXes is about the same as the existing data (256 GB). Comparatively, with the two skewed workloads, the repeated overwrites in the MemTable lead to substantially reduced write I/O. However, the skewed workloads create scattered updates in the key space. This causes slower updates in the MemTable and more partitions being compacted. The Zipfian-Composite workload has weaker spatial locality than Zipfian, resulting in higher compaction I/O cost.

The YCSB Benchmark The Yahoo Cloud Serving Benchmark (YCSB) [13] is commonly used for evaluating KV-store performance under realistic workloads. We use the 256 GB stores constructed in the random-write experiments and run the YCSB workloads from A to F with four threads. The details of the workloads are described in Table 2. In workload E, a *Scan* operation performs a seek and retrieves the next 50 KV-pairs. As shown in Figure 18, RemixDB outperforms the other stores except in workload D, where the read requests (95%) query the most recent updates produced by the insertions (5%). This access pattern exhibits strong locality, and most of the requests are directly served from the MemTable(s) in every store. Meanwhile, LevelDB's performance (1.1 MOPS) is hindered by slow insertions caused by the single-threaded compaction.

Even though REMIXes do not show an advantage over Bloom filters in the micro-benchmarks (see Figure 11c), RemixDB outperforms the other stores in workloads B and C, where point query is the dominant operation. The reason is that a point query in the multi-level LSM-tree has a high cost selecting candidate tables on the search path. Specifically, for each L_0 table, about two key comparisons are used to check if the seek key is covered by the table. If the key is not found at L_0, a binary search is used to select a table at each deeper level L_i ($i \geq 1$) until the key is found. Furthermore, a Bloom filter's size is about 600 KB for a 64 MB table in this setup. Accessing a Bloom filter performs up to seven random memory accesses, which leads to excessive cache misses in a large store [22]. The REMIX-indexed partitions in RemixDB form a globally sorted view, on which a point query can be quickly answered with a binary search.

Table 2: YCSB Workloads

Workload	A	B	C	D	E	F
Operations	R: 50% U: 50%	R: 95% U: 5%	R: 100%	R: 95% I: 5%	S: 95% I: 5%	R: 50% M: 50%
Req. Dist.	Zipfian			Latest	Zipfian	

R: Read, U: Update, I: Insert, S:Scan, M: Read-Modify-Write.

6 Related Work

Improving Search with Filters Bloom filters [4] have been indispensable for LSM-tree based KV-stores in reducing the computation and I/O costs of point queries on a multi-leveled store layout [15]. However, range queries cannot be handled by Bloom filters because the search targets are implicitly specified by range boundaries. Prefix Bloom filters [19] can accelerate range queries [20, 26], but they can only handle closed-range queries on common-prefix keys (with an upper bound). Succinct Range Filter (SuRF) [49] supports both open-range and closed-range queries. The effectiveness of using SuRFs is highly dependent on the distribution of keys and query patterns. Rosetta [37] uses multiple layers of Bloom filters to achieve lower false positive rates than SuRFs. However, it does not support open-range queries and has prohibitively high CPU and memory costs with large range queries. A fundamental limitation of the filtering approach is that it cannot reduce search cost on tables whose filters produce positive results. When the keys in the target range are in most of the overlapping tables, range filters do not speed up queries but cost more CPU cycles in the search path. In contrast, REMIXes directly attack the problem of having excessive table accesses and key comparisons when using merging iterators in range queries. By searching on a globally sorted view, REMIXes improve range query performance with low computation and I/O cost.

Improving Search with Efficient Indexing KV-stores based on B-trees or B+-trees [11, 39] achieve optimal search efficiency by maintaining a globally sorted view of all the KV data. These systems require in-place updates on the disk, which lead to high WA and low write throughput. KVell [35] achieves very fast reads and writes by employing a volatile full index to manage unordered KV data on the disk. However, the performance benefits come at a cost, including high memory demand and slow recovery. Similarly, SLM-DB [31] stores a B+-tree [29] in non-volatile memory (NVM) to index KV data on the disk. This approach does not have the above limitations, but it requires special hardware support and increased software complexity. These limitations are also found in NVM-enabled LSM-trees [32, 48]. Wisckey [36] stores long values in a separate log to reduce index size for search efficiency. However, the approach requires an extra layer of indirection and does not improve performance with small KV-pairs that are commonly seen in real-world workloads [8, 47]. Bourbon [14] trains learned models to accelerate searches on SSTables but does not support string keys. REMIXes are not subject to these limitations. They accelerate range queries in write-optimized LSM-tree based KV stores by creating a space-efficient persistent sorted view of the KV data.

Read and Write Trade-offs Dostoevsky and Wacky [16, 17] navigate LSM-tree based KV-store designs with different merging policies to achieve the optimal trade-off between reads and writes. Tiered compaction has been widely adopted for minimizing WA in LSM-tree based KV-stores [34, 40, 42]. Other write-optimized indexes, such as Fractal trees and B^{ε}-trees, are also employed in KV-store designs [12, 44]. The improvements on write performance often come with mediocre read performance in practice, especially for range queries [24]. REMIXes address the issue of slow reads in tiered compaction. They achieve fast range query and low WA simultaneously.

7 Conclusion

We introduce the REMIX, a compact multi-table index data structure for fast range queries in LSM-trees. The core idea is to record a globally sorted view of multiple table files for efficient search and scan. Based on REMIXes, RemixDB effectively improves range query performance while preserving low write amplification using tiered compaction.

Acknowledgements

We are grateful to our shepherd William Jannen, the anonymous reviewers, Xingsheng Zhao, and Chun Zhao, for their valuable feedback. This work was supported in part by the UIC startup funding and US National Science Foundation under Grant CCF-1815303.

References

[1] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. “LinkBench: a database benchmark based on the Facebook social graph”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD’13)*. 2013, pp. 1185–1196.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. “Workload analysis of a large-scale key-value store”. In: *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMATRICS’12)*. 2012, pp. 53–64.

[3] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. “FloDB: Unlocking Memory in Persistent Key-Value Stores”. In: *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys’17)*. 2017, pp. 80–94.

[4] Burton H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. In: *Commun. ACM* 13.7 (1970), pp. 422–426.

[5] Simona Boboila and Peter Desnoyers. “Write Endurance in Flash Drives: Measurements and Analysis”. In: *8th USENIX Conference on File and Storage Technologies (FAST’10)*. 2010, pp. 115–128.

[6] Dhruba Borthakur et al. “Apache hadoop goes realtime at Facebook”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD’11)*. 2011, pp. 1071–1080.

[7] Lucas Braun et al. “Analytics in Motion: High Performance Event-Processing AND Real-Time Analytics in the Same Database”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD’15)*. 2015, pp. 251–264.

[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. “Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook”. In: *18th USENIX Conference on File and Storage Technologies, (FAST’20)*. 2020, pp. 209–223.

[9] Fay Chang et al. “Bigtable: A distributed storage system for structured data”. In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 1–26.

[10] Guoqiang Jerry Chen et al. “Realtime Data Processing at Facebook”. In: *Proceedings of the 2016 International Conference on Management of Data, (SIGMOD’16)*. 2016, pp. 1087–1098.

[11] Howard Chu. *LMDB: Lightning Memory-Mapped Database Manager*. URL: http://www.lmdb.tech/doc/ (visited on 09/01/2020).

[12] Alexander Conway et al. “SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 2020)*. 2020, pp. 49–63.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking Cloud Serving Systems with YCSB”. In: *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC’10)*. 2010, pp. 143–154.

[14] Yifan Dai et al. “From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI’20)*. 2020, pp. 155–171.

[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. “Monkey: Optimal Navigable Key-Value Store”. In: *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD’17)*. 2017, pp. 79–94.

[16] Niv Dayan and Stratos Idreos. “Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging”. In: *Proceedings of the 2018 International Conference on Management of Data (SIGMOD’18)*. 2018, pp. 505–520.

[17] Niv Dayan and Stratos Idreos. “The Log-Structured Merge-Bush & the Wacky Continuum”. In: *Proceedings of the 2019 International Conference on Management of Data (SIGMOD’19)*. 2019, pp. 449–466.

[18] Giuseppe DeCandia et al. “Dynamo: amazon’s highly available key-value store”. In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP’07)*. 2007, pp. 205–220.

[19] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. “Longest prefix matching using bloom filters”. In: *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM’03)*. 2003, pp. 201–212.

[20] Facebook. *RocksDB*. URL: https://github.com/facebook/rocksdb (visited on 06/11/2020).

[21] Facebook. *RocksDB Tuning Guide*. URL: https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide (visited on 07/12/2020).

[22] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. “Cuckoo Filter: Practically Better Than Bloom”. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT’14)*. 2014, pp. 75–88.

[23] G. E. Forsythe. “Algorithms”. In: *Commun. ACM* 7.6 (1964), pp. 347–349.

[24] Eran Gilad et al. “EvenDB: optimizing key-value storage for spatial locality”. In: *Proceedings of the Fifteenth EuroSys Conference 2020 (EuroSys’20)*. 2020, 27:1–27:16.

[25] Anil K. Goel et al. “Towards Scalable Real-Time Analytics: An Architecture for Scale-out of OLxP Workloads”. In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1716–1727.

[26] Google. *LevelDB*. URL: https://github.com/google/leveldb (visited on 05/03/2019).

[27] Laura M. Grupp et al. “Characterizing flash memory: anomalies, observations, and applications”. In: *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. 2009, pp. 24–33.

[28] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “The Unwritten Contract of Solid State Drives”. In: *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys’17)*. 2017, pp. 127–144.

[29] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. “Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree”. In: *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST’18)*. 2018, pp. 187–200.

[30] Frank K. Hwang and Shen Lin. “A simple algorithm for merging two disjoint linearly ordered sets”. In: *SIAM Journal on Computing* 1.1 (1972), pp. 31–39.

[31] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. “SLM-DB: Single-Level Key-Value Store with Persistent Memory”. In: *17th USENIX Conference on File and Storage Technologies (FAST’19)*. 2019, pp. 191–205.

[32] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “Redesigning LSMs for Nonvolatile Memory with NoveLSM”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 2018)*. 2018, pp. 993–1005.

[33] Donald Ervin Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison-Wesley, 1998.

[34] Avinash Lakshman and Prashant Malik. “Cassandra: a decentralized structured storage system”. In: *Operating Systems Review* 44.2 (2010), pp. 35–40.

[35] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. “KVell: the design and implementation of a fast persistent key-value store”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP’19)*. 2019, pp. 447–461.

[36] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “WiscKey: Separating Keys from Values in SSD-conscious Storage”. In: *14th USENIX Conference on File and Storage Technologies (FAST’16)*. 2016, pp. 133–148.

[37] Siqiang Luo et al. “Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD’20)*. 2020, pp. 2071–2086.

[38] Patrick E. O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O’Neil. “The Log-Structured Merge-Tree (LSM-Tree)”. In: *Acta Informatica* 33.4 (1996), pp. 351–385.

[39] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. “Berkeley DB”. In: *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*. 1999, pp. 183–191.

[40] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. “PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees”. In: *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP’17)*. 2017, pp. 497–514.

[41] Pandian Raju et al. “mLSM: Making Authenticated Storage Faster in Ethereum”. In: *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage’18)*. 2018.

[42] ScyllaDB. *ScyllaDB*. URL: https://github.com/scylladb/scylla (visited on 09/01/2020).

[43] UT Systems and Storage Lab. *PebblesDB*. URL: https://github.com/utsaslab/pebblesdb (visited on 08/03/2019).

[44] Tokutec Inc. *TokuDB*. URL: http://www.tokutek.com (visited on 09/01/2020).

[45] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. “Towards an Unwritten Contract of Intel Optane SSD”. In: *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage’19)*. 2019.

[46] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. “LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data”. In: *2015 USENIX Annual Technical Conference (USENIX ATC 2015)*. 2015, pp. 71–82.

[47] Juncheng Yang, Yao Yue, and K. V. Rashmi. “A large scale analysis of hundreds of in-memory cache clusters at Twitter”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI’20)*. 2020, pp. 191–208.

[48] Ting Yao et al. “MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 2020)*. 2020, pp. 17–31.

[49] Huanchen Zhang et al. “SuRF: Practical Range Query Filtering with Fast Succinct Tries”. In: *Proceedings of the 2018 International Conference on Management of Data, (SIGMOD’18)*. 2018, pp. 323–336.

High Velocity Kernel File Systems with Bento

Samantha Miller Kaiyuan Zhang Mengqi Chen Ryan Jennings
Ang Chen[‡] Danyang Zhuo[†] Thomas Anderson

University of Washington [†]Duke University [‡]Rice University

Abstract

High development velocity is critical for modern systems. This is especially true for Linux file systems which are seeing increased pressure from new storage devices and new demands on storage systems. However, high velocity Linux kernel development is challenging due to the ease of introducing bugs, the difficulty of testing and debugging, and the lack of support for redeployment without service disruption. Existing approaches to high-velocity development of file systems for Linux have major downsides, such as the high performance penalty for FUSE file systems, slowing the deployment cycle for new file system functionality.

We propose Bento, a framework for high velocity development of Linux kernel file systems. It enables file systems written in safe Rust to be installed in the Linux kernel, with errors largely sandboxed to the file system. Bento file systems can be replaced with no disruption to running applications, allowing daily or weekly upgrades in a cloud server setting. Bento also supports userspace debugging. We implement a simple file system using Bento and show that it performs similarly to VFS-native ext4 on a variety of benchmarks and outperforms a FUSE version by 7x on 'git clone'. We also show that we can dynamically add file provenance tracking to a running kernel file system with only 15ms of service interruption.

1 INTRODUCTION

Development and deployment velocity is a critical aspect of modern cloud software development. High velocity delivers new features to customers more quickly, reduces integration and debugging costs, and reacts quickly to security vulnerabilities. However, this push for rapid development has not fully caught up to operating systems, despite this being a long-standing goal of OS research [1, 6, 16, 25, 44]. In Linux, the most widely used cloud operating system, release cycles are still measured in months and years. Elsewhere in the cloud, new features are deployed weekly or even daily.

Slow Linux development can be attributed to several factors. Linux has a large code base with relatively few guardrails, with complicated internal interfaces that are easily misused. Combined with the inherent difficulty of programming correct concurrent code in C, this means that new code is very likely to have bugs. The lack of isolation between kernel modules means that these errors often have non-intuitive effects and are difficult to track down. The difficulty of implementing kernel-level debuggers and kernel testing frameworks makes this worse. The restricted and different kernel programming environment also limits the number of trained developers. Finally, upgrading a kernel module requires either rebooting the machine or restarting the relevant module, either way rendering the machine unavailable during the upgrade. In the cloud setting, this forces kernel upgrades to be batched to meet cloud-level availability goals.

Slow development cycles are a particular problem for file systems. Recent changes in storage hardware (e.g., low latency SSDs and NVM, but also density-optimized QLC SSD and shingle disks) have made it increasingly important to have an agile storage stack. Likewise, application workload diversity and system management requirements (e.g., the need for container-level SLAs, or provenance tracking for security forensics) make feature velocity essential. Indeed, the failure of file systems to keep pace has led to perennial calls to replace file systems with blob stores that would likely face many of the same challenges despite having a simplified interface [2].

Existing alternatives for higher velocity file systems sacrifice either performance or generality. FUSE is a widely-used system for user-space file system development and deployment [17]. However, FUSE can incur a significant performance overhead, particularly for metadata-heavy workloads [48]. We show that the same file system runs a factor of 7x slower on 'git clone' via FUSE than as a native kernel file system. Another option is Linux's extensibility architecture eBPF. eBPF is designed for small extensions, such as to implement a new performance counter, where every operation can be statically verified to complete in bounded time. Thus, it is a poor fit for implementing kernel modules like file systems with complex concurrency and data structure requirements.

Our research hypothesis is that we can enable high-velocity development of kernel file systems without sacrificing performance or generality, for existing widely used kernels like Linux. Our trust model is that of a slightly harried kernel developer, rather than an untrusted application developer as with FUSE and eBPF. This means supporting a user-friendly development environment, safety both within the file system and across external interfaces, effective testing mechanisms, fast debugging, incremental live upgrade, high performance, and generality of file system designs.

To this end, we built Bento, a framework for high-velocity development of Linux kernel file systems. Bento hooks into

Linux as a VFS file system, but allows file systems to be dynamically loaded and replaced without unmounting or affecting running applications except for a short performance lag. As Bento runs in the kernel, it enables file systems to reuse well-developed Linux features, such as VFS caching, buffer management, and logging, as well as network communication. File systems are written in Rust, a type-safe, performant, non-garbage collected language. Bento interposes thin layers around the Rust file system to provide safe interfaces for both calling into the file system and calling out to other kernel functions. Leveraging the existing Linux FUSE interface, a Bento file system can be compiled to run in userspace by changing a build flag. Thus, most testing and debugging can take place at user-level, with type safety limiting the frequency and scope of bugs when code is moved into the kernel. Because of this interface, porting to a new Linux version requires only changes to Bento and not the file system itself. Bento additionally supports networked file systems using the kernel TCP stack. The code for Bento is available at https://gitlab.cs.washington.edu/sm237/bento.

We are using Bento for our own file system development, specifically to develop a basic, flexible file system in Rust that we call Bento-fs. Initially, we attempted to develop an equivalent file system in C for VFS to allow a direct measurement of Bento overhead. However, the debugging time for the VFS C version was prohibitive. Instead, we quantitatively compare Bento-fs with VFS-native ext4 with data journaling, to determine if Bento adds overhead or restricts certain performance optimizations. We found no instances where Bento introduced overhead – Bento-fs performed similarly to ext4 on most benchmarks we tested and never performs significantly worse while outperforming a FUSE version of Bento-fs by up to 90x on Filebench workloads. Bento-fs achieves this performance without sacrificing safety. We use CrashMonkey [34] to check the correctness and crash consistency of Bento-fs; it passes all depth two generated tests. With Bento, our file system can be upgraded dynamically with only around 15ms of delay for running applications, as well as run at user-level for convenient debugging and testing. To demonstrate rapid feature development within Bento, we add file provenance tracking [26, 35] to Bento-fs and deploy it to a running system.

Bento's design imposes some limitations. While Rust's compile-time analysis catches many common types of bugs, it does not prevent deadlocks and or semantic guarantees such as correct journal usage—those errors must be debugged at runtime. While correctness testing is possible at user-level, performance testing generally must be done in the kernel. Also, like other live upgrade solutions, Bento upgrades also require backward-compatibility of the new code with the previous data layout on disk—though the file system itself can perform disk layout changes. The current implementation of Bento imposes some usability limitations similar to FUSE, such as only supporting one mounted file system per inserted file system module. And while we compare Bento-fs performance to ext4, we should note that Bento-fs is a prototype and lacks some of ext4's more advanced features.

In this paper, we make the following contributions:

- We design and implement Bento, a framework that enables high-velocity development of safe, performant file systems in Linux.
- We develop an API that enables kernel file systems written in a type-safe language with both user and kernel execution and live upgrade.
- We demonstrate Bento's benefits by implementing and evaluating a file system developed atop Bento with ext4-like performance, and show that we can add provenance support without rebooting.

2 MOTIVATION

Development velocity is becoming increasingly important for the Linux kernel to adapt to emerging use cases and address security vulnerabilities. In this section, we describe several approaches for extending Linux file systems, and outline the properties of Bento.

2.1 High Velocity is Hard

Linux needs to adapt to support emerging workloads, address newfound vulnerabilities, and manage new hardware. On average 650,000 lines of Linux code are added and 350,000 removed every release cycle, resulting in a growth of roughly 1.5 million lines of code per year. Linux file systems are no exception in needing to adapt — with rapid change in both storage technologies and emerging application demands.

As a concrete example, consider what is needed to add a feature like data provenance to a Linux file system. Increasingly, enterprise customers want to track the source data files used in producing each data analysis output file to perform security forensics. While this might be implemented with existing tools for system call tracking, that would be incomplete — the file system has more comprehensive information (e.g., whether two file paths are hard links to the same inode); a distributed file system can further enable cross-network forensics. To implement this as a new feature in the file system, developers have to modify the file system, test it, and push this modification to production clusters.

The most widely used approach is to directly modify the kernel source code. Linux has standard kernel interfaces for extending its key subsystems — e.g., virtual file systems (VFS) for file systems, netfilter for networking, and Linux Security Module (LSM) for security features. Sometimes, it is also possible to add new features using loadable kernel modules, which can be integrated at runtime without kernel recompilation or reboot. Several VFS filesystems, including ext4, overlayfs, and btrfs, are implemented in the kernel source and can be inserted as loadable kernel modules.

However, high velocity kernel development (including kernel file system development) is hard to come by. To start with, kernel modifications are notoriously difficult to get right. Kernel code paths are complex and easy to accidentally misuse.

Bug	Number	Effect on Kernel
Use Before Allocate	6	Likely `oops`
Double Free	4	Undefined
`NULL` Dereference	5	`oops`
Use After Free	3	Likely `oops`
Over Allocation	1	Overutilization
Out of Bounds	4	Likely `oops`
Dangling Pointer	1	Likely `oops`
Missing Free	18	Memory Leak
Reference Count Leak	7	Memory Leak
Other Memory	1	Variable
Deadlock	5	Deadlock
Race Condition	5	Variable
Other Concurrency	1	Variable
Unchecked Error Value	5	Variable
Other Type Error	8	Variable

Table 1: Low-level bugs in released versions of OverlayFS, AppArmor, and Open vSwitch Datapath between 2014-2018, categorized as memory bugs, concurrency bugs, or type errors, and the likely effect of each bug on kernel operation.

Worse, debugging kernel source code is much harder than user-level debugging. This is because a kernel debugger operates below the kernel, typically remotely, and it cannot leverage Posix APIs such as ptrace. Upgrading kernel modules is also an intrusive operation. In the case of file systems, this requires shutting down applications, unmounting the old file system and remounting the new, and restarting the application. In a multi-tenant cloud setting, most cloud services are upgraded live on a daily or weekly basis. To meet four or five nine application uptime service-level objectives [33] within a reboot model, however, kernel changes need to be batched and applied en masse every few months. Getting needed functionality upstreamed into Linux, so that it is compatible with the 1.5M lines of new code being added each year, takes even longer.

To provide intuition into the difficulty of developing and deploying new kernel features, Table 1 shows an analysis we conducted of bug-fix git commits from 2014-2018 for three modules that modify core Linux functionality used by Docker containers: OverlayFS, AppArmor, and Open vSwitch Datapath. We divide bugs in these systems into two types. One set are semantic bugs in the high-level correctness properties of each module. These can range from mission critical to configuration errors, but generally impair just the functionality of the module. These accounted for 50% of the total bugs fixed in these modules.

The second set concern low-level bugs that are apply to any C language module, but when found in the kernel can potentially undermine the correctness or operation of the rest of the kernel. We categorized these as (1) memory bugs, such as `NULL` pointer dereferences, out-of-bounds errors, and memory leaks; (2) concurrency bugs, such as deadlocks and race conditions; and (3) type errors, such as incorrect usage of kernel types (e.g., interpreting error values as valid data). Of the 50% of fixed bugs that were low-level bugs, we found that 68% are memory bugs. Of these, half are a type of memory leak. Many of the bugs occur in error-handling code, e.g., incorrect checking of return values, missing cleanup procedures. Such bugs are hard to uncover by testing but can lead to serious impacts on the integrity of the kernel. Of all identified low-level bugs, 26% caused a kernel `oops` which either kills the offending process or panics the kernel. An additional 34% of the analyzed bugs result in a memory leak, potentially causing out-of-memory problems or even DoS attack vectors. Many of these low-level bugs, particularly memory and type errors, result from inherent challenges of C code and could be prevented if the programming language had more safety checks.

2.2 Existing Alternatives

Besides directly modifying the Linux kernel, there are two other approaches to adding functionality to Linux, with their respective pros and cons.

Upcall (FUSE [17]): One common technique, particularly for file systems and I/O devices, is to implement new functionality as a userspace server. A stub is left in the kernel that converts system calls to upcalls into the server. Filesystem in Userspace (FUSE) does this for file systems. As opposed to implementing new file system functionality directly in the kernel, this isolates low-level memory errors such as use-after-free to the userspace process. (Low-level bugs can still affect file system functionality, of course.) Development speed is faster because engineers can use familiar debugging tools like gdb. All this comes at a performance cost for metadata-operations [48]. Our evaluation (§5.2) confirms this finding, revealing even worse performance overheads than previously reported, particularly for write-heavy workloads. Additionally, FUSE file systems can't reuse many existing kernel features, such as disk accesses through the buffer cache. Userspace file systems can mitigate the performance overhead by sharing mapped memory with the kernel, but this neither fully removes the performance overhead due to the extra kernel crossing nor allows the file system to access existing kernel functionality.

In-Kernel Interpreter: Using an interpreter inside the kernel for a dynamically loaded program in a safe language is another approach to ensure safety of kernel extensions. Linux supports eBPF (extended Berkeley Packed Filter) [32], an in-kernel virtual machine that allows code to be dynamically loaded and executed in the kernel at predefined points defined by the kernel. eBPF is used heavily for packet filtering, system call filtering, and kernel tracing. The idea is to allow kernel customization in a safe manner. The Linux eBPF virtual machine validates memory safety and execution termination before it JIT compiles the virtual machine instructions into native machine code. As such, eBPF can sandbox untrusted extensions, but the restrictions placed on eBPF make it very difficult to implement larger or more complex pieces of functionality. We argue that untrusted eBPF extensions are not the right model for kernel file system extensibility, as it is particularly difficult to imagine implementing mutable file system operations using eBPF and still enforcing crash consistency.

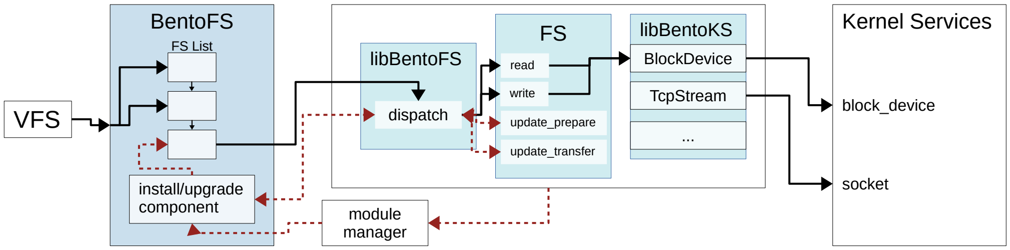


Figure 1: Design of Bento. Shaded components are parts of Bento. BentoFS is in C. The other shaded components are in Rust. Solid black lines represent the common-case operation pathway, detailed in §3.2 and §3.3. Dashed red lines represent the install/upgrade pathway and are described in §3.4

2.3 Our Approach: Bento

We have designed Bento for high velocity development of Linux file systems with the following properties:

- **Safety:** Any bugs in a newly installed file system should be limited, as much as possible, to applications or containers that use that file system.
- **Performance:** Performance should be similar to that of the same functionality implemented using VFS.
- **Generality:** There are a large variety of file system designs that developers might want to implement. Bento should not limit the types of file systems that can be developed.
- **Compatibility:** File systems added to Linux via our framework should work with existing application binaries without recompiling or relinking. Further, Bento should not require substantial changes to Linux's internal architecture, to make Bento easier to upstream.
- **Live upgrades:** The framework should support dynamic upgrades to running file system code, transparently to applications, except for a small delay.
- **User-level debugging:** File system code should be easily migrated between userspace and the kernel to enable user-level debugging and correctness testing.

At a high level, Bento achieves the first three goals by enabling developers to write file systems in Rust, a type-safe, non-garbage collected, general-purpose language that is receiving increasing attention for kernel implementations. Of course, safely using Rust within Linux is a challenge of its own. The other three goals are achieved via careful architectural design. To provide *compatibility* without sacrificing *safety*, Bento avoids directly using the Linux VFS interface, because it requires data structures to be directly passed back and forth between the file system and the kernel, making it difficult to provide verifiable data structure ownership safety. Instead, Bento introduces a message-passing based API for file systems that enforces ownership safety. Second, Bento introduces a different API to enable safe access to C-language kernel services, by translating unsafe kernel interfaces into ones that can be safely used by Rust. For *live upgrades*, Bento includes a component that quiesces the running file system and then transfers file system-defined state to the new instance, passing ownership of long-lived, in-memory data structures between the file systems so they can be shared across the upgrade. For *user-level debugging*, Bento is designed with the same set of API calls whether it runs in the kernel or in the userspace. A simple build flag change is sufficient to choose a different mode.

2.4 Rust Primer

As background, Rust is a strongly-typed, memory safe, data race free, non-garbage collected language. With these properties, Rust is able to provide strong safety guarantees without high performance overhead or the performance unpredictability caused by garbage collectors. These provide useful building blocks for Bento.

Rust relies on its type system to enforce memory safety. The type system restricts how objects can be created and cast, so if an object exists and is of a certain type, this guarantees that the memory backing the object is valid and correctly represents that type. Since raw pointers can be `NULL` and can be cast to nonequivalent types, dereferencing pointers and creating strongly type objects from pointers is unsafe and must be tagged as `unsafe` to compile. Calling unsafe functions is additionally unsafe. Although some systems allow unsafe Rust, Bento requires that its file systems contain no unsafe code.

Rust prevents most memory leaks by tracking the lifetime of objects. All objects must be owned by one variable at a time. When the variable owning an object goes out of scope, the lifetime of the object is over and the memory backing the object can be safely reclaimed. References allow other variables to refer to data without claiming ownership of the memory. References are either immutable or mutable, enabling read-only or read-write accesses, respectively; references cannot outlive the owner. Developers can provide custom functionality to be performed when an object goes out of scope by implementing the `drop` method. Leaking memory is not a safety violation in Rust, so the `drop` function is not guaranteed to be called, but memory leaks must be explicit instead of accidental.

Data races are avoided by enforcing that all objects, except those that can be safely modified concurrently, must only have one mutable reference at a time. For non-thread safe objects that must be shared between threads, synchronization mechanism such as locking must be used to safely obtain references. Acquiring the lock gives the caller access to the underlying data. Lock acquisitions methods generally return a guard that automatically unlocks the lock in `drop`, preventing the caller from forgetting to unlock. However, deadlocks, such as circular waiting for locks, are possible in safe Rust code as preventing them is beyond the power of the Rust type system.

These represent about 7% of the low-level bugs found in our analysis of popular kernel modules.

3 THE BENTO SYSTEM

In this section, we describe the architecture of Bento, explain how it interfaces with VFS and the rest of the kernel, and detail how it enables live upgrades and user-level debugging.

3.1 The System Architecture

Figure 1 shows the Bento architecture; the shaded portions are the Bento framework. Bento is a thin layer that, to the rest of Linux, operates like a normal VFS file system. The Linux kernel is unmodified other than the introduction of Bento. In turn, like VFS, Bento defines a set of function calls that Bento file systems implement and provides a mechanism for file systems to register themselves with the framework by exposing the necessary function pointers. Unlike VFS, Bento is designed to support file systems written in safe Rust.

Bento consists of three components. First, BentoFS interposes between VFS and the file system module and acts as a controller that manages registering and running file systems. BentoFS is written in C and inserted as a separate kernel module. The other two components are Rust libraries that are compiled into the file system module. LibBentoFS translates unsafe calls from BentoFS into the safe file operations API that is implemented by the file system. LibBentoKS provides a safe API for file systems to access kernel services, such as to perform I/O. The file system itself is written in safe Rust and is compiled as a Rust static library that includes libBentoFS and libBentoKS. When a file system module is loaded, it registers itself with BentoFS which adds it to the list of active file systems.

3.2 Interacting with VFS

The VFS layer poses a fundamental challenge to memory safety. For example, VFS file systems allocate a single inode data structure to hold both VFS and file system-specific data. When the kernel needs a new inode, it requests one from the file system which allocates it from its own memory pool. Both sides access their half of the data structure, and when done, the kernel releases the inode to the file system so the memory can be reclaimed. Independent of whether this is a good design pattern for minimizing kernel memory errors, it is inconsistent with Rust compile time analysis and therefore would compromise our ability to prevent memory safety errors within the file system code itself.

Instead, we define a new interface for safe kernel file systems. A selection of this API is in Table 2; the rest in the appendix. The BentoFS module receives all calls from the VFS layer, determines which mounted file system is the target, and handles any necessary operations on kernel data structures. BentoFS then sends requests to the libBentoFS dispatch function using a similar API to that of the file system, but with unsafe pointers instead of Rust data structures. LibBentoFS parses the request, converts pointers to safe data structures, and calls the correct function in the file system. The key idea is that the file system's compiler can statically verify its own data accesses, including its inode. To create an inode, BentoFS calls into the file system (via libBentoFS) and gets back an opaque reference (the inode number). In turn, BentoFS allocates and returns to VFS a separate kernel inode data structure. BentoFS never touches the contents of the file system inode.

Bento File Operations API (partial)
bento_init(&mut self, req, devname, fc_info)
bento_destroy(&mut self, req)
bento_read(&self, req, ino, fh, offset, size, reply)
bento_write(&self, req, ino, fh, offset, data, flags, reply)
bento_update_prepare(&mut self) -> Option<TransferOut>
bento_update_transfer(&mut, Option<TransferIn>)

Table 2: A subset of the Bento File Operations API. *req* includes the user application's uid, gid, and pid. *reply* includes data or error values. The full API is included in supplementary material.

BentoFS and libBentoFS are responsible for ensuring that Rust's safety properties are maintained as memory is passed across the File Operations API so the assumptions made by the Rust compiler will be true. When passing references to kernel memory to the file system, such as data for read and write calls, BentoFS guarantees that the memory will remain valid until the call completes and, if a mutable reference is passed, must ensure that no other thread is modifying the memory. When passing references to structured data, BentoFS and libBentoFS also ensure that the memory is correctly structured and never cast to an incompatible type. Passing ownership across the File Operations API requires careful handling of the memory in libBentoFS and is only done during live upgrade (§3.4).

3.3 Interacting with Kernel Services

Bento file systems need access to kernel functionality such as block I/O for access to underlying storage devices. These kernel interfaces, like those in the VFS layer, are not designed with type safety in mind and so cannot be directly used by a Bento file system. Instead, libBentoKS implements safe versions of kernel data structures and functions needed by file systems.

As an example, we will focus on kernel block I/O. File systems in Linux access block devices via the buffer cache. To read from (or write to) a block device, a Linux file system calls `__bread_gfp`, passing in a pointer to the `block_device` data structure, a block number, the block size, and a page allocation flag. This function returns a `buffer_head` data structure representing the requested block. The block's data is represented as a pointer and size in the `buffer_head`. The file system can then read and/or write to this memory region. When the file system is done using the `buffer_head`, it must call `brelse` or buffers can be leaked.

Like many kernel interfaces, kernel block I/O relies heavily on pointers. However, as described in §2.4, raw pointers cannot be deferenced in safe Rust, and directly exposing these pointers to the file system results in safety errors. If the block I/O functions exposed to the file system accept a pointer, the block I/O functions cannot be marked safe and the file system as a whole cannot be safe.

Exposing kernel services safely. Bento provides wrapping abstractions for kernel services so they can be used safely by the file system. These abstractions can be used like any other Rust data structures and functions. Several of the provided abstractions are detailed in Table 3.

To be concrete, we address the example discussed above. We provide a safe `BlockDevice` abstraction to represent a kernel block device. A `BlockDevice` takes the name of the block device file and the block size; it contains a pointer to the kernel block device and the block size as fields. It provides several methods, including a safe `bread` method that takes a block number as an argument, performs safety checks, and calls `__bread_gfp` using the correct page allocation flag. The `bread` method returns a `BufferHead` that wraps the kernel `buffer_head`. A `BufferHead` method converts the pointer and size fields into a sized memory region that can be used safely. That method must use unsafe code to make the sized memory region out of the unsized pointer and size fields, but the file system can call the method safely. To prevent accidental memory leaks, we call the `brelse` function in the `drop` method of the `BufferHead` wrapper. With this, buffer management has the same properties as memory management in Rust: memory leaks are possible but difficult.

LibBentoKS provides synchronization primitives including `RwLock<T>`, a wrapper around the kernel read-write semaphore. It has the same interface as the Rust standard library `RwLock<T>`, a read-write lock that protects data of type `T`. To obtain an immutable reference to the protected data, the user must acquire the read lock; to obtain a mutable reference, the user must acquire the write lock. `ReadGuard` calls `up_read` in `drop` and `WriteGuard` calls `up_write` in `drop`, preventing the user from forgetting to unlock.

In addition libBentoKS provides an implementation of the Rust global allocator that uses `kmalloc` and `kfree` for small regions (less than 8 pages) and uses `vmalloc` and `vfree` for larger regions. In this way, file system developers can use dynamically allocated types such as a growable array (Rust's `alloc::vec::Vec`) and collection types (from Rust's `alloc::collections`). LibBentoKS provides `TcpStream` and `TcpListener` to support networked file systems.

These abstractions can, in some cases, add a small amount of performance overhead. If a kernel function has requirements on its arguments, the wrapping method likely will need to perform a runtime check to ensure that the requirements hold.

3.4 File System Upgrade

To enable online upgrades that are transparent to applications using the file system, we must first identify when it is safe to upgrade the file system and how to handle long-lived file system state. If an upgrade occurs while file system operations are still pending, there may be race conditions where some operations are executed on the old file system and others on the new, leading to correctness problems. In addition, any state that affects the semantic behavior of the file system, such as in-progress disk requests, file system journals, and TCP connections for networked file systems, must be correctly preserved across the upgrade. State that affects performance but not semantics, such as clean data in caches, can be optionally preserved.

Bento addresses these challenges by ensuring that the old file system is in a quiescent state and that semantic state is transferred to the new file system. Bento quiesces the file system by pausing new calls into the file system module during the upgrade and waiting for in progress operations to complete. To achieve this, Bento uses a read-write lock on the file system connection. All calls into libBentoFS acquire the read lock, while upgrades acquire the write lock. Therefore, file system operations can be executed concurrently in normal mode but will be blocked during an upgrade; the upgrade will be blocked until previous operations complete.

Second, a constraint on the old file system is that it must be able to transfer its semantic state to the new file system. Of course, the specific content of this state will vary from file system to file system. Each file system defines two data structures: one that is returned when the file system is removed and one that is expected when the file system is replacing a previous live file system. This design pattern, of needing to write code to support both past and future versions, is common in cloud settings. During upgrade, ownership of the data structure is passed from the old file system to the new one. BentoFS handles passing the data structure from the old file system to the new file system. The detailed mechanisms involved for live upgrades are shown in Figure 1 and described below:

1. A new file system upgrade instance is loaded into the kernel. At module load, it calls into BentoFS to register itself and indicate that it is an upgrade.
2. BentoFS identifies the file system that needs to be unloaded and acquires the lock to pause new operations and wait for existing operations to complete.
3. BentoFS sends a `bento_update_prepare` request to the old file system through libBentoFS.
4. The old file system instance handles the `bento_update_prepare` request, performing any necessary cleanup and creating and returning its defined output state transfer struct to BentoFS through libBentoFS.
5. BentoFS sends a `bento_update_transfer` request to the new file system through libBentoFS, passing the state transfer data structure to the new file system.
6. The new file system instance initializes itself using the provided state and returns.
7. BentoFS modifies the connection state by replacing the old file system reference with the new file system reference and releases the write lock, allowing calls to proceed to the new instance.

3.5 Userspace Debugging Support

Bento also introduces a feature that enables a new file system to be seamlessly hoisted to userspace for debugging. This enables developers to leverage gdb and other familiar utilities

Object Type	Method	Kernel Equivalent	Description
BlockDevice	bread(&self, ...) -> Result<BufferHead>	__bread_gfp(...)	Read a block from disk
	getblk(&self, ...) -> Result<BufferHead>	__getblk_gfp(...)	Get access to a block
	sync_all(&self) -> Result<i32>	blkdev_issue_flush(...)	Flush the block device
BufferHead	data(&self) -> &[u8]	buffer_head->b_data	Get read access to data
	data_mut(&mut self) -> &mut [u8]	buffer_head->b_data	Get write access to data
	drop(&mut self)	brelse(...)	Release the buffer
	sync_dirty_buffer(&mut self) -> Result<c_int>	sync_dirty_buffer(...)	Sync a block
GlobalAllocator	alloc(&self, ...) -> *mut u8	__kmalloc(...)/vmalloc(...)	Allocate memory
	dealloc(&self, ...)	kfree(...)/vfree(...)	Free allocated memory
RwLock<T>	new(data:T) -> RwLock<T>	init_rwsem(...)	Create a RwLock of type *T*
	read(&self) -> LockResult<ReadGuard<'_T>>	down_read(...)	Acquire the read lock
	write(&self) -> LockResult<WriteGuard<'_T>>	down_write(...)	Acquire the write lock

TcpStream	connect(addr: SocketAddr) -> Result<TcpStream>	sock_create_kern(...)
		kernel_connect(...)
			Create and connect

	read(&mut self, ...) -> Result<usize>	kernel_recvmsg(...)	Read a message
	write(&mut self, ...) -> Result<usize>	kernel_sendmsg(...)	Send a message
	drop(&mut self)	sock_release(...)	Cleanup the TcpStream

TcpListener	bind(addr: SocketAddr) -> Result<TcpListener>	sock_create_kern(...)
		kernel_bind(...)
		kernel_listen(...)
			Create, bind, and listen

	accept(&self) -> Result<(TcpStream, SocketAddr)>	kernel_accept(...)	Accept a connection

Table 3: Kernel Services API. These are some of the data structures and methods provided to the file system. Methods that take *&mut self* can modify the object. Methods that take *&self* can access but not modify the object.

for higher velocity development. Debugged code can then be dropped back into the kernel without any modification. Bento supports this feature by exposing identical interfaces to both the kernel version and the userspace version of a developed file system. Whether the file system runs in the kernel or at userspace is determined by a compilation configuration flag which specifies which libraries will be linked and how the file system should register itself during initialization.

Our solution leverages Linux kernel FUSE support to forward file operations to userspace. By itself, this is not sufficient — a FUSE file system is not runnable in the kernel. At a high level, we design our kernel interfaces to mirror existing userspace interfaces when possible, and implement userspace libraries to expose additional abstractions otherwise.

Many kernel interfaces can be designed to expose the same interfaces as userspace abstractions. For example, kernel read-write semaphores are used the same way as Rust's `std::sync::RwLock<T>` and the kernel TCP stack provides similar interfaces to Rust's `std::net::TcpStream` and `std::net::TcpListener`. In these cases, our kernel services API provides interfaces that are identical to the analogous userspace interface.

However, some kernel interfaces do not have obvious userspace analogues. The File Operations API (Table 2), for example, adds functions to implement state transfer and passes immutable references to ensure correct concurrency behavior. Additionally, operations on the backing storage device are performed differently from the kernel and userspace. FUSE file systems typically use file I/O to access the storage device while kernel file systems directly interface with the kernel buffer cache. Using a file I/O interface in the kernel would significantly hinder performance and functionality, adding extra data copies and preventing certain optimizations. However, there is no standard userspace abstraction that closely mirrors the kernel buffer cache.

To address this, we provide two additional libraries The userspace version of libBentoFS translates calls from FUSE into the File Operations API. The userspace version of libBentoKS implements a basic buffer cache that uses file I/O under the hood, providing the `BlockDevice` and `BufferHead` abstractions to Bento file systems when running at user level.

4 IMPLEMENTATION & EXPERIENCES

We have developed Bento as a Linux kernel module for BentoFS and a Rust library containing both libBentoKS and libBentoFS in 5240 lines of C and 5072 lines of Rust. The userspace versions of libBentoKS and libBentoFS are another 986 lines of Rust. The current implementation targets Linux kernel version 4.15. The file system is compiled as a Rust a static library, which can be linked with any required C code to generate the .ko kernel module. Kernel code in Rust cannot use standard libraries, but we do enable use of the Rust alloc crate.

4.1 BentoFS

We built BentoFS by modifying the existing Linux FUSE kernel module. In place of upcalls, BentoFS communicates with libBentoFS using function calls. A file system module registers itself with BentoFS by providing a pointer to the `dispatch` function when it is mounted. Like the VFS layer, BentoFS maintains a list of active file systems, locking the list and adding and removing entries when file systems are registered or unregistered. This list is additionally locked during a live upgrade.

Upgrade State Transfer. Ownership of state transfer data structures must be moved between the Rust file system modules during an upgrade to allow the new file system instance to take ownership of state owned by the old file system instance. We implement this ownership transfer in libBentoFS using the Rust *Box* type. When the old file system instance returns its state to libBentoFS, we create a *Box* to take ownership of the data and pass the box as a raw pointer to BentoFS. The new libBentoFS converts the pointer back to a *Box*, claiming ownership of the data before passing it to the file system. Rust deletes the old file system data structure when it goes out of scope at the end of the transfer; the old file system is uninstalled in the background.

4.2 Experiences Using Bento

We began this project developing both a Bento version of a file system and its VFS equivalent in C, as a way to quantify the performance cost of Bento. However, we eventually stopped development on the VFS version because implementing and debugging new features were significantly more time consuming and difficult than for the Bento version. In VFS, we were much more likely to accidentally write memory errors, such as `NULL` pointer dereferences and memory leaks. These bugs took much longer to diagnose and fix than bugs in the Bento version because they would crash the kernel, forcing us to reboot between tests, and they were difficult to isolate.

We further illustrate our experience developing with Bento on three axes: functionality, performance, and correctness.

Functionality. Using Bento, we implemented Bento-fs, a file system designed to have ext4-like performance, in 3038 lines of safe Rust code. Bento-fs is structurally similar to the xv6 file system, a simple file system included in MIT's teaching operating system xv6 [12]. This simplicity made the xv6 file system an attractive starting point for our prototype. Bento-fs includes several modifications for improved functionality and performance. For example, xv6 does not fully support the functionality necessary to run our benchmarks. Likewise, we added double indirect blocks to support files up to 4GB, instead of 4MB in xv6.

We also added a provenance feature to Bento-fs. The architecture of provenance tracking is borrowed from existing work [26, 35]. It consists of two pieces: a) a file system component that tracks file creations, deletions, and opens; and b) a syscall-level component that tracks the process hierarchy and operations on open file descriptors, such as dup and sendmsg.

The file system-level component is implemented by logging information to a special file. To track existing files, 'create', 'rename', 'symlink', and 'unlink' operations log the user process ID of the request, the names and inode numbers of relevant files, any request flags, and, for 'unlink', whether or not the file was deleted. The current implementation does not track hard links, but adding such support could follow a similar strategy. Since Bento-fs is not called for every read or write operation due to kernel caching, we track file accesses by logging 'open' and 'close' calls, recording the read/write mode of the open call along with the process ID of the request and the inode number of the file. If a file is opened as writable while another file is opened as readable, provenance tracking assumes that the writable file's contents depends on the readable file's contents.

The syscall-level component tracks process creation through 'fork'/'exec' and operations on open file descriptors so the provenance system can correctly handle instances where a process gains access to a file without using the open syscall. This component is implemented as a collection of eBPF programs that log the relevant system calls, namely 'clone', 'exec', 'pipe', 'dup', 'dup2', and 'sendmsg'. 'Open' calls are also logged so the file descriptors used in the system calls can be matched to the file system tracking on file names.

Overall, these features were added to Bento-fs in 145 lines of code in two weeks of development. In our development process, we never caused a crash of the operating system and were able to test and debug code within minutes of making changes. In fact, many of our changes worked correctly once they compiled, something that has not been true of our C development.

Performance. To be able to bound the overhead imposed by Bento by comparing it to ext4, we added various optimizations to Bento-fs to match ext4 behavior. We particularly noticed overhead on multi-threaded and metadata intensive benchmarks. The xv6 free inode and free block implementations, for example, are needlessly inefficient. The journal used by xv6 is small by default and assumes that each operation will use the maximum number of blocks, limiting it to only three concurrent operations at once. It also commits operations to the device synchronously when transactions are completed. We increased the size of the log and leveraged the Linux journal module JBD2 (also used by ext4). In JBD2, transactions request the required number of blocks and commit in the background. [1]

Similarly, xv6 uses an inefficient list structure for directories. We added tree-structured directories that use the hash of the file name to locate directory entries.

Most of the code changes for the journal modifications were in libBentoKS and mkfs. Tree structured directories were implemented within Bento-fs in around 800 lines of code, split across utility functions for the hash tree and directory lookup, linking, and reading. Having access to dynamically allocated data structures from Rust's `alloc` crate simplified this implementation. The tree structure uses the B-tree implementation provided by the crate and the directory lookup, linking, and reading code use Rust's dynamically allocated array `Vec`.

Correctness. We tested the correctness of our file system using CrashMonkey [34]. It generates workloads based on operations supported by the file system, and exhaustively tests all combinations up to a defined sequence length. We ran the `seq-2` benchmarks [34], which test sequences of two operations, using the operations supported by Bento-fs. This resulted in 47314 benchmarks in total. CrashMonkey did

[1] Although we implemented a log manager for the userspace version, it is likely less optimized than the kernel version, and there may be additional ways to improve userspace write performance that we have not yet discovered.

not find any crash consistency bugs in Bento-fs. It found a known bug from the FUSE kernel module in the C code used in BentoFS where opening a directory then calling `rmdir` followed by `mkdir` on the directory name before closing it resulted in an unusable directory due to inode reuse. We fixed this by always allocating a new inode during directory creation.

The provenance extension to Bento-fs was also used by two groups of students to create two applications in the context of a class. One of these applications automatically recreated derived files when input files changed, specifically recompiling an executable based on the input C files, inspired by past work on transparent make [47]. The other application performed automatic directory synchronization, syncing files in a local directory to remote storage. In these student projects, we found that Bento was robust enough to support a smooth development experience.

5 EVALUATION

Our evaluation of Bento aims to answer several questions: a) How well does Bento-fs perform on different workloads? b) How robust is the file system under crash consistency testing? and c) How expensive are live upgrades?

5.1 Experimental setup

Baselines. We compare: a) ext4-o: ext4, the default file system on most Linux versions, using the default `data=ordered` option with metadata journaling, b) ext4-j: ext4 with data journaling (`data=journal` mode) c) Bento-fs, and d) Bento-fs running in userspace. We focus our evaluation on ext4 with journaling because Bento-fs also implements data journaling. Note that Bento-fs has implemented only a subset of ext4's optimizations. The userspace version of Bento interacts with the storage device by opening it with the `O_DIRECT` flag.

Environment. All experiments were run on a machine with Intel Xeon Gold 6138CPU (2 sockets, each with 20 cores, 40 hyperthreads), 96 GB DDR4 RAM, and a 480 GB Intel Optane SSD 900P Series with 2.5 GB/s sequential read speed and 2 GB/s sequential write speed. All benchmarks were run using the SSD as the backing device using the cores and memory on the socket connected to the SSD.

5.2 Microbenchmarks

We ran microbenchmarks from the Filebench benchmarking suite. The workloads included sequential read, random read, sequential write, random write, and create and delete benchmarks. All workloads except for sequential write are run with both 1 thread and 40 threads. Read and write benchmarks were executed on a 4GB file using four different operation sizes: 4, 32, 128, and 1024KB. The create workloads create 800,000 16KB files in the same directory, allocating half before the start of the benchmark. The delete workloads delete 300,000 16KB files across many directories, with an average of 100 files per directory. All benchmarks were run 10 times, and averages and standard deviation were calculated. Table 4 shows the results on ext4 with both the default metadata journaling and data journaling, Bento-fs, and Bento-user, the userspace version of Bento-fs. Results are colored based on the performance compared to ext4.

Reads. Reads on all three file systems have similar performance for all sizes and both single-threaded and 40-threaded, and large reads achieve greater bandwidth than provided by the device. This is because data is cached quickly after the first read, and all subsequent reads hit in the page cache. The userspace version uses the kernel cache in the FUSE kernel module before forwarding requests to userspace, so it performs similarly to direct kernel implementations.

Writes. For small write benchmarks, Bento-fs and ext4-j have fairly similar write performance. Bento-fs has higher performance than ext4-j and similar performance to ext4-o on large write benchmarks due to slight implementation differences. Whereas ext4-j logs blocks to the journal on the write syscall path, Bento-fs logs asynchronously in the writeback cache when data is flushed. This performance difference is more prominent for single-threaded benchmarks with large writes because these are more likely to stress the journal in ext4-j without stressing the writeback cache. For all cases, the user-level implementation is much slower because it incurs additional kernel crossings and issues block I/O from userspace. Each operation must first pass from the kernel back to the userspace, which will then be translated into several read/write operations on the storage device. Each system call to the device file must in turn pass through the VFS layer to reach the kernel block cache; this is much slower than direct accesses to the kernel block cache by a kernel file system. Additionally, Bento-user does not have access to the JBD2 module, so it uses a simpler journal that is less efficient on large write workloads. This journal is also affected by slow userspace block I/O.

Creates+Deletes. On the create and delete benchmarks, ext4-j and Bento-fs have similar performance. Bento-fs outperforms ext4-j on single-threaded creates, likely due to the write speedup. Ext4-o outperforms Bento-fs on multi-threaded creates. Both ext4 modes and Bento-fs outperform the user-level file system for the same reason as the write benchmarks.

5.3 Application Workloads

Next, we run three application-style workloads from Filebench, four applications, and two workloads each on two different key-value stores. All workloads were run 10 times and averages and standard deviation were calculated. From Filebench, we ran 'varmail', 'fileserver', and 'webserver'. (1) The 'varmail' mail-serving workload uses 16 threads to create and delete 1000 files in one directory and performs reads and writes followed by fsyncs to these files. (2) The 'fileserver' file-serving workload uses 50 threads to create and delete 10,000 files across 500 directories and executes reads and appends to these files. (3) The 'webserver' web-serving workload uses 100 threads to read from 1000 small (16KB average size) files across around 50 directories and append to an operation log. All benchmarks execute for one minute. For application

Benchmark	ext4-o	ext4-j	Bento-fs	Bento:ext4-j	Bento-user	user:ext4-j
seq. read, 1-t, 4k	286 (±2)	287 (±2)	289 (±4)	1.01	290 (±2)	1.01
seq. read, 1-t, 32k	1811 (±20)	1796 (±21)	1817 (±18)	1.01	1807 (±18)	1.00
seq. read, 1-t, 128k	4170 (±55)	4071 (±75)	4119 (±82)	1.01	4112 (±50)	1.01
seq. read, 1-t, 1024k	6434 (±129)	6580 (±197)	6730 (±197)	1.02	6510 (±160)	0.99
seq. read, 40-t, 4k	429 (±7)	433 (±9)	436 (±7)	1.00	429 (±9)	0.99
seq. read, 40-t, 32k	3372 (±65)	3561 (±332)	3488 (±184)	0.98	3417 (±56)	0.96
seq. read, 40-t, 128k	17668 (±143)	17878 (±162)	17784 (±132)	0.99	17833 (±168)	1.00
seq. read, 40-t, 1024k	21407(±1774)	22024 (±101)	22082 (±339)	1.00	22136 (±101)	1.00
rand. read, 1-t, 4k	150 (±1)	149 (±2)	149 (±2)	1.01	149 (±3)	1.00
rand. read, 1-t, 32k	1037 (±6)	1044 (±6)	1049 (±8)	1.00	1041 (±6)	0.99
rand. read, 1-t, 128k	2901 (±20)	2955 (±36)	2957 (±33)	1.00	2908 (±31)	0.98
rand. read, 1-t, 1024k	5836 (±68)	5961 (±152)	5967 (±116)	1.00	5890 (±131)	0.99
rand. read, 40-t, 4k	223 (±24)	211 (±2)	217 (±5)	1.02	218 (±5)	1.02
rand. read, 40-t, 32k	1717 (±34)	1712 (±34)	1737 (±37)	1.01	1738 (±31)	1.02
rand. read, 40-t, 128k	9265 (±104)	9232 (±70)	9206 (±132)	1.00	9224 (±55)	1.00
rand. read, 40-t, 1024k	21635 (±46)	21650 (±49)	21637 (±50)	1.00	21569 (±54)	1.00
seq. write, 1-t, 4k	234 (±7)	172 (±3)	252 (±6)	1.46	3.7 (±0.0)	0.02
seq. write, 1-t, 32k	860 (±86)	409 (±1)	1003 (±65)	2.45	4.0 (±0.1)	0.01
seq. write, 1-t, 128k	1058 (±109)	430 (±44)	1774 (±352)	4.12	4.0 (±0.1)	0.01
seq. write, 1-t, 1024k	1365 (±0)	469 (±62)	1843 (±329)	3.93	4.0 (±0.0)	0.01
rand. write, 1-t, 4k	142 (±3)	120 (±1)	139 (±2)	1.16	8.5(±0.14)	0.07
rand. write, 1-t, 32k	875 (±7)	395 (±22)	898 (±9)	2.27	10.1 (±0.0)	0.03
rand. write, 1-t, 128k	1952 (±16)	330 (±18)	2167 (±62)	6.55	10.3 (±0.1)	0.03
rand. write, 1-t, 1024k	3051 (±35)	309 (±8)	3789 (±56)	12.24	10.1 (±0.3)	0.03
rand. write, 40-t, 4k	230 (±3)	208 (±4)	241 (±14)	1.15	9.2 (±0.1)	0.04
rand. write, 40-t, 32k	1237 (±46)	357 (±61)	1500 (±34)	4.20	10.0 (±0.2)	0.03
rand. write, 40-t, 128k	1414 (±43)	303 (±10)	1894 (±39)	6.24	10.4 (±0.1)	0.03
rand. write, 40-t, 1024k	1391 (±49)	296 (±13)	1924 (±78)	6.50	11.0 (±0.0)	0.04
create, 1-t, ops/s	12510 (±418)	8564 (±186)	12087 (±390)	1.41	194 (±5)	0.02
create, 40-t, ops/s	34377(±2157)	17858 (±0)	18819 (±663)	1.05	216 (±2)	0.01
delete, 1-t, ops/s	23331 (±878)	22913 (±0.3)	24997 (±0)	1.09	827 (±11)	0.03
delete, 40-t, ops/s	60493(±7088)	63253(±7101)	57253(±6258)	0.91	808 (±27)	0.01

Table 4: Performance results for ext4 in `data=ordered` mode (ext4-o), and `data=journal` mode (ext4-j), Bentofs, and a userspace version of Bento-fs (Bento-user) on Filebench microbenchmarks using varying operation sizes and 1 and 40 threads. Reads and writes are measured in MBps. Reads and writes are cached in the kernel and so can outperform the 2.5 GBps and 2.0 GBps device read and write speed. Results are averaged over 10 runs and standard deviations are included in parentheses. Color indicates performance relative to ext4-j. Bento-fs performs similarly to ext4-j for most benchmarks. Both significantly outperform Bento-user.

workloads, we used 'tar', 'untar', and 'grep' on the Linux kernel source code and 'git clone' on the xv6 source repository.

We also evaluate read and write workloads on the Redis [41] and RocksDB [43] key-value stores. Redis is an in memory key-value store used in distributed environments. By default, it periodically dumps the database to a file but can be configured to also log all operations to an append-only-file (AOF) for persistence. In our evaluation, we use the AOF and configure it to sync every second. We run the 'set' and 'get' workloads from redis-benchmark, the provided benchmarking utility, for 1,000,000 operations using 100B values. RocksDB is a persistent key-value store developed by Facebook based on Google's LevelDB [14]. Using db_bench, the included benchmarking utility, we evaluate the 'fillrandom' and 'readrandom' workloads each for 1,000,000 operations using 100B values.

Filebench: Figure 2 presents the application-style Filebench results for the three file systems described earlier, plus Bento-fs with file provenance (Bento-prov). Across all benchmarks, Bento-fs (with or without provenance) outperforms Bento-user by 10-400x due to the reasons discussed earlier. For varmail and webserver, ext4-j and Bento-fs exhibit similar performance, but for fileserver, Bento-fs significantly outperforms ext4-j due to an unintentional quirk in the benchmark. Filebench 'fileserver' executes many sequences of create-write-delete operations, but it does not sync the file before the file is deleted. With writeback caching, Bento recognizes that the pages belong to files that no longer exist, and drops the writes. In ext4-j, on the other hand, writes are associated with the appropriate location on the storage device during the write syscall path by mapping the written page to the appropriate buffer head. This writeback code path therefore has no need to identify the written file and executes the block I/O regardless of whether the file exists or not. Like Bento-fs, ext4-o is able to drop the writes to the deleted files so both file systems show similar performance.

Applications: Figure 3 shows the results for application

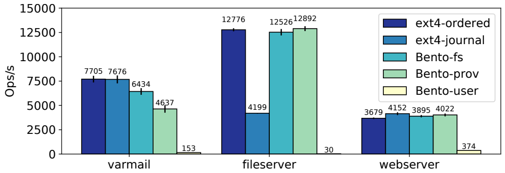


Figure 2: Performance results for ext4 in `data=ordered` mode and `data=journal` mode, Bento-fs, Bento-fs with provenance, and a userspace version of Bento-fs on Filebench application-style workloads in ops/s. Bento-user performs much worse on all benchmarks. Bento-fs and Bento-prov outperform ext4-journal on 'fileserver' due to different handling of un-synced writes to deleted files.

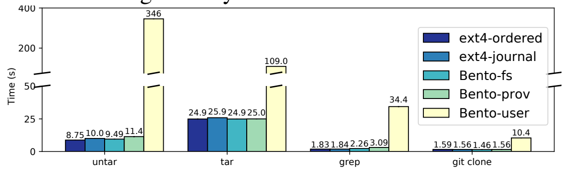


Figure 3: Performance results for ext4 in `data=ordered` mode and `data=journal` mode, Bento-fs, Bento-fs with provenance, and a userspace version of Bento-fs on application workloads 'tar', 'untar', and 'grep' on Linux source code and 'git clone' on xv6. Bento-user performs much worse than the other file systems. Ext4-journal performs somewhat better than Bento-fs and Bento-prov on 'grep'.

workloads. Here, Bento-fs outperforms Bento-user by 4-36x. The difference is particularly noticeable for 'untar' which involves many creates. Creates are particularly impacted by slow block I/O from userspace due to the large number of separate disk operations needed to modify the directory, allocate an inode, and fill the allocated inode. Relative to ext4-j, Bento-fs performs similarly on 'untar', 'tar', and 'git clone' and 19% worse on 'grep'. The slowdown is due to optimized page caching in ext4 that is not implemented in Bento-fs. Relative to ext4-o, Bento-fs performs 13% worse on 'untar' due to data journaling and the lack of delayed allocation. On other benchmarks, ext4-o shows similar results to ext4-j.

For most tested workloads, Bento-prov has similar performance to Bento-fs. Bento-fs outperforms Bento-prov on 'varmail' by 39%, 'untar' by 13%, 'grep' by 68% because Bento-prov logs information on creates, deletes, opens, and closes. Similarly, Bento-prov is 25% slower on the multithreaded create microbenchmark.

Key-Value Stores: Figure 4 shows the results for Redis ('set' and 'get') and RocksDB ('fillrandom' and 'readrandom') workloads on the four file systems. Due to caching, Bento-user performs similarly to the others on read-intensive workloads, but it performs much worse on writes. Bento-fs and Bento-prov show similar performance to ext4-j and ext4-o on reads but slightly outperform them on writes.

5.4 Live Upgrade

In this section, we measure the effect of a live upgrade on application file system performance during an upgrade of the file system from Bento-fs to Bento-prov. We do not use Filebench for these benchmarks so we can collect latency of individual operations. We ran two tests, both using a directory that initially contained 400,000 files. In the first, we executed a single thread that repeatedly created and deleted files. In the second, we executed 10 threads that repeatedly wrote and synced 64Kb writes to random files; we used 10 threads because with too many threads any service interruption caused by the upgrade was hidden by the latency variability of individual operations. In both tests, we upgraded to the version with provenance tracking after 0.5 seconds and completed the test after another 0.5 seconds. We converted the latency measurements into throughput by calculating the number of operations that occur each 5ms interval to smooth the data slightly. The results are shown in Figure 5a and Figure 5b.

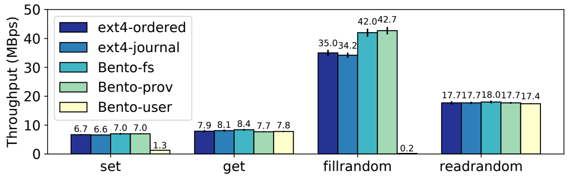


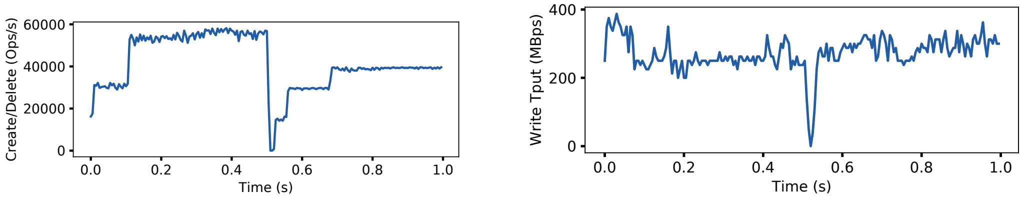
Figure 4: Performance results for ext4 in `data=ordered` mode and `data=journal` mode, Bento-fs, Bento-fs with provenance , and a userspace version of Bento-fs on Redis 'set' and 'get' and RocksDB 'fillrandom' and 'readrandom'. Bento-user performs much worse on write benchmarks.

These graphs show a performance drop where the upgrade occurred at 0.5 seconds. In both tests, the upgrade took around 15ms, during which time the file system was unavailable and a single operation per thread was blocked in the kernel. The performance recovered after the upgrade completed but create and delete performance was lower because the provenance-tracking file system performs extra work on these operations.

6 RELATED WORK

Using safe languages for kernel development. Several systems, including Pilot [40], SPIN [6], Singularity [20], Biscuit [13], Redox [42], and Tock [23] write the entire operating system, including the kernel, in a high-level language. SPIN leverages type safety to allow application-specific customization of kernel behavior. We are also not the first to integrate Rust into the Linux kernel [24, 27]. The Berkeley Packet Filter (eBPF) [32] is a type safe language for safe extensibility in Linux. Users can insert eBPF programs at predefined kernel locations, and the kernel verifies the safety of the inserted programs before running them. ExtFUSE [8] has enabled writing parts of a stackable file system using eBPF. Compared to these, Bento shows that it is possible to develop feature-rich file systems in a safe language to allow continuous integration of new features into a commodity operating system.

Software fault isolation and verification. An alternative approach is to allow development in an unsafe language (e.g., C) but do additional compile-time and runtime checks to prevent memory errors from affecting the rest of the system. Software fault isolation (SFI) [9, 30, 49] is a technique for sandboxing the impact of faults in C modules to the module itself; SFI has been widely used for protecting kernel device



(a) Create+delete operations in ops/s.

(b) Synced writes with 10-threads in MBps.

Figure 5: Performance during an upgrade from Bento-fs to Bento-prov, a provenance-tracking version of Bento-fs. At 0.5 seconds, Bento-fs is upgraded to Bento-prov. The system experiences around 15ms of downtime.

drivers. We chose to use Rust instead as it has lower runtime overhead and provides the additional benefit of bug prevention in addition to sandboxing errors. Software verification is a powerful tool for producing bug-free kernel code, and it has been shown that a simple, single-threaded file system can be verified [45]. Extending that work to handle concurrency and high performance file systems is still ongoing.

Moving kernel features to userspace. Microkernel design, where kernel services run in userspace, is another way to speed operating system development [1, 25] especially when safety and/or development velocity are more important than raw performance. Filesystem in Userspace (FUSE) is a good example in the Linux file system context. Many file systems have been developed in FUSE; when people need performance, they often re-implement the system inside the kernel [10, 19] using VFS. With Bento, developers no longer need to choose between performance and development velocity.

A related approach is to run the userspace OS service on dedicated processor cores, where applications communicate with the service via asynchronous message queues in shared memory [4, 7, 22, 31]. To date, this approach has only been proposed and not implemented for file systems [28]. Performance can often be competitive with an equivalent kernel implementation, except when processors need to busy wait or when the system needs page remapping for efficient zero copy I/O.

Rump kernels (or anykernels) enable running unmodified kernel code as userspace libraries by hijacking system calls and providing userspace implementations of necessary kernel internals. They are used for untrusted execution of kernel code, e.g., when mounting an untrusted file system, or userspace debugging. Implementations exist for NetBSD as a rump kernel [21] and Linux as the libOS [46] and Linux Kernel Library [39] projects; similarly, User Mode Linux [15] enables running a Linux kernel as a userspace process.

OS live upgrade. There are three main commercially available tools for live upgrade of Linux systems: ksplice [3, 36], kpatch [38], or kGraft [37]. All three perform live upgrade of Linux kernel diffs and focus on security patches that do not modify data structure layout. The internals of each approach differ, but all three reroute calls from modified functions to new functions. Some research systems provide support for upgrade of more complex components. Most similar to Bento's design is K42 [5], a research operating system that enables upgrade of modular components by quiescing the component then transferring state to the new instance and updating references. PROTEOS [18], another research operating system, also supports live upgrade of modular components. DynAMOS [29] and LUCOS [11] enable live upgrade of complex components in Linux without the need for state quiescence by using shadow data structures and virtualization, respectively, to maintain state.

Stackable file systems. Stackable designs construct complex file systems by stacking layers of functionality on top of simple base file systems, enabling high velocity development. File system stacking is natively supported by VFS and is used by the overlay file system and eCryptfs, but these file systems still suffer from the velocity problems caused by kernel C code. FiST [50] proposed a framework for development of portable stackable file systems written in a new high-level language, augmented with C code. This improves velocity by reducing the complexity of code written by developer, but cannot support complex file system data structures and cannot provide safety guarantees about the C code.

7 CONCLUSION

Bento is a framework for high velocity development of Linux kernel file systems that enables several goals: safety, performance, generality, compatibility with existing operating systems, ability to do live upgrade, and support for easy debugging. Bento provides these properties for file systems written in Rust, by translating Linux interfaces into safe interfaces with restricted memory sharing, supporting live upgrade with state transfer, and exposing identical interfaces to kernel and userspace file systems for userspace debugging. We implement Bento-fs, a simple file system using Bento and show that it has similar performance to ext4 and significantly outperforms the version of Bento-fs compiled to run in userspace. We develop a provenance tracking version of Bento-fs, and show that we can transparently upgrade Bento-fs to it with only 15 ms of service interruption to running applications.

Acknowledgements. We would like to thank Remzi Arpaci-Dusseau for his helpful feedback on earlier drafts of this paper. We would also like to thank our anonymous reviewers and our shepherd, Rob Ross, for their helpful comments and feedback. This work is partially supported by the National Science Foundation grant CNS-1856636 AM04. This work was also supported by Google and Huawei.

REFERENCES

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. In *Summer USENIX*, 1986.

[2] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *SOSP*, 2019.

[3] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. In *EuroSys*, 2009.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

[5] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing Dynamic Update in an Operating System. ATEC, 2005.

[6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *SOSP*, 1995.

[7] Brian Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level Interprocess Communication for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2), May 1991.

[8] Ashish Bijlani and Umakishore Ramachandran. Extension Framework for File Systems in User Space. In *USENIX ATC*, 2019.

[9] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast Byte-granularity Software Fault Isolation. In *SOSP*, 2009.

[10] Ceph. Ceph kernel module. https://github.com/ceph/ceph-client.

[11] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live Updating Operating Systems Using Virtualization. In *VEE*, 2006.

[12] Russ Coxx, Frans Kaashoek, and Robert Morris. Xv6, a simple Unix-like teaching operating system, 2020.

[13] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *OSDI*, 2018.

[14] Jeff Dean and Sanjay Ghemawat. LevelDB: A Fast Persistent Key-Value Store, 2011.

[15] J. Dike. A user-mode port of the Linux kernel. In *Annual Linux Showcase & Conference*, 2000.

[16] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *SOSP*, 1995.

[17] Filesystem in Userspace. https://github.com/libfuse/libfuse.

[18] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and Automatic Live Update for Operating Systems. In *ASPLOS*, 2013.

[19] GlusterFS. Glusterfs kernel module. https://staged-gluster-docs.readthedocs.io/en/release3.7.0beta1/Features/libgfapi/.

[20] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS OSR*, 2007.

[21] Antti Kantee. Rump File Systems: Kernel Code Reborn. In *USENIX Annual Technical Conference*, 2009.

[22] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas E. Anderson. TAS: TCP Acceleration as an OS Service. In *EuroSys*, 2019.

[23] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *SOSP*, 2017.

[24] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. Securing the Device Drivers of Your Embedded Systems: Framework and Prototype. In *ARES*, 2019.

[25] Jochen Liedtke. On Microkernel Construction. In *SOSP*, 1995.

[26] Lineage File System. https://crypto.stanford.edu/~cao/lineage.

[27] Linux-kernel-module-rust. https://github.com/fishinabarrel/linux-kernel-module-rust.

[28] Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Sudarsun Kannan. File Systems as Processes. In *HotStorage*, 2019.

[29] Kristis Makris and Kyung Dong Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *EuroSys*, 2007.

[30] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-principal Modules. In *SOSP*, 2011.

[31] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A Microkernel Approach to Host Networking. In *SOSP*, 2019.

[32] Steven McCanne and Jacobson Van. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Winter USENIX*, 1993.

[33] Jeffrey C. Mogul and John Wilkes. Nines are Not Enough: Meaningful Metrics for Clouds. In *HotOS*, 2019.

[34] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *OSDI*, 2018.

[35] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-Aware Storage Systems. In *ATEC*, 2006.

[36] Oracle Ksplice. https://ksplice.oracle.com/.

[37] Vojtech Pavlik. kGraft: Live Kernel Patching. https://www.suse.com/c/kgraft-live-kernel-patching/.

[38] Josh Poimboeuf. Introducing kpatch: Dynamic Kernel Patching. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching.

[39] O. Purdila, L. A. Grijincu, and N. Tapus. LKL: The Linux kernel library. In *9th RoEduNet IEEE International Conference*, pages 328–333, 2010.

[40] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An Operating System for a Personal Computer. *Commun. ACM*, 23(2):81–92, February 1980.

[41] Redis. https://redis.io.

[42] Redox. https://www.redox-os.org/.

[43] RocksDB. https://rocksdb.org/.

[44] Marc Rozier, Vadim Abrossimov, François Armand, I. Boule, Michel Gien, Marc Guillemont, F. Herrmann, Claude Kaiser, S. Langlois, Pierre Leonard, and W. Neuhauser. CHORUS Distributed Operating Systems. *Computing Systems*, 1988.

[45] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button Verification of File Systems via Crash Refinement. In *OSDI*, 2016.

[46] H. Tazaki, Ryo Nakamura, and Y. Sekiya. Operating System with Mainline Linux Network Stack. 2015.

[47] Amin Vahdat and Thomas E. Anderson. Transparent result caching. In *1998 USENIX Annual Technical Conference, New Orleans, Louisiana, USA, June 15-19, 1998*. USENIX Association, 1998.

[48] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *FAST*, USA, 2017. USENIX Association.

[49] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *SOSP*, 1993.

[50] Erez Zadok and Jason Nieh. FiST: A Language for Stackable File Systems. *ACM SIGOPS Operating Systems Review*, 34, 03 2002.

API Function	Description
bento_init(&mut self, req, devname, fc_info)	Initialize the file system.
bento_destroy(&mut self, req)	Destroy the file system.
bento_lookup(&self, req, parent, name, reply)	Lookup a file
bento_forget(&self, req, ino, nlookup)	Forget lookups of a file
bento_getattr(&self, req, ino, reply)	Get attributes
bento_setattr(&self, req, args..., reply)	Set attributes
bento_readlink(&self, req, ino, reply)	Read a symbolic link
bento_mknod(&self, req, parent, name, mode, rdev, reply)	Create a file node
bento_mkdir(&self, req, parent, name, mode, reply)	Create a directory
bento_unlink(&self, req, parent, name, reply)	Unlink a file
bento_rmdir(&self, req, parent, name, reply)	Remove a directory
bento_symlink(&self, req, parent, name, link, reply)	Create a symbolic link
bento_rename(&self, req, parent, name, newparent, newname, flags)	Rename a file
bento_link(&self, req, ino, newparent, newname, reply)	Create a hard link
bento_open(&self, req, ino, flags, reply)	Open a file
bento_read(&self, req, ino, fh, offset, size, reply)	Read data from a file
bento_write(&self, req, ino, fh, offset, data, flags, reply)	Write data to a file
bento_flush(&self, req, ino, fh, lock_owner, reply)	Called on each close of a file
bento_release(&self, req, ino, fh, flags, lock_owner, flush, reply)	Called on the last close of an open file
bento_fsync(&self, req, ino, fh, datasync, reply)	Sync a file
bento_opendir(&self, req, ino, flags, reply)	Open a directory
bento_readdir(&self, req, ino, fh, offset, reply)	Read a directory
bento_releasedir(&self, req, ino, fh, flags, reply)	Called on the last close of a directory
bento_fsyncdir(&self, req, ino, fh, datasync, reply)	Sync a directory
bento_statfs(&self, req, ino, reply)	Get file system statistics
bento_setxattr(&self, req, ino, name, value, flags, position, reply)	Set extended attributes of a file
bento_getxattr(&self, req, ino, name, size, reply)	Get extended attributes of a file
bento_listxattr(&self, req, ino, size, reply)	List extended attributes of a file
bento_removexattr(&self, req, ino, name, reply)	Remove an extended attribute of a file
bento_access(&self, req, ino, mask, reply)	Check file permissions
bento_create(&self, req, parent, name, mode, flags, reply)	Create and open a file
bento_getlk(&self, req, ino, fh, lock_owner, start, end, typ, pid, reply)	Test for a file lock
bento_setlk(&self, req, ino, fh, lock_owner, start, end, typ, pid, sleep, reply)	Acquire a file lock
bento_bmap(&self, req, ino, blocksize, idx, reply)	Map a block index within a file
bento_update_prepare(&mut self) -> Option<TransferOut>	Prepare to be removed during a live upgrade
bento_update_transfer(&mut, Option<TransferIn>)	Initialize during a live upgrade

Table 5: The full File Operations API, based on the FUSE lowlevel API with *bento_update_prepare* and *bento_update_transfer* added for live upgrade. File systems implement a subset of the provided functions. The *req* includes the requesting application's user id, group id, and process id. The *reply* data structures are used to return data or error values.

Scalable Persistent Memory File System with Kernel-Userspace Collaboration

Youmin Chen, Youyou Lu, Bohong Zhu,
Andrea C. Arpaci-Dusseau[†], Remzi H. Arpaci-Dusseau[†], Jiwu Shu[*]
Tsinghua University [†] *University of Wisconsin – Madison*

Abstract

We introduce KUCO, a novel direct-access file system architecture whose main goal is scalability. KUCO utilizes three key techniques – collaborative indexing, two-level locking, and versioned reads – to offload time-consuming tasks, such as pathname resolution and concurrency control, from the kernel to userspace, thus avoiding kernel processing bottlenecks. Upon KUCO, we present the design and implementation of KUCOFS, and then experimentally show that KUCOFS has excellent performance in a wide range of experiments; importantly, KUCOFS scales better than existing file systems by up to an order of magnitude for metadata operations, and fully exploits device bandwidth for data operations.

1 Introduction

Emerging byte-addressable persistent memories (PMs), such as PCM [22, 34, 51], ReRAM [3], and the recently released Intel Optane DCPMM [27], provide performance close to DRAM and data persistence similar to disks. Such high-performance hardware increases the importance of redesigning *efficient* file systems. In the past decade, the systems community has proposed a number of file systems, such as BPFS [11], PMFS [14], and NOVA [43], to minimize the software overhead caused by a traditional file system architecture. However, these PM-aware file systems are part of the operating system and applications need to trap into the kernel to access them, where system calls (syscalls) and the virtual file system (VFS) still incur non-negligible overhead. In this regard, recent work [13, 21, 28, 39] proposes to deploy file systems in userspace to access file data directly (i.e., direct access), thus exploiting the high performance of PM.

Despite these efforts, we find that another important performance metric – *scalability* – still has not been well addressed, especially when multicore processors meet fast PMs. NOVA [43] improves multicore scalability by partitioning internal data structures and avoiding using global locks. However, our evaluation shows that it still fails to scale well due to the existence of the VFS layer. Even worse, some userspace file system designs further exasperate the scalability problem by introducing a centralized component. For example, Aerie [39] ensures the integrity of file system metadata by sending expensive inter-process communications (IPCs) to a trusted process (TFS) that has the authority to update metadata. Strata [21], as another example, avoids the involvement of a centralized process in normal operations by directly recording updates in PM logs, but requires a KernFS to apply them (including both data and metadata) to the file system, which causes one more time of data copying. The trusted process (e.g., TFS or KernFS) in both file systems is also responsible for concurrency control, which inevitably becomes the bottleneck under high concurrency.

In this paper, we revisit the file system design by introducing a *kernel-userspace collaboration* architecture, or KUCO, to achieve both direct access performance and high scalability. KUCO follows a classic *client/server model* with two components, including a userspace library (named Ulib) to provide basic file system interfaces, and a trusted thread (named Kfs) placed in the kernel to process requests sent by Ulib and perform critical updates (e.g., metadata).

Inspired by distributed file system designs, e.g., AFS [17], that improve scalability by minimizing server loads and reducing client/server interactions, KUCO presents a novel task division and collaboration between Ulib and Kfs, which offloads most tasks to Ulib to avoid a possible Kfs bottleneck. For metadata scalability, we introduce a *collaborative indexing* technique to allow Ulib to perform pathname resolution before sending requests to Kfs. In this way, Kfs can update metadata items directly with the pre-located addresses provided by Ulib. For data scalability, we first propose a *two-level locking* mechanism to coordinate concurrent writes to shared files. Specifically, Kfs manages a write lease for each file and assigns it to the process that intends to open the file. Instead, threads within this process lock the file with a range-lock completely in userspace. Second, we introduce a *versioned read protocol* to achieve direct reads even without interacting with Kfs, despite the presence of concurrent writers.

KUCO also includes techniques to enforce data protection and improve baseline performance. KUCO maps the PM space into userspace in readonly mode to prevent buggy programs from corrupting file data. Userspace direct writes are achieved with a *three-phase write protocol*. Before Ulib writes a file, Kfs switches the related PM pages from readonly to writeable by toggling the permission bits in the page table. A *pre-allocation* technique is also used to reduce the number of interactions between Ulib and Kfs when writing a file.

With the KUCO architecture, we build a PM file system named KUCOFS, which gains userspace direct-access performance and delivers high scalability simultaneously. We evaluate KUCOFS with file system benchmarks and real-world

[*] Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

applications. The evaluation results show that KucoFS scales better than existing file systems by an order of magnitude under high contention workloads (e.g., creating files in the same directory or writing data in a shared file), and delivers slightly higher throughput under low contention. It also hits the bandwidth ceiling of PM devices for normal data operations. In summary, we make the following contributions:

- We conduct an in-depth analysis of state-of-the-art PM-aware file systems and summarize their limitations on solving the software overhead and scalability problems.
- We introduce Kuco, a *userspace-kernel collaboration* architecture with three key techniques, including *collaborative indexing*, *two-level locking*, and *versioned read* to achieve high scalability.
- We implement a PM file system named KucoFS based on the Kuco architecture, and experimentally show that KucoFS achieves up to one order of magnitude higher scalability for metadata operations, and fully exploits the PM bandwidth for data operations.

2 Motivation

In the past decade, researchers have developed a number of PM file systems, such as BPFS [11], SCMFS [41], PMFS [14], HiNFS [29], NOVA [43], Aerie [39], Strata [21], SplitFS [28], and ZoFS [13]. They are broadly categorized into three types. First, *kernel-level file systems*. Applications access them by trapping into the kernel for both data and metadata operations. Second, *userspace file systems* (e.g., Aerie [39], Strata [21], and ZoFS [13]). Among them, Aerie [39] relies on a trusted process (TFS) to manage metadata and ensure the integrity of it. The TFS also coordinates concurrent reads and writes to shared files with a distributed lock service. Strata [21], in contrast, enables applications to append their updates directly to a per-process log, but requires background threads (KernFS) to asynchronously digest logged data to storage devices. ZoFS avoids using a centralized component and allows userspace applications to update metadata directly with the help of a new hardware feature named Intel Memory Protection Key (MPK). Note that Aerie, Strata, and ZoFS still rely on the kernel to enforce coarse-grained allocation and protection. Third, *hybrid file systems* (e.g., SplitFS [28] and our proposed Kuco). SplitFS [28] presents a coarse-grained split between a user-space library and an existing kernel file system. It handles data operations entirely in userspace, and processes metadata operations through the Ext4 file system. Table 1 provides a summary of existing PM-aware file systems and how well they behave in various aspects.

❶ ***Multicore scalability.*** NOVA [43], a state-of-the-art kernel file system for PMs, is carefully designed to improve scalability by introducing the per-core allocator and per-inode log. Nevertheless, VFS still limits its scalability for certain operations. We experimentally show this by deploying NOVA on Intel Optane DCPMMs (detailed experimental setup is described in § 5.1), and use multiple threads to create, delete, or rename files in the same directory. As shown in Figure 1a, their throughput is almost unchanged as we increase the number of threads, since VFS needs to acquire the lock of the parent directory. Aerie [14] relies on a centralized TFS to handle metadata operations and enforce concurrency control. Although Aerie batches metadata changes to reduce communication with the TFS, our evaluation in §5 shows that the TFS still inevitably becomes the bottleneck under high concurrency. In Strata [21], the KernFS needs to digest logged data and metadata in the background. If an application completely uses up its log, it has to wait for an in-progress digest to complete before it can reclaim log space. As a result, the number of digestion threads limits Strata's overall scalability. Both Aerie and Strata interact with the trusted process (TFS/KernFS) via expensive IPCs, which introduces extra syscall overhead. ZoFS does not require a centralized component, so it achieves much higher scalability. However, ZoFS still fails to scale well when processing operations that require allocating new spaces from the kernel (e.g., `creat` and `append`, see Figures 7d, 7f, and 7g in their paper). Our evaluation shows that SplitFS scales poorly for both data and metadata operations because it 1) does not support sharing between different processes, and 2) relies on Ext4 to update metadata (see Figures 7 and 9).

		NOVA	Aerie/Strata	ZoFS	SplitFS	KucoFS
Category		Kernel	Userspace		Hybrid	
❶ Scalability	Metadata	Medium (§5.2.1)	Low (§5.2.1)	Medium (Fig. 7g in [13])	Low (§5.2.1)	High (§5.2.1)
	Read	Medium (§5.2.2)	Low (§5.2.2)	High	Low (journaling in Ext4)	High (§5.2.2)
	Write	Medium (§5.2.3)	Low (§5.2.3)	Medium (Fig.7f in [13])		High (§5.2.3)
❷	Softeware overhead	High	Low	Medium (`sigsetjump`)	Medium (metadata)	Low
❸ Other issues	Avoid stray writes	✓	✗	✓	✗	✓
	Read protection	POSIX	Partition	Coffer	POSIX	Partition
	Visibility of updates	Immediately	After batch/ After digest	Immediately	`append`: After sync	Immediately
	Hardware required	None	None	MPK	None	None

Table 1: Comparison of different NVM-aware file systems.

❷ ***Software overhead.*** Placing a file system in the kernel faces two types of software overhead, i.e., the syscall and VFS overhead. We investigate such overhead by still analyzing NOVA, where we collect the latency breakdown of common file system operations. Each operation is performed on 1 million files or directories with a single thread. We make two observations from Figure 1b. First, syscalls take up to 21% of the total execution time (e.g., `stat` and `open`). Also, after a process traps into the kernel, the OS may schedule other tasks before returning control to the original one. Hence, syscalls bring extra uncertainty for latency-sensitive applications [12, 33]. Second, Linux kernel file systems are implemented by overriding VFS functions, and VFS causes non-negligible overhead. Although recent PM file

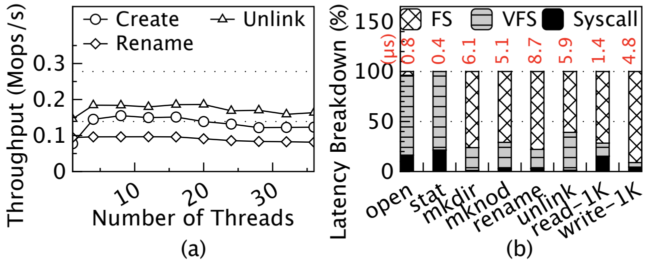


Figure 1: Software overhead and scalability of NOVA.

systems [9, 11, 14, 29, 40, 43, 50] use direct access (DAX) to bypass the page cache in VFS, we find that an average of 34% of the time is still spent in the VFS layer for NOVA. ZoFS [13] deploys a file system in userspace to avoid trapping into the kernel; however, it still incurs extra software overhead. ZoFS allows userspace applications to update metadata directly, which may cause a normal program to be terminated when accessing metadata that is corrupted by malicious attackers. To achieve graceful error return, ZoFS invokes a `sigsetjump` instruction at the beginning of each syscall, which causes extra delays (~200 ns). SplitFS requires a kernel file system to handle metadata operations, so it still introduces kernel overhead.

❸ ***Other issues.*** First, misused pointers can lead to writes to incorrect locations and corrupt the data, which is known as *stray writes* [14]. Strata [21] exposes the per-process operation log and the DRAM cache (including both metadata and data) to userspace applications. Aerie [39] and SplitFS [28] map a subset of the file system image to userspace. Hence, stray writes can easily corrupt the data in these areas, and such corruptions are permanent in NVM even after reboots. Second, Aerie, Strata, and SplitFS improve performance by delaying the visibility of the newly written data to other processes until issuing a `fsync`, forcing applications to make corresponding adjustments. Third, ZoFS heavily relies on the MPK mechanism, if an application also needs to use MPK, they may compete for the limited MPK resources.

To summarize, it is hard to achieve high scalability and low software overhead with existing file system designs, and this motivates us to introduce the Kuco architecture.

3 The Kuco Architecture

In this paper, we introduce the Kuco architecture to show that a *client/server model* can be adopted to realize the two goals simultaneously. The central idea underlying Kuco is a fine-grained task division and collaboration between the *client* and *server*, where most loads are offloaded to the client part to avoid the server from becoming the bottleneck.

3.1 Overview

Figure 2 shows the Kuco architecture. It follows a *client/server model* with two parts, including a userspace library and a global kernel thread, which are called Ulib and Kfs, respectively. An application accesses Kuco by linking with

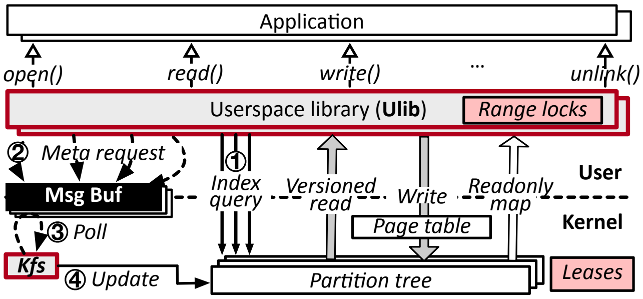


Figure 2: The Kuco architecture. metadata updates (①-④): Ulib interacts with Kfs via *collaborative indexing*; read: direct access via *versioned read*; write: direct access based on a *three-phase write protocol* and *two-level locking* for concurrency control.

Ulib first, and different Ulib instances (i.e., applications) interact with Kfs via separate memory message buffers. Like existing userspace file systems [21, 39], Kuco maps the PM space to userspace to support direct read and write accesses. To protect file system metadata from being corrupted, Kuco does not allow applications to update metadata directly; instead, such requests are posted to Kfs, and Kfs then updates metadata on behalf of them.

Kuco delivers high scalability with a fine-grained task division and collaboration between Ulib and Kfs. For metadata scalability, Kuco incorporates the *collaborative indexing* mechanism to offload the pathname traversal job from Kfs to userspace (§3.2). Instead of sending metadata operations (e.g., `creat` or `unlink`) to Kfs directly, Ulib first finds all the related metadata items in userspace, and then encapsulates such information in the request before sending it out. Therefore, Kfs can perform metadata modifications directly with the given addresses. For data scalability, a *two-level locking* mechanism is used to handle concurrent writes to shared files (§3.3). Specifically, Kfs uses a lease-based distributed lock to resolve write conflicts between different applications (or processes). Concurrent writes from the same process are serialized using a pure userspace range lock, which can be acquired without the involvement of Kfs. Kuco further introduces the *versioned read* technique to perform file reading in userspace (§3.5). By adding extra version bits in data block mappings (which map logical file data to physical PM addresses), Kuco can read a consistent version of data blocks without interacting with Kfs to acquire the lock, despite that there are other concurrent writers.

To further prevent buggy programs from corrupting file data, PM space is mapped to userspace in readonly mode. Kuco enables userspace direct writes on readonly addresses by placing Kfs in the kernel with a *three-phase write protocol* (§3.4). Before Ulib writes a file, Kfs modifies the permission bits in the page table first to switch the involved data pages from readonly to writable. To further reduce the number of interactions between Ulib and Kfs when writing a file, Kuco adopts *pre-allocation*, where Ulib can allocate more free pages from Kfs than desired. Except for the write protection mechanism that prevents stray writes, the PM space in KUCO is then divided into different partition trees, which act as the minimum unit for read protection. By applying KUCO in a file system named KUCOFS and putting all techniques together, KUCOFS gains direct-access performance, delivers high scalability, and ensures the kernel-level data protection.

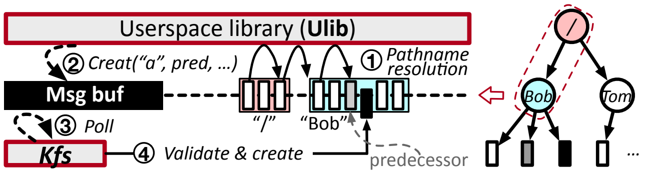


Figure 3: Creating a file (①-④) with *collaborative indexing*.

3.2 Collaborative Indexing

In a typical *client-server model*, whenever Kfs receives a metadata request, it needs to find the related metadata (e.g., *inodes* that describe file attributes, or *dentries* that map file names to inode numbers) by performing iterative pathname resolution from the root inode to the directory containing this file. Such pathname traversal overhead is a heavy burden for Kfs, especially when a directory contains a large number of sub-files or with deep directory hierarchies.

To address this issue, we propose to offload the pathname resolution task from Kfs to Ulib. By mapping partition trees to userspace, Ulib can find the related metadata items directly in userspace, and then sends a metadata update request to Kfs by encapsulating the metadata addresses in the request as well. In this way, Kfs can update metadata directly with the given addresses, and the pathname resolution overhead is offloaded from Kfs to userspace.

Figure 3 shows how KUCO creates a file with a pathname of "/Bob/a". Ulib first finds the predecessor dentry of file "a" in the dentry list of "Bob" (①). It then sends a creat request to Kfs, and the address of the predecessor is put in the message too (②). Kfs then creates the file after receiving the request (③④), which includes creating an inode of this file, and then inserting a new dentry in the parent directory's dentry list with the given predecessor. To delete a file, both the inode of this file and dentry in the parent directory should be deleted, so both of their addresses are kept in the unlink request before Ulib sends it. Note that atime is disabled by default, enabling readonly operations (e.g., stat, readdir) to be performed in userspace without posting extra requests to Kfs.

In KUCO, Ulibs produce pointers and Kfs consumes them. This "one-way" pointer sharing paradigm simplifies ensuring the correctness and safety of KUCO. On the one hand, metadata items are placed in a metadata area with separate address space and Ulib can only pass the addresses of two types of metadata items (i.e., dentry and inode). Hence, we add an identifier field at the beginning of each metadata item, which helps Kfs to check the metadata type – any addresses not in the metadata area or not pointing to a dentry/inode is considered invalid. On the other hand, Kfs also performs consistency checking based on the file system internal logic:

First, Ulib might read an inconsistent directory tree. For example, when Kfs is creating new files in a directory, concurrent Ulibs may read an inconsistent dentry list of this directory. To address this issue, we organize the dentry list of each directory with a skip list [32] and each dentry is indexed by the hash value of the file name. Skip list has multiple layers of linked list-like data structure. Each higher layer acts as an "express lane" for the lower list layer. The list-based structure enables lock-free atomic updates by performing pointer manipulations. Besides, there are only insert and delete operations to the dentry list performed by a single Kfs, including rename operations which are performed by first inserting a new node and then deleting the old one. Therefore, a read to a dentry is always performed to a consistent one even without acquiring the lock.

Second, with such a lock-free design, userspace applications may read metadata items that are being deleted by Kfs, causing the "read-after-delete" anomaly. To safely reclaim the deleted items, we need to ensure that no threads access it anymore. We address this issue by using an epoch-based reclamation mechanism (EBR) [15]. EBR maintains a global epoch and three reclaim queues, where the execution is divided into epochs and reclaim queues are maintained for the last three epochs. Each thread also owns a private epoch. Items deleted in epoch *e* are placed into the queue for epoch *e*. Each time Ulib starts an operation, it reads the global epoch and updates its own epoch to be equal to the global one. It then checks the private epochs of others. If all Ulibs are active in the current epoch *e*, then a new epoch begins. At this time, all threads are active either in *e* or in *e+1*, and items in the queue related to *e-1* can be reclaimed safely. We also add a *dirty* flag in each inode/dentry. Kfs deletes a metadata item by setting its *dirty* flag to an invalid state, preventing applications from reading the already deleted items.

Third, Kfs needs to handle conflicting metadata operations properly. For example, when multiple Ulibs are performing metadata operations concurrently, the pre-located metadata item of one Ulib might be deleted or renamed by another concurrent Ulib before Kfs accesses it. Hence, this item is no longer valid and its address cannot be used by Kfs anymore. It is also possible that a malicious process attacks Kfs by providing arbitrary addresses. Luckily, only the Kfs can update metadata, and it can validate the pre-located metadata before processing the operation. Specifically, Kfs checks if the pre-located item still exists or is still the predecessor, and avoids creating files with the same name. When the validation fails, Kfs then resolves the pathname itself and returns an error code to the Ulib if the operation fails anyway.

Discussion. First, KUCO ensures that all metadata operations are processed atomically. For creat, Kfs atomically inserts a new dentry in the skip list only after an inode has been created, to make the created file visible; For unlink, it

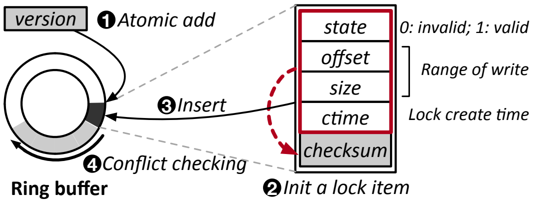


Figure 4: Direct access range-lock. Each opened file owns a range-lock. ❶-❹ show the steps to acquire a lock.

atomically deletes the dentry before deleting other fields. `Rename` involves updating two dentries (create a new entry in the destination path, and then delete the old one), so a program can see two same files on both places at some point in time. We leverage the *dirty* flag in each dentry to prevent such an inconsistent state. Specifically, the old entry on the source path is set to *dirty* before creating the new entry, and is then set to *invalid* after the new entry is created. As a whole, we can observe that metadata operations always change the directory tree atomically, and Ulib is guaranteed to have a consistent view of the directory tree even without acquiring the lock. Second, Kuco's scalability is further improved by avoiding using locks — concurrent metadata updates are all delegated to the global Kfs, so they can be processed without any locking overhead (only Kfs can update metadata) [16, 35]. Kuco ensures the crash consistency of metadata via an operation log, which will be discussed in §4.2.

3.3 Two-Level Locking

Kuco introduces a *two-level locking service* to coordinate concurrent writes to shared files, which prevents Kfs from being frequently involved in concurrency control. First, Kfs assigns *write leases* (in the kernel, see Figure 2) on files to enforce coarse-grained coordination between different processes, as in Aerie and Strata [21, 39]. Only the process that holds a valid write lease (not yet expired) can write the file. We assume that Ulib applies for leases infrequently, and this is based on the fact that it is not the common case for multiple processes to frequently and concurrently write the same file. More fine-grained sharing between processes can be achieved via shared memory or pipes [21]. *Read leases* are not needed in Kuco (see Section 3.5).

Second, we introduce a *direct access range-lock* to serialize concurrent writes between threads within the same process. Once a Ulib acquires the write lease of a file, it creates a range lock for this file in userspace, which is actually a DRAM ring buffer (as shown in Figures 4). A thread writes a file by acquiring the range-lock first, and it is blocked if a lock conflict occurs. Each slot in the ring buffer has five fields, which are state, offset, size, ctime, and a checksum. The checksum is the hash value of the first four fields. We also place a version at the head of each ring buffer to describe the order of each write operation. To acquire the lock of a file, Ulib firstly increments the version with an atomic `fetch_and_add` (i.e., ❶). It then inserts a lock item into a specific slot in the ring buffer (❷ and ❸, the location is determined by the fetched version modulo the ring buffer size). The insertion is blocked when this slot overlaps with the head of the ring buffer. After this, Ulib traverses the ring buffer backward to find the first conflicting lock item (i.e., their written data overlaps). If such a conflict exists, Ulib verifies its checksum, and then polls on its state until it is released. Ulib also checks its ctime field repeatedly to avoid the deadlock if a thread aborts before it releases the lock (❹). With this design, multiple threads can write different data pages in the same file concurrently.

3.4 Three-Phase Write

Once the lock has been required, Ulib can actually write file data. Since PM spaces are mapped to userspace in readonly mode, Ulib cannot write file data directly. Instead, we propose a *three-phase write protocol* to perform direct writes. To ensure the crash consistency, Kuco follows a copy-on-write (CoW) approach to write file data, where the newly written data is always redirected to new PM pages. Similar to NOVA [43] and PMFS [14], we use 4 KB as the default data page size. The write protocol in Kuco consists of three steps. First, Ulib locks the file via *two-level locking* and sends a request to Kfs to allocate new PM pages. Note that, by using a CoW way, space allocation is necessary for both `overwrite` and `append` operations. Kfs also needs to modify the related page table entries to make these allocated PM pages writable before sending the response message back. Second, Ulib copies both the unmodified data from the old place and new data from the user buffer to the allocated PM pages, and persists them via flush instructions. Third, Ulib sends another request to Kfs to update the metadata of this file (i.e., inode, block mapping), switch the newly written pages to readonly, and finally releases the lock.

Furthermore, we introduce the *pre-allocation* mechanism to avoid allocating new PM pages from Kfs for every write operations. Specifically, we allow Ulib to allocate more free pages from Kfs than desired (4 MB at a time in our implementation). In this way, Ulib can use local free PM pages without interacting with Kfs for most write operations. When an application exits, the unused pages are given back to the Kfs. For an abnormal exit, these free pages are temporarily non-reusable by other applications, but still can be reclaimed during the recovery phase (see §4.2). *Pre-allocation* also helps with reducing the overhead of updating page table entries. When the Kfs updates page table entries after each allocation, it needs to flush the related TLB entries explicitly to make the modifications visible. *Pre-allocation* allows allocating multiple data pages at a time, so the TLB entries can be flushed in batch.

3.5 Versioned Read

In the write protocol, both old and new versions of data pages are temporarily kept due to the CoW way, providing us the

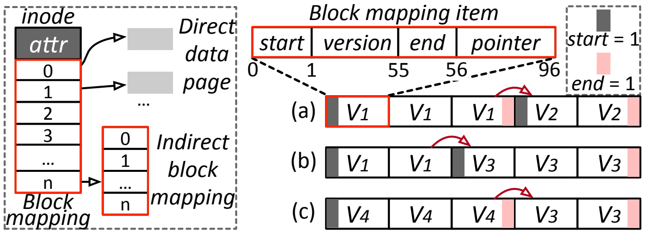


Figure 5: Block mapping format and the versioned read protocol. Mapping items with the same version correspond to the same write operation. The above three consistent cases describe how the *start* and *end* bits can be formatted when the version changes.

opportunity to read file data even without blocking writes. However, block mappings that map a logical file to physical pages are still updated in place by Kfs. This dirves us to design the *versioned read* mechanism to achieve user-level direct reads without any involvement of the Kfs, regardless of concurrent writers.

Versioned Read is designed to allow userspace reads without locking the file, while ensuring that readers never read data from incomplete writes. To achieve this, Kuco uses an Ext2-like [6] block mapping to index data pages and embeds a version field in each pointer of the block mapping. As shown in Figure 5, each 96-bit block mapping item contains four fields, which are start, version, end and pointer. For a `write` operation, say, writing three data pages, Kfs updates the related block mapping items with the following format: $\boxed{1}\boxed{V_1}\boxed{0}\boxed{P_1}$ $\boxed{0}\boxed{V_1}\boxed{0}\boxed{P_2}$ $\boxed{0}\boxed{V_1}\boxed{1}\boxed{P_3}$. In particular, all three items share the same version (i.e., V_1), which is provided by Ulib when it acquires the range lock (in Section 3.3). The start bit of the first item and the end bit of the last item are set to 1. We only reserve 40-bit for the pointer field since it points to a 4 KB-aligned page and the lower 12 bits can be discarded.

With this format, readers can read a consistent snapshot of data pages when one of the three cases is met in Figure 5:

a) No overlapping. When two updates to a file are performed on non-overlapping pages, items with the same version should be enclosed with both a start bit and an end bit (V_1 and V_2 in case a).
b) Overlaps the end part. When a thread overwrites the end part of a former write, a reader should always see a start bit when the version increases ($V_1 \rightarrow V_3$ in case b).
c) Overlaps the front part. When a thread overwrites the first half of a former write, a reader should always see an end bit before the version decreases ($V_4 \rightarrow V_3$ in case c).

If Ulib meets any case other than the above three cases, it indicates that Kfs is updating the block mapping for some other incomplete writes. In this case, Ulib needs to validate again by re-scanning the sequence of the related versions. After Ulib succeeds in the version checking, it then reads the associated data pages. As a whole, Kuco utilizes the embedded versions to detect incomplete writes and retries until reading a consistent snapshot of data.

Read Semantics. In a multi-thread/process execution, versioned read is slightly different from legacy locked read in that it allows concurrent writes. For example, a write starts and has not yet been completed, but in-between, there is a read, which reads an old snapshot of data. In this case, the execution still equals to a serializable order (e.g., "read➛write", "➛" indicates *happens-before*). Versioned read has the same semantic as locked read within each thread, because a read or write has to complete before issuing the next one.

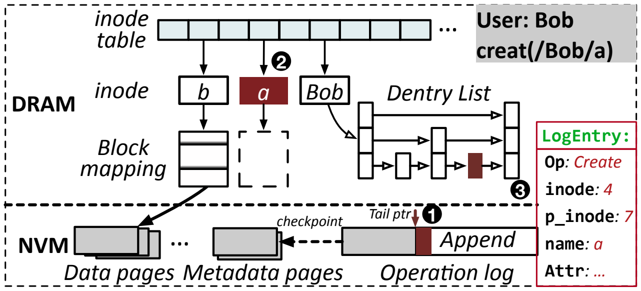


Figure 6: Data layout of a partition tree in KucoFS. `creat` operation with three steps is also shown.

4 KucoFS Implementation

In this section, we describe how the Kuco architecture is applied in a persistent memory file system named KucoFS.

4.1 Data Layout

KucoFS organizes partition trees of Kuco in a hybrid way using both DRAM and PM (Figure 6). In DRAM, an array of pointers (inode table) is placed at a predefined location to point to the actual inodes. The first element in the inode table points to the root inode of the current partition tree. With this, Ulib can find any files from the root inode in userspace. As discussed before, the dentry list of a directory is organized into a skip list, which is also placed in DRAM.

For efficiency, KucoFS only operates on the DRAM metadata for normal requests. To ensure the durability and crash consistency of metadata, KucoFS places an append-only persistent operation log in PM for each partition tree. When the Kfs updates the metadata, it first atomically appends a log entry, and then actually updates the DRAM metadata (see §4.2). When system failures occur, the DRAM metadata can always be recovered by replaying the log entries in the operation log. In addition to the operation log, the extra PM space is cut into 4 KB data pages and metadata pages. Free PM pages are managed with both a bitmap in PM and a free list in DRAM (for fast allocation), and the bitmap is lazily persisted by the Kfs during the checkpoint phase.

4.2 Crash Consistency and Recovery

Metadata consistency. KucoFS ensures the metadata consistency by ordering updates to DRAM and PM. Figure 6 shows the steps of how Kfs creates a file when it receives a `creat` request from Ulib. In ❶, Kfs reserves an unused inode number from the inode table and appends a log entry to the operation log. This log entry records the inode number, file name, parent directory inode number, and other attributes. In ❷, it allocates an inode with each field filled, and updates the inode table to point to this inode. In ❸, it then inserts a dentry into the dentry list with the given address of the predecessor, to make the created file visible. A creation fails if the same dentry already exists (avoid creating the same files). To delete a file, Kfs appends a log entry first, deletes the dentry in the parent directory with the given addresses, and finally frees the related spaces (e.g., inode, data pages and block mapping). If a crash happens before the operation is finished, the DRAM metadata updates will be lost, but Kfs can reconstruct them to the newest state by replaying the log after recovery. For `rename` oprations, except for system failures, the kernel thread may crash and cause the dirty flag to be in an inconsistent state. However, we consider the whole file system crashes if the kernel thread crashes, which requires the file system to be rebooted, and the above logging technique ensures that `rename` operation is also crash-consistent.

Data consistency. KucoFS handles file write operations by first updating data pages in a CoW way, and then appending a log entry in the operation log to record the metadata modifications. At this point, the write is considered *durable*. Then, KucoFS can safely update DRAM metadata to make this operation visible. when a system failure occurs before the log entry is persisted, KucoFS can roll back to its last consistent state since old data and metadata are untouched. Otherwise, this write operation is made visible by replaying the operation log after recovery.

Log cleaning and recovery. We introduce a checkpoint mechanism to avoid the operation log from growing arbitrarily. When the Kfs is not busy, or the size of the log exceeds a threshold (1MB on our implementation), we use a background kernel thread to trigger a checkpoint, which applies metadata modifications in the operation log to PM metadata pages. The bitmap that is used to manage the PM free pages is updated and persisted as well. After that, the operation log is truncated. Background digestion never blocks front-end operations, and the only impact is that log cleaning consumes extra PM bandwidth. However, metadata are typically small-sized and bandwidth consumption is not high.

Each time KucoFS is rebooted from a crash, Kfs first replays the un-checkpointed log entries in the operation log, so as to make PM metadata pages up-to-date. It then copies PM metadata pages to DRAM. The free list of PM data pages is also reconstructed according to the bitmap stored in PM. Crashing again during the recovery is not a concern since the log has not yet been truncated and can be replayed again. Keeping redundant copies of metadata between DRAM and PM introduces higher consumption of PM/DRAM space, but we believe it is worth the efforts. With structured metadata in DRAM, we can perform fast indexing directly in DRAM; appending log entries in the log saves the number of updates to PMs, which reduces the persistence overhead. In the future, we plan to reduce the DRAM footprint by only keeping active metadata in DRAM.

4.3 Write Protection

KucoFS strictly controls updates to the file system image. Both in-memory metadata and the persistent operation log are critical, so the Kfs in the kernel is the only one that is allowed to update them. File pages are mapped to userspace in readonly mode. Applications can only write data to newly allocated PM pages and existing data pages cannot be modified. KucoFS also provides process-level isolation for userspace data structures. The message buffer and range locks are privately owned by each process, so an attacker cannot access them in other processes, except that it performs a privilege escalation attack. Such security issues are out of the scope of this work. As such, we conclude that KucoFS achieves the same write protection as kernel file systems.

Preventing stray writes. Unlike many existing userspace file systems that are vulnerable to stray writes [21, 28, 39], KucoFS prevents this issue by mapping the PM space in readonly mode. Note that there is still a temporary, writable window (less than 1 μs) for the newly-written pages after a write operation is finished but before the permission bits are changed. This is unavoidable, as same as in existing kernel file systems like PMFS. Fortunately, this rarely happens. Besides, range locks and message buffers in userspace might also be corrupted by stray writes. For this threat, we add checksum and lease fields at each slot, which can be used to check whether the inserted element has been corrupted or not.

4.4 Read Protection

KucoFS organizes its directory tree with partition trees, which act as the minimal unit for access control. Each partition tree is self-contained, consisting of metadata and data in PM, and the related metadata copy in DRAM. KucoFS does not allow file/directory structures to span across different partitions. When a program accesses KucoFS, only the partition trees it has access to are mapped to its address space, but other partition trees are invisible to it.

In KucoFS, read access control is strengthened with the following compromises. First, similar to existing userspace file system [13, 39], KucoFS cannot support "write-only" or complex permission semantics such as POSIX access control lists (ACLs), since existing page table only has a single bit to indicate a page is readonly or read-write. Second, KucoFS does not support flexible data sharing between users, because it is hard to change the permission of a specific file (e.g., via `chmod`) with the partition tree design [13,21,31]. Yet there are several practical approaches: ❶ creating a standalone partition that applications with different permissions have access to it; ❷ posting user-level RPCs between different applications to acquire the data. We believe such a tradeoff is not likely to be an obstacle, since KucoFS still supports efficient data sharing between applications within the same user, which is the more common case in real-world scenarios [13].

4.5 Memory-Mapped I/O

Supporting DAX feature in a copy-on-write file system needs extra efforts, since files are out-of-place updated in normal `write` operations [43]. Besides, DAX leaves great challenges for programmers to correctly use PM space with atomicity and crash consistency. Taking these factors into consideration, we borrow the idea from NOVA to provide `atomic-mmap`, which has higher consistency guarantees. When an application maps a file into userspace, Ulib copies file data to its privately managed data pages, and then sends a request to Kfs to map these pages into contiguous address space. When the application issues a `msync` system call, Ulib then handles it as a write operation, so as to atomically make the updates in these data pages visible to other applications.

4.6 KucoFS's APIs

KucoFS provides a POSIX-like interface, so existing applications are able to access it without any modifications to the source code. We achieves this by setting the `LD_PRELOAD` environment variable. Ulib intercepts all APIs in standard C library that are related to file system operations. Ulib processes syscalls directly if the prefix of an accessed file matches with a predefined string (e.g., `/kuco/usr1`). Otherwise, the syscalls is processed in legacy mode. Note that `read` or `write` operations only pass file descriptors in the parameter list. Ulib distinguishes them from legacy syscalls via a mapping table [23], which tracks files of KucoFS.

4.7 Examples: Putting It All Together

Finally, we summarize the design of the Kuco architecture and KucoFS by walking through an example of writing 4 KB of data to a new file and then reading it out.

`Open`. Before sending an `open` request, Ulib pre-locates the related metadata first. Since this is a new file, Ulib cannot find it directly. Instead, it finds the predecessor in its parent directory's dentry list for latter creation. The address, as well as other information (e.g., file name, `O_CREAT` flags, etc.), are encapsulated in the `open` request. When the Kfs receives the request, it creates this file based on the given address. It also needs to assign a write lease to this process. Then, the Kfs sends a response message. After this, Ulib creates a file descriptor and a range lock for this opened file, and returns to the application.

`Write`. The application then uses a `write` call via Ulib to write 4 KB of data to this newly created file. First, Ulib tries to lock the file via the *two-stage locking service*. Since the write lease is still valid, it acquires the lock directly through the range-lock. Ulib blocks the program when there are write conflicts and wait until other concurrent threads have released the lock. After that, Ulib can acquire the lock successfully. It then allocates a 4 KB-page from the pre-allocated pages, copies the data into it, and flushes them out of the CPU cache. Ulib also needs to post an extra request to the Kfs to allocate more free data pages once the pre-allocated space is used up. Finally, Ulib sends the `write` request to the Kfs to finish the rest steps, including changing the permission bits of the written data pages to readonly, appending a log entry to describe this write operation, and updating the DRAM metadata. Ulib finally unlocks the file in the range lock.

`Read`. KucoFS enables reading file data without interacting with the Kfs. To read the first 4 KB from this file, Ulib finds the inode in userspace and reads the first block mapping item. The version checking is performed to ensure its state satisfies one of the three conditions described in Section 3.5. After this, Ulib can safely read the data page pointed by the pointer in the mapping item.

`Close`. Ulib also needs to send a `close` request to the Kfs upon closing this file. Kfs then reclaims the write lease since it will not access this file anymore.

5 Evaluation

In our evaluation, we try to answer the following questions:

- Does KucoFS achieve the goal of delivering direct access performance and high scalability?
- How does each individual technique in KucoFS help with achieving the above goals?
- How does KucoFS perform under macro-benchmark and real-world applications?

5.1 Experimental Setup

Testbed. Our experimental testbed is equipped with 2× Intel Xeon Gold 6240M CPUs (36 physical cores), 384 GB DDR4 DRAM, and 12 Optane DCPMMs (256GB per module, 3 TB in total). We perform all experiments on the Optane DCPMMs residing on NUMA 0 (1.5 TB), whose read bandwidth peaks at 37.6 GB/s and the write bandwidth is 13.2 GB/s. The server is installed with Ubuntu 19.04 and Linux 5.1, the newest kernel version supported by NOVA.

Compared systems. We evaluate KucoFS against NVM-aware file systems including PMFS [14], NOVA [43], SplitFS [28], Aerie [39], and Strata [21], as well as traditional file systems with DAX support including Ext4-DAX [2] and XFS-DAX [38]. Strata only supports a few applications and has trouble running multi-threaded workloads. Similar to previous papers [13,49], we only show part of its performance results in § 5.3 and § 5.4. We only evaluate SplitFS in §5.2 and § 5.4 since it only supports a subset of APIs. For a fair comparison, we deploy SplitFS with *strict* mode, which ensures both durability and atomicity. ZoFS is not open-sourced so we did not evaluate it. Aerie is based on Linux 3.2.2, which cannot support Optane DCPMMs. Hence, we compare with Aerie [39] by emulating persistent memory with DRAM, which injects extra delays. Due to limited space, we only describe Aerie's experimental data verbally without adding extra figures.

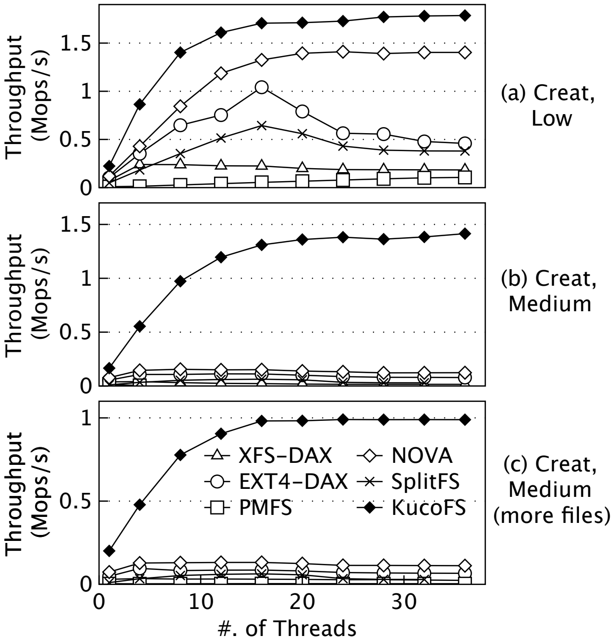


Figure 7: `creat` performance with FxMark. Low: in different folders; medium: in the same folder; more files: each thread creates one million files.

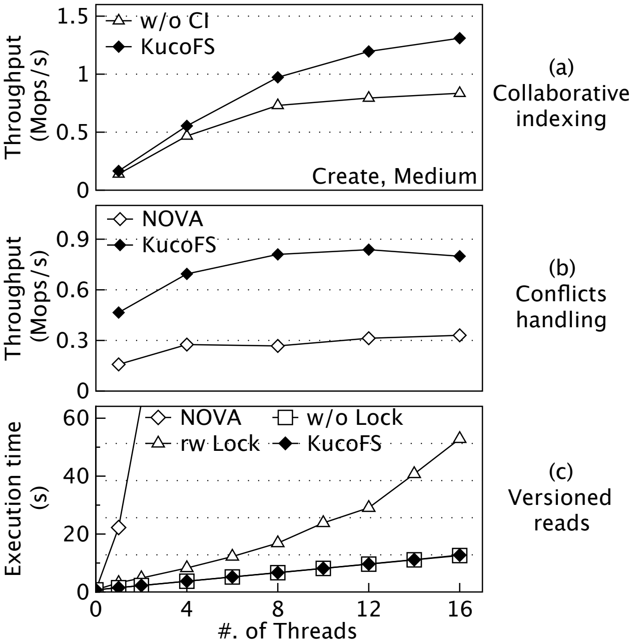


Figure 8: Benefits of *collaborative indexing* and *versioned read*. w/o CI: KucoFS without collaborative indexing.

5.2 Effects of Individual Techniques

We use FxMark [25] to analyze the effects of individual techniques, which explores the scalability of basic file system operations. FxMark provides 19 micro-benchmarks, which is categorized based on four criteria: data types (i.e., data or metadata), modes (i.e., read or write), operations, and sharing levels. We only evaluate the commonly used operations (e.g., read, write, mknod, etc.) due to the limited space.

5.2.1 Effects of Collaborative Indexing

Basic performance. In KucoFS, `creat` operation requires posting requests to Kfs, so we choose this operation to show the effects of *collaborative indexing*. FxMark evaluates `creat` operations by letting each client thread create 10 K files in private directories (i.e., low sharing level) or a shared directory (i.e., medium). As shown in Figures 7a and 7b, KucoFS exhibits the highest performance among the compared file systems and its throughput never collapses, regardless of the sharing level. XFS-DAX, Ext4-DAX and PMFS use a global lock to perform metadata journaling in a shared log, which leads to their poor scalability. NOVA shows excellent scalability under low sharing level by avoiding global locks (e.g., it uses per-inode log and partitions its free spaces). However, all kernel file systems fail to scale under the medium sharing level since VFS needs to lock the parent directory before creating files. SplitFS relies on Ext4 to create files, which accounts for its low scalability. From the ZoFS paper we also find that ZoFS even shows lower throughput than NOVA under low sharing level, since it needs to trap into the kernel frequently to allocate new spaces. Aerie supports synchronizing metadata updates of the created files to TFS with batching (by compromising the visibility), so it achieves comparable performance to that of KucoFS. Aerie fails to work properly when the number of threads increases. The throughput of KucoFS, however, is only decreased slightly with the medium sharing level, which is one order of magnitude higher than other file systems, and 3× higher than that of ZoFS. We explain the high scalability of KucoFS from the following aspects. First, in KucoFS, all metadata updates are delegated to Kfs, so it can update them without any locking overhead. Second, by offloading indexing tasks to userspace, Kfs only needs to do lightweight work.

Larger workload. Furthermore, we measure the scalability of KucoFS in terms of data capacity by extending the workload size. Specifically, we let each thread create 1 million files, 100× larger than the default size in FxMark, and the results are shown in Figure 7c. Compared to the results with a smaller workload size, the throughput of KucoFS drops by 28.5%. This is mainly because a file system needs more time to find a proper slot for insertion in the parent directory when the number of files increases. Even so, KucoFS still outperforms other file systems by an order of magnitude.

Conflict handling. KucoFS requires Kfs to fall back and retry when a conflict occurs, which may impact overall performance. In this regard, we also test how KucoFS behaves when handling operations that conflict with each other. Specifically, we use multiple threads to create the same file concurrently if it does not exist, or delete it instead when it has already been created. We collect the throughput of these successful creations and deletions and the results are shown in Figure 8b. As a comparison, the results of NOVA

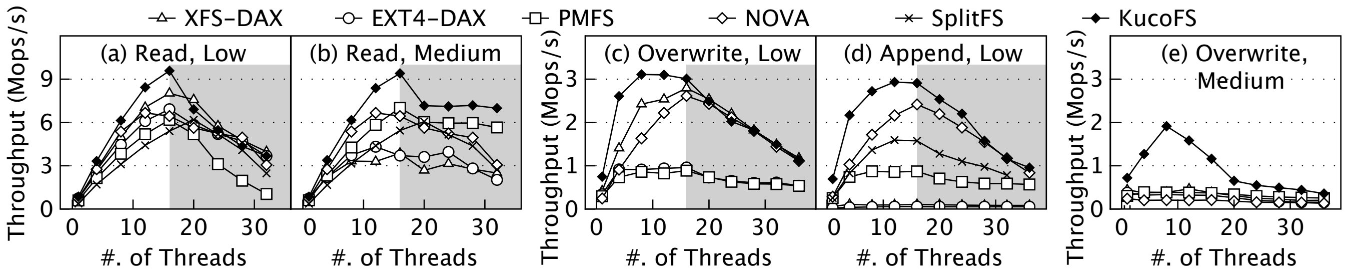


Figure 9: `Read` and `Write` throughput with FxMark. *Low: threads read(write) data from(to) separate files; medium: in the same file but different data blocks; default I/O size: 4 KB; gray area: Optane DCPMMs do not scale on NUMA platform.*

is also shown in the figure. We can observe that KucoFS achieves 2.4× higher throughput than NOVA. In NOVA, a thread needs to acquire the lock before creating or deleting files. Worse, if this creation or deletion fails, other concurrent threads will be blocked unnecessarily since the lock does not protect a valid operation. Instead, in KucoFS, threads can send creation or deletion requests to Kfs without been blocked and Kfs is responsible for determining whether this operation can be processed successfully. Furthermore, since Ulib has already provided related addresses in the request, Kfs can use these addresses to validate metadata items directly, which introduces insignificant overhead.

Breakdown. We also measure the benefit of *collaborative indexing* by comparing with a variant of KucoFS that disables this optimization (i.e., move the metadata indexing tasks back to Kfs, denoted as "*w/o CI*"). Figure 8a shows the results by measuring the throughput of `creat` with a varying number of clients. We make the following observations. First, in the single thread evaluation, *collaborative indexing* does not contribute to improving performance, since moving the metadata indexing task from Ulib back to the Kfs does not reduce the overall latency of each operation. Second, when the number of client threads increases, we find that *collaborative indexing* improves throughput by up to 55%. Since KucoFS only allows the Kfs to update metadata on behalf of multiple Ulib instances, the theoretical throughput limit is $T_{max} = 1/L$ (Ops/s, where L is the latency for Kfs to process one request). Therefore, the offloading mechanism improves performance by shortening the execution time of each request (i.e., L).

5.2.2 Effects of Versioned Read

Figures 9a and 9b show the file read performance of each file system with a varying number of threads under different sharing levels (i.e., low/medium). We make the following observations. First, KucoFS exhibits the highest throughput among the compared file systems, which peaks at 9.4 Mops/s (hardware bandwidth has been fully utilized). The performance improvement stems primarily from the design of *versioned read*, which empowers userspace direct access without the involvement of Kfs. These kernel file systems (e.g., XFS, Ext4, NOVA and PMFS) have to perform context switches and walk through the VFS layer, which impact the read performance. SplitFS only achieves comparable performance to that of NOVA despite its direct-access feature. We find that SplitFS needs to map more PM space to userspace whenever it reads a page that has not been mapped yet, which causes extra overhead. The performance improvement of KucoFS is more obvious for medium sharing level because all the compared systems need to lock the file before actually reading file data. The locking overhead impacts their performance significantly, despite they use shared locks [23]. Second, the read performance of all evaluated file systems drops dramatically when the number of threads keeps increasing (gray area). To get stable results, we first bind threads to NUMA 0 (local access), and the cores at NUMA 1 are used only if the total number of threads is greater than 18. Both we and past work [47] observe that cross-socket accessing to Optane impacts performance greatly. To confirm that our software design is scalable, we deploy NOVA and KucoFS in DRAM, and both of them show scalable read throughput again. Therefore, many recent papers [13, 19] only use the cores from the local NUMA node in their evaluation. With our emulated persistent memory, Aerie shows almost the same performance as that of KucoFS with the low sharing level, but its throughput falls far behind others at a medium sharing level because Aerie needs to interact with the TFS frequently.

We further demonstrate the efficacy of *versioned read* by concurrently reading/writing data from/to the same file. In our evaluation, one read thread is selected to sequentially read a file with an I/O size of 16 KB, and an increasing number of threads are launched to overwrite the same file concurrently (4 KB writes to a random offset). We let the read thread issue read operations for 1 million times and measure its execution time with a varying number of writers. For comparison, we also implement KucoFS *r/w lock* that reads file data by acquiring read-write locks in the range-lock ring buffer, and KucoFS *w/o lock* that reads file data directly without a correctness guarantee. We make the following observations from Figure 8b. First, the proposed *versioned read* achieves almost the same performance as that of KucoFS *w/o lock*. This proves that the overhead of version checking is extremely low. We also observe that KucoFS *r/w lock* needs much more time to finish reading (7% to 3.2× more time than KucoFS for different I/O sizes). This is because it needs to use atomic operations to acquire the range lock, which severely impact read performance when conflicts become frequent.

Second, the execution time of NOVA is orders of magnitudes higher than that of KucoFS. NOVA directly uses `mutexes` to synchronize the reader and concurrent writers. As a result, the reader is always blocked by writers.

5.2.3 Effects of Three-Phase Writes

We evaluate both `append` and `overwrite` operations to analyze the write protocol (see Figures 9 c-d). For overwrite operations with low sharing level, some of them exhibit a performance curve that increases first and then decreases. In the rising part, KucoFS shows the highest throughput among the compared systems because it is enabled to write data in userspace directly. XFS and NOVA also show good scalability. Among them, NOVA partitions free spaces to avoid the locking overhead when allocating new data pages, while XFS directly writes data in-place without allocating new pages. Both PMFS and Ext4 fail to scale since they rely on a centralized transaction manager to write data, introducing extra locking overhead. In the decreasing part, their throughput is mainly affected by two factors: the cross-NUMA overhead, which has been explained before, and the poor scalability of Optane's write performance [19]. SplitFS fails to run properly under this setting. For append operations, XFS-DAX, Ext4-DAX and PMFS exhibit bad scalability as the number of threads increases. This is because they use a global lock to manage the free data pages and metadata journal, so the lock contention contributes to the major overhead. Both NOVA and KucoFS show better scalability, and KucoFS outperforms NOVA by from 10% to 2× with an increasing number of threads. The throughput of SplitFS lies between NOVA and Ext4-DAX. This is because, SplitFS first appends data in a staging file, and then re-links it to the original file by trapping into the kernel. On our emulated persistent memory, Aerie shows the worst performance because the trusted service is the bottleneck, where clients need to frequently interact with the TFS to acquire the lock and allocate spaces.

Two-level locking. To analyze the effects of the lock design, we also evaluate overwrite operations with the medium sharing level, where threads write data to the same file at different offsets. As shown in Figure 9e, the throughput of KucoFS is one order of magnitude higher than the other four file systems when the number of threads is small (SplitFS fails to run properly in this setting). The range-lock design in KucoFS enables parallel updates to different data blocks in the same file. The performance of KucoFS drops again when the number of threads grows to more than 8, which is mainly restricted by the ring buffer size in the range-lock (we reserve 8 lock slots in the ring buffer). We also find that ZoFS shows 2× - 3× higher throughput than that of NOVA (Fig.7f in their paper), but it still underperforms KucoFS.

Memory-mapped I/O. Memory-mapped I/O is the most efficient way to access the file system. Kfs in KucoFS constructs all page tables in advance when processing `mmap` requests. For a fair comparison, we add the MAP_POPULATE flag when using `mmap` to access kernel file systems, which builds the page table during the syscall. The experimental results are as expected (not shown in the figure): when we concurrently issue 4KB read/write requests, all the evaluated file systems saturate the hardware bandwidth.

Workload	**Fileserver**		**Webserver**		**Webproxy**		**Varmail**	
R/W Size	16 KB/16 KB		1 MB/8 KB		1 MB/16 KB		1 MB/16 KB	
R/W Ratio	1:2		10:1		5:1		1:1	
Total number of files in each workload is 100K.								
Threads	1	16	1	16	1	16	1	16
XFS-DAX	39K	127K	121K	1.35M	192K	863K	99K	319K
Ext4-DAX	52K	362K	123K	1.33M	316K	2.50M	57K	135K
PMFS	72K	317K	110K	1.25M	218K	1.54M	169K	1.06M
NOVA	71K	537K	133K	1.43M	337K	3.02M	220K	2.04M
Strata	75K	-	105K	-	420K	-	283K	-
KucoFS	99K	683K	141K	1.48M	463K	3.22M	320K	2.55M
🚀	32%	27%	6%	3%	10%	7%	13%	24%

"🚀" indicates the performance improvement over the 2nd-best system.

Table 2: Filebench throughput with 1 and 16 threads (Ops/s).

5.3 Filebench: Macro-Benchmarks

We then use Filebench [1] as a macro-benchmark to evaluate the performance of KucoFS. Table 2 shows both workload settings (similar to that in the NOVA paper) and experimental results with 1 and 16 threads (adding more threads does not contribute to higher throughput with Filebench [13]). We can observe that, first, KucoFS shows the highest performance among all the evaluated workloads. In single-threaded evaluation with Fileserver workload, its throughput is 2.5×, 1.9×, 1.38×, 1.39× and 1.32× as much as that of XFS, Ext4, PMFS, NOVA, and Strata respectively, and is 3.2×, 5.6×, 1.9×, 1.45× and 1.13× higher with Varmail workload. For read-dominated workloads (e.g., webserver/webproxy), KucoFS also shows slightly higher throughput. The performance improvement mainly comes from the direct access feature of KucoFS. Strata also benefits from direct access and performs the second-best in most workloads. We also observe that the design of KucoFS is a good fit for the Varmail workload. This is expected: Varmail frequently creates and deletes files, so it generates more metadata operations and issues system calls more frequently. As described before, KucoFS eliminates the OS-part overhead and is better at handling metadata operations. Besides, Strata shows much higher throughput than NOVA since the file I/Os in Varmail is small-sized. Strata only needs to append these small-sized updates to the operation log, reducing the write amplification dramatically.

Second, KucoFS is better at handling concurrent workloads. With 16 client threads under the Fileserver workload, KucoFS outperforms XFS-DAX by 4.4×, PMFS by 1.2×, and NOVA by 27%. The performance improvement is more obvious for Varmail workload: it achieves 10× higher performance than XFS-DAX and Ext4-DAX on average. Two reasons contribute to its good performance: first, KucoFS incorporates techniques like *collaborative indexing* to enable Kfs to provide scalable metadata accessing performance; second,

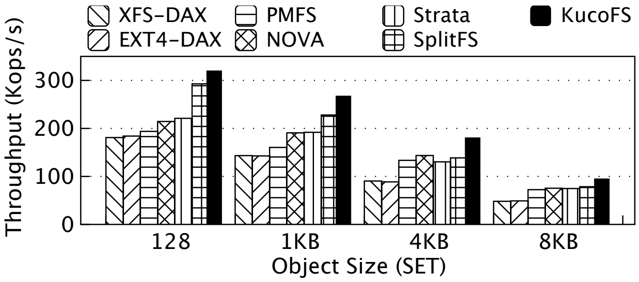


Figure 10: Redis performance with different file systems.

KucoFS avoids using a global lock by letting each client manage private free data pages. NOVA also exhibits good scalability since it uses per-inode log-structure and partitions the free spaces to avoid global locks.

5.4 Redis: Real-World Application

Redis exports a set of APIs allowing applications to process and query structured data, and uses the file system for persistent data storage. Redis has two approaches to persistently record its data: one is to log operations to an append-only-file (AOF), and the other is to use an asynchronous snapshot mechanism. We only evaluate Redis with AOF mode in this paper. Figure 10 shows the throughput of SET operations using 12-byte keys with various value sizes. For small values, the throughput of Redis is 53% higher on average on KucoFS, compared to PMFS, NOVA, and Strata, and 76% higher compared to XFS-DAX and Ext4-DAX. This is consistent with the results of `append` in Section 5.2. With larger object sizes, KucoFS achieves slightly higher throughput than other file systems since most of the time is spent on writing data. Note that Redis is a single-threaded application, so it is reasonable for KucoFS to achieve a throughput of 100 Kops/s with 8KB objects (around 800MB/s). SplitFS is good at handling `append` operations since it processes data-plane operations in userspace. However, it still underperforms KucoFS, because Redis posts `fsync` to flush the AOF file each time it appends new data. Hence, SplitFS needs to trap into the kernel to update metadata, which again causes VFS and syscall overhead.

6 Related Work

Kernel-userspace collaboration. The idea of moving I/O operations from the kernel to userspace has been well studied. Belay et al. [4] abstract the Dune process leveraging the virtualization hardware in modern processors. It enables direct access to the privileged CPU instructions in userspace and executes syscalls with reduced overhead. Based on Dune, IX [5] steps further to improve the performance of data-center applications by separating management and scheduling functions of the kernel (control-plane) from network processing (data plane). Arrakis [31] is a new network server operating system, where applications have direct access to I/O devices and the kernel only enforces coarse-grained protection. FLEX [42] avoids kernel overhead by replacing conventional file operations with similar DAX-based operations, which shares some similarities to SplitFS. While these systems share the same idea of splitting tasks between the kernel and userspace, KucoFS is different in that it exhibits a fine-grained split of responsibilities while enforcing close collaboration.

Persistent memory storage systems. Except for persistent memory file systems mentioned before, we summarize more PM systems here. First, *general PM optimizations.* Yang et al. [46] explore the performance properties and characteristics of Optane DCPMM at the micro and macro levels, and provide a number of guidelines to maximize the performance. Libnvmmio [10] extends userspace memory-mapped I/O with failure atomicity. Many recent papers also designed various data structures that work correctly and efficiently on persistent memory [7, 18, 26, 30, 48, 52]. Second, *PM-aware file systems.* BPFS [11] adopts short-circuit shadow paging to guarantee the metadata and data consistency. SCMFS [41] simplifies the file management by mapping files to contiguous virtual address regions with the virtual memory management (VMM) in existing OS. NOVA-Fortis [44] steps further to be fault-tolerant by providing a snapshot mechanism. Ziggurat [49] is a tiered file system which estimates the temperature of file data and migrates cold data from PM to disks. DevFS [20] pushes the file system implementation into the storage device that has compute capability and device-level RAM. Third, *distributed PM systems.* Hotpot [36] manages PM devices of different nodes in the cluster with a distributed shared persistent memory architecture. Octopus [24, 37] leverages PM and RDMA to build an efficient distributed file system by reducing the software overhead. Similarly, Orion [45] is also distributed persistent memory file system but is built in the kernel. FlatStore [8] is a log-structured key-value storage engine based on RDMA network; it minimizes the flush overhead by batching small-sized requests.

7 Conclusion

In this paper, we introduce a kernel and user-level collaborative architecture named Kuco, which exhibits a fine-grained task division between userspace and the kernel. Based on Kuco, we further design and implement a PM file system named KucoFS and experiments show that KucoFS provides both efficient and highly scalable performance.

Acknowledgements

We sincerely thank our shepherd Donald E. Porter and the anonymous reviewers for their insightful feedback. We also thank Qing Wang and Ramnatthan Alagappan for their excellent suggestions. This material is supported by the National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 62022051, 61832011, 61772300, 61877035), and Huawei (Grant No. YBN2019125112).

References

[1] Filebench file system benchmark. "http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf", 2004.

[2] Support ext4 on NV-DIMMs. "https://lwn.net/Articles/588218", 2014.

[3] IG Baek, MS Lee, S Seo, MJ Lee, DH Seo, D-S Suh, JC Park, SO Park, HS Kim, IK Yoo, et al. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International*, pages 587–590. IEEE, 2004.

[4] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.

[6] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the 1st Dutch International Symposium on Linux*, pages 1–6, 1994.

[7] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[8] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. Hinfs: A persistent memory file system with both buffering and direct-access. *ACM Trans. Storage*, 14(1):4:1–4:30, April 2018.

[10] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing software IO path with failure-atomic memory-mapped interface. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 1–16. USENIX Association, July 2020.

[11] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[12] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[13] Mingkai Dong, Heng Bu, Jiefei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *The 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.

[14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[15] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[16] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[17] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and M. West. Scale and performance in a distributed file system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, page 1–2, New York, NY, USA, 1987. Association for Computing Machinery.

[18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, page 187, 2018.

[19] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent

memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[20] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a true direct-access file system with devfs. In *16th USENIX Conference on File and Storage Technologies*, page 241, 2018.

[21] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. ACM.

[22] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*, pages 2–13, New York, NY, USA, 2009. ACM.

[23] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 90–103, New York, NY, USA, 2019. ACM.

[24] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An rdma-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 773–785, USA, 2017. USENIX Association.

[25] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding manycore scalability of file systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 71–85, Berkeley, CA, USA, 2016. USENIX Association.

[26] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, February 2019. USENIX Association.

[27] Intel Newsroom. Intel$^{@}$ optane^{TM} dc persistent memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html, April 2019.

[28] Kadekodi ohan, Kwon Lee Se, Kashyap Sanidhya, Kim Taesoo, Kolli Aasheesh, and Chidambaram Vijay. Splitfs: A file system that minimizes software overhead in file systems for persistent memory. In *The 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.

[29] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 12:1–12:16, New York, NY, USA, 2016. ACM.

[30] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. ACM.

[31] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[32] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[33] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 145–160, USA, 2018. USENIX Association.

[34] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*, pages 24–33, New York, NY, USA, 2009. ACM.

[35] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 342–358, New York, NY, USA, 2017. ACM.

[36] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 323–337, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. Th-dpms: Design and implementation of an rdma-enabled distributed persistent

memory storage system. *ACM Trans. Storage*, 16(4), October 2020.

[38] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[39] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[40] Ying Wang, Dejun Jiang, and Jin Xiong. Caching or not: Rethinking virtual file system for non-volatile main memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, 2018.

[41] Xiaojian Wu and A. L. Narasimha Reddy. Scmfs: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[42] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 427–439, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[44] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. ACM.

[45] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memories and rdma-capable networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 221–234, USA, 2019. USENIX Association.

[46] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[47] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. *arXiv preprint arXiv:1908.03583*, 2019.

[48] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181, Berkeley, CA, USA, 2015. USENIX Association.

[49] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, 2019.

[50] Deng Zhou, Wen Pan, Tao Xie, and Wei Wang. A file system bypassing volatile main memory: Towards a single-level persistent store. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, CF '18, pages 97–104, New York, NY, USA, 2018. ACM.

[51] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*, pages 14–23, New York, NY, USA, 2009. ACM.

[52] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 461–476, USA, 2018. USENIX Association.

Rethinking File Mapping for Persistent Memory

Ian Neal
University of Michigan

Gefei Zuo
University of Michigan

Eric Shiple
University of Michigan

Tanvir Ahmed Khan
University of Michigan

Youngjin Kwon
School of Computing, KAIST

Simon Peter
University of Texas at Austin

Baris Kasikci
University of Michigan

Abstract

Persistent main memory (PM) dramatically improves IO performance. We find that this results in file systems on PM spending as much as 70% of the IO path performing *file mapping* (mapping file offsets to physical locations on storage media) on real workloads. However, even PM-optimized file systems perform file mapping based on decades-old assumptions. It is now critical to revisit file mapping for PM.

We explore the design space for PM file mapping by building and evaluating several file-mapping designs, including different data structure, caching, as well as meta-data and block allocation approaches, within the context of a PM-optimized file system. Based on our findings, we design HashFS, a hash-based file mapping approach. HashFS uses a single hash operation for all mapping and allocation operations, bypassing the file system cache, instead prefetching mappings via SIMD parallelism and caching translations explicitly. HashFS's resulting low latency provides superior performance compared to alternatives. HashFS increases the throughput of YCSB on LevelDB by up to 45% over page-cached extent trees in the state-of-the-art Strata PM-optimized file system.

1 Introduction

Persistent main memory (PM, also known as non-volatile main memory or NVM) is a new storage technology that bridges the gap between traditionally-slow storage devices (SSD, HDD) and fast, volatile random-access memories (DRAM). Intel Optane DC persistent memory modules [20], along with other PM variants [1,2,4,30,35,50], are anticipated to become commonplace in DIMM slots alongside traditional DRAM. PM offers byte-addressable persistence with only 2–3× higher latency than DRAM, which is highly appealing to file-system designers. Indeed, prior work has re-designed many file-system components specifically for PM, reducing overhead in the IO path (e.g. by eliminating memory copies and bypassing the kernel) [11, 14, 24, 29, 48, 52, 56].

However, not all aspects of the IO path have been examined in detail. Surprisingly, *file mapping* has received little attention in PM-optimized file systems to date. *File mapping*—the translation from logical file offsets to physical locations on the underlying device—comprises up to 70% of the IO path of real workloads in a PM-optimized file system (as we show in §4.8). Even for memory-mapped files, file mapping is still involved in file appends. Yet, existing PM-optimized file systems either simply reuse mapping approaches [11, 14, 24] originally designed for slower block devices [29], or devise new approaches without rigorous analysis [15, 57].

PM presents a number of challenges to consider for file mapping, such as dealing with fragmentation and concurrency problems. Notably, the near-DRAM latency of PM requires reconsidering many aspects of file mapping, such as mapping structure layout, the role of the page cache in maintaining volatile copies of file mapping structures, and the overhead of physical space allocation. For example, some PM-optimized file systems maintain copies of their block mapping structures in a page cache in DRAM, as is traditional for file systems on slower storage devices [11, 24, 29], but others do not [14, 15, 48] and instead rely on CPU caching. It is currently unclear if maintaining a volatile copy of mapping structures in DRAM is beneficial versus relying on CPU caches to cache mapping structures, or designing specialized file map caches.

In this work, we build and rigorously evaluate a range of file mapping approaches within the context of PM-optimized file systems. PM's random access byte addressability makes designing fully random access mapping structures (i.e., hash tables) possible, which expands the design space of possible file mapping structures. To this end, we evaluate and PM-optimize two classic *per-file* mapping approaches (i.e., where each file has its own mapping structure) that use extent trees and radix trees. We propose and evaluate two further, *global* mapping approaches (i.e., where a single structure maps all files in the entire file system). The first uses cuckoo hashing, the second is *HashFS*, a combined global hash table and block allocation approach that uses linear probing. We use these approaches to evaluate a range of design points, from an emphasis on sequential access performance to an emphasis on minimizing memory references, cached and non-cached, as well as the role of physical space allocation.

We evaluate these file-mapping approaches on a series of benchmarks that test file system IO latency and throughput under different workloads, access and storage patterns, IO sizes, and structure sizes. We show how the usage of

PM-optimized file mapping in a PM-optimized file system, Strata [29], leads to a significant reduction in the latency of file IO operations and large gains in application throughput. An evaluation of Strata on YCSB [10] workloads running on LevelDB [17] shows up to 45% improvement in throughput with PM-optimized file mapping (§4). Our analysis indicates that the performance of file-mapping is file system independent, allowing our designs to be applied to other PM file systems.

Prior work investigates the design of PM-optimized *storage structures* [31, 36, 54, 59] and indexing structures for PM-optimized key-value stores [28, 51, 58]. These structures operate on top of memory mapped files and rely on file mapping to abstract from physical memory concerns. Further, PM-optimized storage and indexing structures are designed with per-structure consistency (e.g., shadow paging) [31, 54, 59]. File systems already provide consistency across several metadata structures, making per-structure methods redundant and expensive. In our evaluation, we show that storage structures perform poorly for file mapping (§4.9).

In summary, we make the following contributions:

- We present the design and implementation of four file mapping approaches and explore various PM optimizations to them (§3). We implement these approaches in the Strata file system [29], which provides state-of-the-art PM performance, particularly for small, random IO operations, where file mapping matters most (§4.4). We perform extensive experimental analyses on how the page cache, mapping structure size, IO size, storage device utilization, fragmentation, isolation mechanisms, and file access patterns affect the performance of these file mapping approaches (§4).
- We show that our PM-optimized HashFS is the best performing mapping approach, due to its low memory overhead and low latency (§4.1) during random reads and insertions. We demonstrate that this approach can increase throughput by up to 45% over Strata's page-cached extent trees in YCSB workloads (§4.8).
- We show that using a traditional page cache provides no benefit for PM-optimized file mapping (§4.7). We also demonstrate that PM-optimized storage structures are ill-suited for use as file mapping structures and cannot be simply dropped into PM-optimized file systems (§4.9).

2 File Mapping Background

What is file mapping? *File mapping* is the operation of mapping a logical offset in a file to a physical location on the underlying device. File mapping is made possible by one or more metadata structures (*file mapping structures*) maintained by the file system. These file mapping structures map logical locations (a file and offset) to physical locations (a device offset) at a fixed granularity. File mapping structures have three operations: *lookups*, caused by file reads and writes of existing locations; *insertions*, caused by file appends or IO to unallocated areas of a sparse file; and *deletions*, caused by file truncation (including file deletion) or region de-allocation in a sparse file. Inserts and deletions require the (de-)allocation of physical locations, typically provided by a *block allocator* maintaining a separate free block structure. PM file systems generally map files at block granularity of at least 4KB in order to constrain the amount of metadata required to track file system space [15, 29, 52]. Block sizes often correspond to memory page sizes, allowing block-based PM file systems to support memory-mapped IO more efficiently.

2.1 File Mapping Challenges

The following properties make designing efficient file mapping challenging. Careful consideration must be taken to find points in the trade-off space presented by these properties that maximize performance for a given workload and storage device. PM intensifies the impact of many of these properties.

Fragmentation. Fragmentation occurs when a file's data is spread across non-contiguous *physical locations* on a storage device. This happens when a file cannot be allocated in a single contiguous region of the device, or when the file grows and all adjacent blocks have been allocated to other files. Fragmentation is inevitable as file systems age [42] and can occur rapidly [9, 23].

Fragmentation can magnify the overhead of certain file mapping designs. Many traditional file systems use compact file mapping structures, like extent trees (§3.1), which can use a single *extent entry* to map a large range of contiguous file locations. Fragmentation causes locations to become non-contiguous and the mapping structure to become larger, which can increase search and insert times (§4). This is referred to as *logical fragmentation* and is a major source of overhead in file systems [19], especially on PM (as file mapping constitutes a larger fraction of the IO path on PM file systems). Another consequence of fragmentation is the reduction in sequential accesses. Fragmentation results in the separation of data structure locations across the device, which causes additional, random IO on the device. This degrades IO performance, particularly on PM devices [22].

Fragmentation must be considered as an inevitable occurrence during file mapping, since defragmentation is not always feasible or desirable. Defragmentation is an expensive operation—it can incur many file writes, which can lower device lifespan by more than 10% in the case of SSDs [19]. Similar concerns exist for PM, which may make defragmentation undesirable for PM-optimized file systems as well [18, 59].

Locality of reference. Locality of reference when accessing a file can be used to reduce file mapping overhead. Accesses with locality are typically accelerated by caching prior accesses and prefetching adjacent ones. OS page caches and CPU caches can achieve this transparently and we discuss the role of the OS page cache for PM file mapping in §2.2. However, approaches specific to file mapping can yield further benefits. For example, we can hide part of the file mapping structure traversal overhead for accesses with locality, by remembering the meta-data location of a prior lookup and prefetching the location of the next lookup. Further, with an efficient cache in place, the file mapping structure itself should be optimized for random (i.e., non-local) lookups, as the structure is referenced primarily for uncached mappings.

Mapping structure size. The aggregate size of the file mapping structures as a fraction of available file system space is an important metric of file mapping efficiency. Ideally, a file mapping structure consumes a small fraction of available space, leaving room for actual file data storage. Furthermore, the size of the mapping structure impacts the amount of data that can remain cache-resident.

Traditional file mapping structures are designed to be *elastic*—the size of the structure is proportional to the number of locations allocated in the file system. This means that as the number and size of files increase, the size of the file mapping structure grows as well. However, elastic mapping structures introduce overhead by requiring resizing, which incurs associated space (de-)allocation and management cost.

Concurrency. Providing isolation (ensuring that concurrent modifications do not affect consistency) can be an expensive operation with limited scalability. This is a well-known problem for database mapping structures (called *indexes*) that support operations on arbitrary ranges. For example, insertions and deletions of tree-based range index entries may require updates to inner tree nodes that contend with operations on unrelated keys [16, 49].

Isolation is simpler in *per-file* mapping structures, where the variety and distribution of operations that can occur concurrently is limited. With the exception of sparse files, updates only occur during a change in file size, i.e. when appending new blocks or truncating to remove blocks. File reads and in-place writes (writes to already allocated blocks) only incur file mapping structure reads. For this reason, it is often sufficient to protect per-file mapping structures with a coarse-grained reader-writer lock that covers the entire structure [29].

The most common scenario for per-file mapping structure concurrent access is presented by workloads with one writer (updating the file mapping structure via appends or truncates), with concurrent readers (file reads and writes to existing offsets). In this case, file mapping reads and writes can proceed without contention in the common case. Other mapping structure operations can impact concurrency, but they generally occur infrequently. For example, some file mapping structures require occasional resizing. For consistency, this operation occurs within a critical section but is required only when the structure grows beyond a threshold. Extent trees may split or merge extents and these operations require partial-tree locking (locking inner nodes) but occur only on updates.

For *global* file mapping structures, the possibility of contention is higher, as there can be concurrent writers (appending/truncating different files). Global file mapping structures have to be designed with concurrency in mind. In this case, contention does not favor the use of tree-based indices, but is tractable for hash table structures with per-bucket locks, as contention among writers only occurs upon hash collision.

2.2 File Mapping Non-Challenges

Crash consistency. An overarching concern in file systems is providing consistency—transitioning metadata structures from one valid state to another, even across system crashes. File systems have many metadata structures that often need to be updated atomically across crashes, i.e. a free-list and a file-mapping structure when a new location is allocated for a file. File systems generally employ some form of journaling to ensure these updates can be replayed and made atomic and consistent, even in the case of a crash. This makes crash consistency a non-challenge for file mapping. This is unlike PM persistent data structures, which are typically designed to provide crash consistency within the structure itself.

Page caching. Traditionally, file systems read data and metadata (including file-mapping structures) into a *page cache* in DRAM before serving it to the user. The file system batches updates using this page cache and writes back dirty pages. This is a necessary optimization to reduce the overhead of reading directly from a block device for each IO operation. However, page caches have overheads. A pool of DRAM pages must be managed and reallocated to new files as they are opened, and the pages in the cache must be read and written back to ensure updates are consistent.

For PM, an OS-managed page cache in DRAM may no longer be required for file mapping structures. One analysis [48] found that eschewing the page cache for inodes and directories results in better performance in PM-optimized file systems. Until now, there has been no such consensus on the optimal design point for file mapping structures: some PM-file systems maintain file mapping structures only in PM [15], others maintain file mapping structures only in DRAM [52, 56], still others manage a cache of PM-based file mapping structures in DRAM [11, 29]. However, as we show in §4.7, page-cache management turns out to be a non-challenge, as it is always more efficient to bypass page-caching for file-mapping structures on PM-optimized file systems.

3 PM File Mapping Design

Based on our discussion of the challenges (§2.1), we now describe the design of four PM-optimized file mapping approaches, which we analyze in §4. We first describe two traditional, per-file mapping approaches and their PM optimizations, followed by two global mapping approaches. We discuss the unique challenges faced by each approach, followed by a description of the approach's mapping structure.

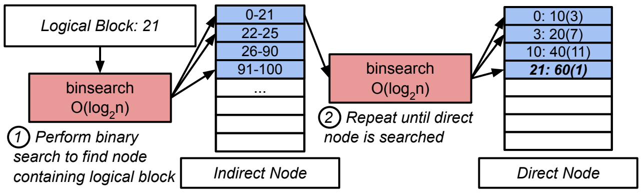


Figure 1: An extent tree lookup example. Each indirect node lists the range of logical blocks in the direct nodes it points to (shown as `<start block-end block>`). Each direct block contains *extents* that map ranges of logical blocks to physical blocks, represented as `<logical block start: physical block start (number of blocks)>`. Traversal of nodes is repeated until the direct mapping node is found.

3.1 Traditional, Per-File Mapping

File mapping is done *per-file* in most traditional file systems. Each file has its own elastic mapping structure which grows and shrinks with the number of mappings the structure needs to maintain. The mapping structure is found via a separate, fixed-size inode table, indexed by a file's *inode number*. Per-file mapping generally provides high spatial locality, as all mappings for a single file are grouped together in a single structure, which is traditionally cached in the OS page cache. Adjacent logical blocks are often represented as adjacent mappings in the mapping structure, leading to more efficient sequential file operations. Concurrent access is a minor problem for per-file mapping approaches due to the restricted per-file access pattern (§2.1). Physical block allocation is performed via a separate block allocator, which is implemented using a red-black tree in our testbed file system (§4).

Challenges. Per-file mapping structures must support resizing, which can be an expensive operation. Since they need to grow and shrink, per-file mapping structures have multiple levels of indirection, requiring more memory references for each mapping operation. Fragmentation destroys the compact layout of these structures and exacerbates overhead by increasing the amount of indirection and thus memory accesses.

We now discuss two common mapping structures used for per-file mapping.

3.1.1 Extent Trees

The extent tree is a classic per-file mapping structure, but is still used in modern PM-optimized file systems [11, 29]. Extent trees are B-trees that contain *extents*, which map file logical blocks to physical blocks, along with a field that indicates the number of blocks the mapping represents. Extents can also be indirect, pointing to further extent tree nodes rather than to file data blocks. In order to perform a lookup, a binary search is performed on each node to determine which entry holds the desired physical block. This search is performed on each level of the extent tree, as shown in Fig. 1.

We create a PM-optimized variant of extent trees, based on the implementation from the Linux kernel [33]. Traditionally, extent tree operations are performed on copies of tree blocks in the page cache. As using a page cache for PM mapping structures leads to unnecessary copy overhead (see §4.7), we instead design our extent tree to operate directly on PM. Consistency under concurrent extent access is guaranteed by a per-extent entry valid bit, which is only enabled once an extent is available to be read. Consistency for complex operations (resizing, splitting extents, etc.) is provided by an undo log. We also keep a cursor [5] in DRAM of the last accessed path, improving performance for accesses with locality.

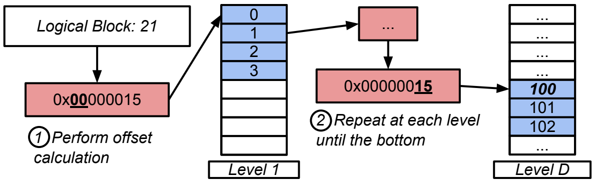


Figure 2: An example radix tree performing a lookup. Each level is indexed by using a portion of the logical block number as an offset. The last level offset contains a single physical block number.

Design considerations. Extent trees have the most compact representation of any of the mapping structures that we evaluate, since multiple mappings can be coalesced into a single extent, leading to small structure size. Extent trees, however, require many memory accesses, as they must perform a binary search at each level of the extent tree to find the final mapping. Cursors can simplify the search, but only for sequential scans through the structure and repeat block accesses.

3.1.2 Radix Trees

Radix trees are another popular per-file mapping structure [15], shown in Fig. 2. Each node in a radix tree occupies a physical block on the device (typically 4KB), with the exception of a few entries that can be stored within the inode directly. A lookup starts at the top level node resolving the top N bits of the logical block number. With 8 byte pointers and 4KB per node, we can resolve $N = log_2(\frac{4096}{8}) = 9$ bits per radix tree node. The second level node resolves the next N bits, and so on. Radix trees grow and shrink dynamically to use as many levels as required to contain the number of mappings (e.g., a file with $N < 9$ would only need a single-level tree). The last level node contains direct mappings to physical blocks. To accelerate sequential scans, a cursor in DRAM is typically used to cache the last place searched in the tree. As each pointer is simply a 64-bit integer (a physical block number), consistency can be guaranteed through atomic instructions of the CPU. Consistency for resizing is provided by first recording modifications in an undo log.

Design considerations. Radix trees are less compact than extent trees, and typically require more indirection. Thus, tree traversal will take longer on average, since radix trees grow faster than the compact extent trees. However, radix trees have the computationally simplest mapping operation—a series of offset calculations, which only require one memory access per level of the tree. Hence, they often perform fewer memory accesses than extent trees, as extent trees make $O(log(N))$ memory accesses per extent tree node (a binary search) to find the next node to traverse.

3.2 PM-specific Global File Mapping

Based on our analysis of file mapping performance and the challenges posed by per-file mapping approaches, we design two PM-specific mapping approaches that, unlike the per-file mapping approaches used in existing PM-optimized file systems, are *global*. This means that, rather than having a mapping structure per file, these approaches map both a file, as represented by its inode number (inum), and a logical block to a physical block. Hence, they do not require a separate inode table.

Both of the global mapping structures we design are hash tables. The intuition behind these designs is to leverage the small number of memory accesses required by the hash table structure, but to avoid the complexity of resizing per-file mapping structures as files grow and shrink. We are able to statically create these global structures, as the size of the file system—the maximum number of physical blocks—is known at file-system creation time. Hash tables are not affected by fragmentation as they do not use a compact layout. Consistency under concurrent access can frequently be resolved on a per hash bucket basis due to the flat addressing scheme.

We employ a fixed-size translation cache in DRAM that caches logical to physical block translations to accelerate lookups with locality. It is indexed using the same hash value as the full hash table for simplicity. The goal of the fixed-size translation cache is to provide a mechanism for the hash table structures that is similar to the constant-size cursor that the extent and radix trees use to accelerate sequential reads [5]. This cache contains 8 entries and is embedded in an in-DRAM inode structure maintained by Strata, our testbed file system (§4). This fixed-size translation cache is different from a page cache, which caches the mapping structure, rather than translations. The translation cache reduces the number of PM reads required for sequential read operations, benefiting from the cache locality of the in-DRAM inode structure, accessed during file system operations (§4.1).

Global Structure Challenges. Since these global mapping structures are hash tables, they exhibit lower locality due to the random nature of the hashing scheme. These global hash tables potentially exhibit even lower locality than a per-file hash table might experience, since a per-file hash table would only contain mappings relevant to that file, where the mappings in a global hash table may be randomly distributed across a much larger region of memory. However, the translation cache ameliorates this challenge and accelerates mappings with locality.

We now discuss two global file mapping hash table designs.

3.2.1 Global Cuckoo Hash Table

We show a diagram for the first global hash table structure in Fig. 3. In this hash table, each entry maps `<inum>`, `<logical block>: <physical block>(<# of blocks>)`. This hash table uses cuckoo hashing [38], which means each entry is hashed twice using two different hash functions. For lookups, at most two locations have to be consulted. For insertions, if both potential slots are full, the insertion algorithm picks one of the existing entries and evicts it by moving it to its alternate location, continuing transitively until there are no more conflicts. We use cuckoo hashing for this design instead of linear probing to avoid having to traverse potentially long chains of conflicts in pathological cases, bounding the number of memory accesses required to find a single index to 2.

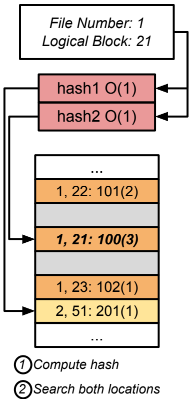


Figure 3: Global hash table with cuckoo hashing.

The hash table is set up as a contiguous array, statically allocated at file-system creation. Consistency is ensured by first persisting the mapping information (physical block number, size) before persisting the key (inum, logical block), effectively using the key as a valid indicator. Consistency for complex inserts (i.e., inserts which cause shuffling of previous entries) is maintained by first recording operations in the file-system undo log. As complex updates are too large for atomic compare-and-swap operations, we use Intel TSX [21] in place of a per-entry lock to provide isolation. This isolation is required as inserts may occur concurrently across files—this is not a challenge for per-file mapping structures, which can rely on per-file locking mechanisms for isolation.

One issue with hash tables is that they generally present a one-to-one mapping, which is not conducive to representing ranges of contiguously allocated blocks like traditional mapping structures. To compensate, each entry in this hash table also contains a field which includes the number of file blocks that are contiguous with this entry. This mapping is maintained for every block in a series of contiguous blocks; for example, if logical blocks 21..23 are mapped contiguously to device blocks 100..102, the hash table will have entries for 100 with size 3 (shown in Fig. 3 as `100(3)`), 101 with size 2, and so on. Each entry also contains a reverse index field which describes how many blocks come before it in a contiguous range so that if a single block is removed from the end of a contiguous block range, all entries in the group can be updated accordingly. This hash table can also do parallel lookups for multiple blocks (e.g., by using SIMD instructions to compute hashes in parallel) to make lookup more efficient for ranged accesses (i.e. reading multiple blocks at a time) if fragmentation is high. Ranged nodes are crucial for the performance of large IO operations (we discuss this further in

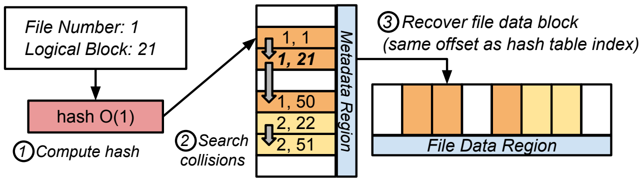


Figure 4: HashFS and an example lookup operation. Rather than storing a physical block number in the hash entry, the offset in the hash table is the physical block number.

§4.4). Ranged nodes are also cached in the translation cache to accelerate small, sequential IO operations.

Design considerations. There is a trade-off between cuckoo hashing and linear probing. Cuckoo hashing accesses at most 2 locations, but they are randomly located within the hash table, resulting in poor spatial locality. Linear probing may access more locations when there are conflicts, but they are consecutive and therefore have high spatial locality with respect to the first lookup. Therefore, a linear probing hash table with a low load factor may access only 1 or 2 cache lines on average, which would outperform a cuckoo hash table in practice. We explore this trade-off by comparing this global hash table scheme to a linear probing scheme (described in §3.2.2) to determine which scheme has better performance in practice—we measure the latency of lookups and modifications separately in our microbenchmarks (§4.1–§4.5) and end-to-end performance in our application workloads (§4.8).

3.2.2 HashFS

We also design a second global hash table, with the idea of specifically reducing the cost of the insert operation, which normally involves interacting with the block allocator. To this end, we build a single hash table that also provides block allocation, which we call *HashFS*. We present a diagram of this structure in Fig. 4.

HashFS is split into a metadata region and a file data region. Physical blocks are stored in the file data region, which starts at $fileDataStart$. A lookup resolves first to the metadata region. The corresponding physical block location is calculated from the offset of the entry in the metadata region. For example, if the hash of `<inum=1, lblk=21>` resolves to offset i in the metadata region, the location of the corresponding physical block is $(fileDataStart + i \times blockSize)$, with $blockSize = 4KB$.

Unlike the cuckoo hash table (§3.2.1), this table does not have any ranged nodes, instead providing a pure one-to-one mapping between logical blocks and physical blocks. This uses a constant space of 8 bytes per 4KB data block in the file system, for a total space overhead of $< 0.2\%$ of PM. Another advantage to this scheme is that the hash table entries are extremely simple—a combined inum and logical block, which fits into a 64-bit integer. Consistency can therefore be guaranteed simply with intrinsic atomic compare-and-swaps.

In order to efficiently implement large IO and sequential access, we perform vector hash operations using SIMD instructions. This is possible due to PM's load/store addressability and adequate bandwidth. If an IO operation does not use the maximum SIMD bandwidth, the remaining bandwidth is used to prefetch following entries, which are then cached in the translation cache. The global hash table uses linear probing, which inserts conflicting hash values into an offset slot in the same area [8]. This is in contrast to separate chaining, which allocates separate memory for a linked list to handle conflicts. An advantage of linear probing is that conflicting entries are stored in adjacent locations, reducing the overhead of searching conflicts by avoiding misses in the PM buffer [43]. Additionally, unlike cuckoo hashing, linear probing never relocates entries (i.e., rehashing), which preserves the correspondence between the index of the metadata entry and file data block. As discussed previously, we measure the trade-offs of linear probing and cuckoo hashing in our evaluation (§4.1–§4.5 and §4.8).

How is file mapping affected by...	Design Question	§
Locality?	Optimize for specific workloads?	4.1
Fragmentation?	Make robust against file system aging?	4.2
File size?	Specialize for different file sizes?	4.3
IO size?	Optimize for sequential access?	4.4
Space utilization?	Make file mapping structure elastic?	4.5
Concurrency?	Is ensuring isolation important?	4.6
Page caching?	Is page caching necessary?	4.7
Real workloads?	Are mapping optimizations impactful?	4.8
Storage structures?	Can we reuse PM storage structures?	4.9

Table 1: The questions we answer in our evaluation.

HashFS is not fundamentally limited to 8 byte entries. Rather than performing atomic compare-and-swap operations on an 8 byte entry, other atomic update techniques can ensure isolation (e.g., Intel TSX). The existing logging mechanisms provide crash consistency regardless of the size of the HashFS entries. The overall space overhead is low, even with larger entries (e.g., $< 0.4\%$ of the total file system capacity with 16 byte entries).

Design considerations. HashFS, like the global cuckoo hash table, also has very low spatial locality, but improves sequential access by prefetching via SIMD parallelism. HashFS's computational overhead is low on average, as HashFS only has to compute a single hash function per lookup. However, conflicts are expensive, as they are resolved by linearly following chains of entries. This means that HashFS may perform worse at high load factors (i.e. as the file system becomes more full).

4 Evaluation

We perform a detailed evaluation of the performance of our PM file-mapping approaches over a series of microbenchmarks and application workloads and discuss the performance characteristics of each mapping structure. We then demonstrate that the non-challenges we discussed in §2.2 are indeed non-challenges in PM-optimized file systems. We summarize the questions we answer in our analysis in Table 1.

Experimental setup. We run our experiments on an Intel Cascade Lake-SP server with four 128GB Intel Optane DC PM modules [20]. To perform our analysis, we integrate our mapping approaches into the Strata file system [29]. We choose Strata for our analysis as it is one of the best performing open-source PM-optimized file systems, which is actively used by state-of-the-art research [39]. We configure Strata to only use PM (i.e., without SSD or HDD layers). Our file-mapping structures are integrated into both the user-space component of Strata (LibFS, which only reads mapping structures) and the kernel-space component of Strata (KernFS, which reads and updates mapping structures based on user update logs "digested" [29] from LibFS). Each experiment starts with cold caches and, unless noted, mapping structures are not page cached.

Generating fragmentation. We modify the PM block allocator in Strata to accept a *layout score* parameter at file-system initialization that controls the level of fragmentation encountered during our experiments. Layout score is a measure of file-system fragmentation which represents the ratio of file blocks that are contiguous to non-contiguous file blocks [42]. A layout score of 1.0 means all blocks are contiguous and a layout score of 0.0 means no blocks are contiguous. We allocate blocks in fragmented chunks such that the resulting files have an average layout score that is specified by the initialization parameter, which fragments both allocated file data and free space. These fragmented chunks are randomly distributed throughout the device to simulate the lack of locality experienced with real-world fragmentation. By making this modification, we can effectively simulate fragmentation without using high-overhead aging methods [25, 42]. Unless otherwise stated, we use a layout score of 0.85 for our experiments, as this was determined to be an average layout score for file systems in past studies [42].

Experimental results. Unless otherwise stated, we report the average latency of the file mapping operation over 10 repeated measurements, including all overhead associated with the mapping structure, such as hash computation and undo logging. Error bars report 95% confidence intervals. For insertion/truncation operations, the latency of the file mapping operation includes the overhead of the block allocator. Note that HashFS uses its own block allocation mechanisms (§3.2.2). All other evaluated file mapping structures use the block allocator already present in Strata.

4.1 Locality

We analyze how particular access patterns impact the performance of file mapping. We perform an experiment using a 1GB file where we perform single-block reads and appends in either a "cold cache" scenario (i.e., only perform one operation and then reset the experiment) or a "hot cache" scenario (i.e., perform 100,000 operations). We perform reads and appends as they exercise file mapping structure reads and file mapping structure writes, respectively. We measure the latency of the file mapping operation and report the average (in CPU cycles) in Fig. 5.

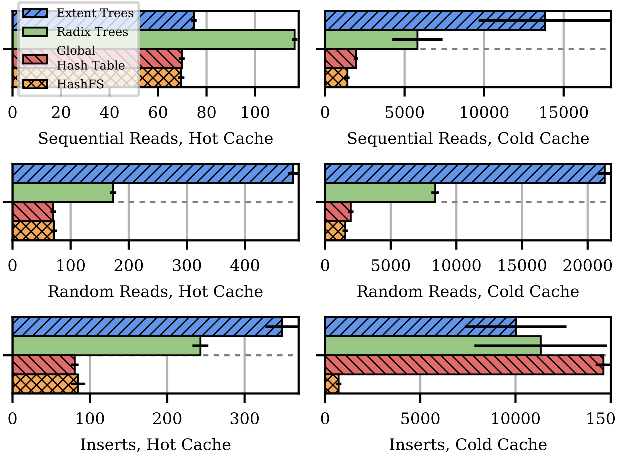


Figure 5: Locality test, measuring the latency of the file mapping operation in cycles (using `rdtscp`).

On average, hot cache operations result in low latency for all file mapping approaches, especially for sequential reads, for which the per-file cursor optimizations and global SIMD prefetching were designed. The greatest difference is in the cold cache case, where global file mapping is up to 15× faster than per-file mapping. This is due to the cost of tree operations when operating on a cold cache. Tree operations must traverse multiple levels of indirection, making multiple memory accesses per level of indirection; the global file-mapping structures, on the other hand, make fewer than two accesses to PM on average. The exception to the performance benefits of global file-mapping structures is the performance of cold global cuckoo hash table inserts, which perform on par with radix trees (within confidence intervals) and extent trees. The main factor causing the high variability in cold cache insert performance for these structures is the overhead of the block allocator, which also has persistent and volatile metadata structures which incur last level cache misses (e.g., a persistent bitmap, a volatile free-list maintained as a red-black tree, and a mutex for providing exclusion between kernel threads [29]). In contrast, HashFS, which does not use Strata's block allocator, is between 14–20× faster than all other file-mapping structures in this case. Overall, this initial experiment shows us that a global hash table structure can dramatically outperform per-file tree structures, particularly for access patterns without locality.

4.2 Fragmentation

We now measure the effect that file-system fragmentation has on file mapping. We perform this experiment on a single 1GB

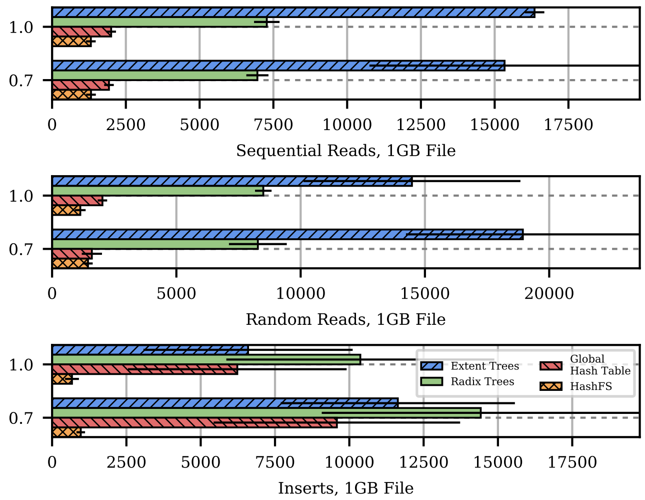


Figure 6: Fragmentation test, measuring the impact of high (0.7) and low (1.0) fragmentation on file mapping latency.

file, and vary the fragmentation of the file from no fragmentation (layout score 1.0) to heavily fragmented (layout score 0.7 represents a heavily-fragmented file system according to prior work [42]). We measure the average "cold cache" file mapping latency (i.e., one operation before reset, as described in the previous experiment) over 10 trials of each operation and present the results of our experiment in Fig. 6.

Between the different levels of fragmentation, the only major difference is the performance of the extent trees. Intuitively, this difference arises from the way that extent trees represent extents—if the file is not fragmented, the extent tree can be much smaller and therefore much faster to traverse. The other evaluated mapping approaches are unaffected by fragmentation for reads. For insertions, the block allocator takes longer on average to find free blocks, giving HashFS an advantage. We conclude that per-file mapping performs poorly on fragmented file systems, while HashFS is unaffected by fragmentation.

4.3 File Size

The per-file tree structures have a depth that is dependent on the size of the file, whereas the global hash table structures are flat. We therefore measure the latency of file mapping across three file sizes: 4KB, 4MB, and 4GB, representing small, medium, and large files. We report the average latency (over 10 trials) for each file mapping operation in Fig. 7.

As expected, the per-file structures grow as the size of the file increases, which results in more indirect traversal operations per file mapping, resulting in longer latency. The range of this increase is naturally smaller for sequential reads (less than 10% across extent trees and radix trees for sequential reads, but 34% for random reads) due to the inherent locality of the data structure). The mapping overhead for the hash tables, however, is static, showing that the hash table structures do not suffer performance degradation across file sizes.

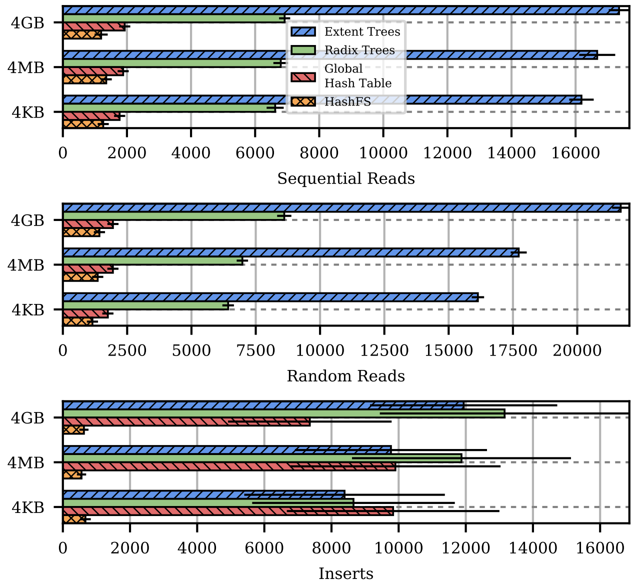


Figure 7: Impact of file size on file mapping latency.

4.4 IO Size

Our previous experiments perform single block operations (i.e., reading or writing to a single 4KB block). However, many applications batch their IO to include multiple blocks. We therefore measure the impact on file mapping as an application operates on 4KB (a single block), 64KB (16 blocks), and 1MB (256 blocks). We then measure the proportion of the IO path that comprises file mapping and report the average (over 10 trials) in Fig. 8. Since IO latency increases with IO size regardless of the mapping structure, we present the results of this experiment as a ratio of IO latency to normalize this increase in latency and to specifically show the impact of the file mapping operation on the overall latency.

As the number of blocks read in a group increases, the gap between the tree structures and the hash table structures closes. Radix and extent trees locate ranges of blocks together in the same leaf node and thus accelerate larger IO operations. At the same time, as the number of blocks increases, the proportion of the file system time spent on file mapping drops dramatically. For example, for an IO size of 1MB, radix tree mapping takes only 3% of the IO path for reads. At this IO size, radix trees perform best for both sequential and random reads. However, the impact of this performance increase is not seen at the application level (§4.8). We also note that without the ranged node optimization introduced for the cuckoo hash table (§3.2.1), the mapping latency ratio would remain constant for the global cuckoo hash table, rather than decreasing as the IO size increases.

We also see the advantage HashFS has over the global cuckoo hash table. Not only does HashFS have lower file mapping latency for all operations compared to cuckoo hashing, it also has the lowest insertion time, since it is able to bypass the traditional block allocation overhead.

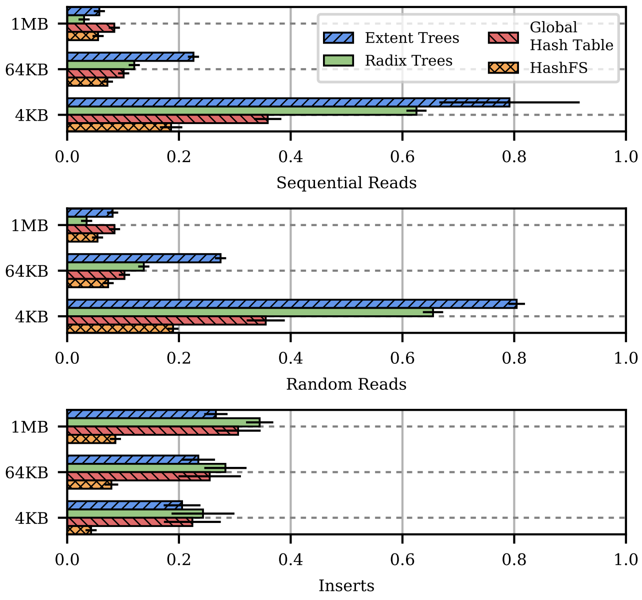


Figure 8: IO size test. We report the average proportion of the IO path spent performing file mapping.

4.5 Space Utilization

A potential disadvantage of using a global hash table structure is that collisions grow with the percentage of the structure used, meaning that as file system utilization grows, file mapping latency may increase as the number of collisions increase. We measure this effect by creating a file system with a total capacity of 128GB. We then measure the proportion of the IO path that comprises file mapping when the overall space utilization ranges from moderately full (80%) to extremely full (95%). Our prior experiments demonstrate the performance on a mostly empty file system. We perform this microbenchmark using "cold cache" (i.e., one operation per file), single-block file mapping operations as per previous experiments. We report the average file mapping latency ratio (over 10 trials) in Fig. 9. Since IO latency increases with overall file system utilization, we present the results of this experiment as a fraction of IO latency to account for the increase in overall latency.

As expected, the latency of the mapping operation for per-file tree structures is unchanged when comparing low utilization to high utilization. The main difference is the latency of the two global hash table structures. At low utilization, as seen in previous experiments (§4.4), HashFS outperforms global cuckoo hashing. At 80% utilization, the performance of random and sequential reads are very similar for HashFS and the global cuckoo hash table (within ±3%, or within error). HashFS still outperforms the global cuckoo hashing table for insertions. At 95% utilization, however, HashFS incurs between 12–14% more latency for reads than the global cuckoo hash table, and has equivalent performance for insertions.

We conclude that while higher utilization causes HashFS to degrade in performance relative to global cuckoo hashing, both file mapping structures still outperform the per-file mapping structures even at high utilization.

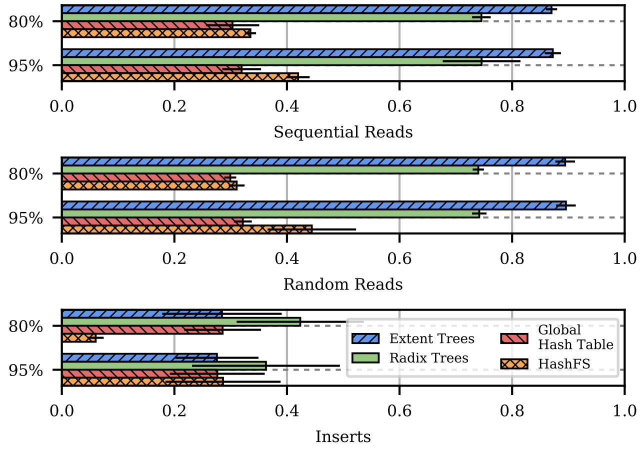


Figure 9: Space utilization test, measuring the proportion of the IO path spent in file mapping versus the overall utilization of the file system (in percent).

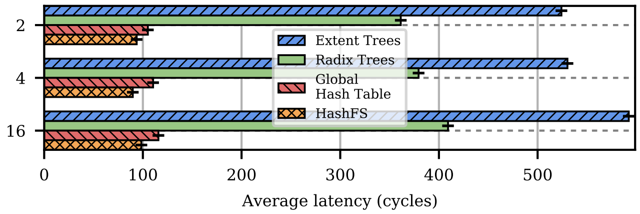


Figure 10: Average file mapping latency, varying # of threads.

4.6 Concurrency

We now conduct an experiment to quantify how well our file-mapping approaches perform under concurrent access. To simulate a high-contention scenario, we conduct an experiment with multiple threads reading and writing the same file (every thread reads and then appends to the same file). This causes high read-write contention between Strata's KernFS component which asynchronously updates the file-mapping structures and the user-space LibFS which reads the file-mapping structures, causing contention for both the per-file and global file-mapping structures. Each file-mapping structure manages its own synchronization—Strata manages the synchronization of other file metadata via a per-file reader-writer lock.

We show the result of the experiment in Fig. 10. We see that, as the number of threads increases, the latency of file mapping increases slightly (up to 13% for extent trees for an 8× increase in concurrency) across all file mapping approaches. Furthermore, the ranking among approaches does not change across any number of threads. This shows that common concurrent file access patterns allow per-file mapping approaches to use coarse-grained consistency mechanisms without impact on scalability, while the transactional memory and hash bucket layout optimizations for our two global file mapping

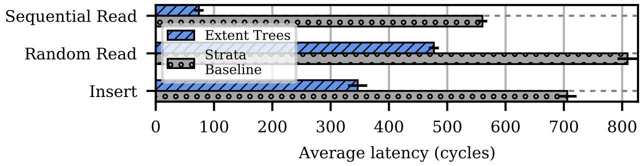


Figure 11: Page-cache experiment, measuring the average file-mapping latency of PM-optimized extent trees and page-cached extent trees (Strata Baseline).

Workload	Average file size	# of files	IO size (R/W)	R/W ratio
fileserver	128KB	1,000	1MB/16KB	1:2
webproxy	16KB	1,000	16KB/16KB	5:1

Table 2: Filebench workload configurations.

approaches can effectively hide synchronization overheads.

In summary, we find that the isolation mechanisms used in our PM-optimized file-mapping structures are not bottlenecks. All structures are scalable for common file access patterns.

4.7 Page Caching

We now discuss why traditional page-caching should not be employed for PM file-mapping structures. To demonstrate why, we provide a microbenchmark that compares Strata's default mapping structure (page-cached extent trees) to our implementation of extent trees (which is based on Strata's implementation, but bypasses the page cache and operates directly on PM). In this experiment, we open a 1GB file and perform 1,000,000 operations on it (single block reads or inserts) and report the average file mapping latency in cycles.

We show the results in Fig. 11. We can see that, even after many iterations, the dynamic allocation overhead of the page cache is not amortized in the read case. The insertion case is also slower due to the page cache overhead—the PM-optimized extent trees write updates directly back to PM.

Based on the results of this experiment, it is clearly more beneficial to perform file mapping directly on PM, as it reduces the number of bytes read and written to the device, reduces DRAM overhead, and decreases the overall overhead of the IO path. Therefore, we advocate for the use of lower overhead caching methods, such as cursors, rather than relying on the page cache.

4.8 Application Workloads

We provide two application benchmarks to measure the overall benefit our PM-optimized file-mapping structures have on application throughput. We compare our file-mapping structures to the file-mapping structure present in Strata, which is a per-file, page-cached extent tree. We use this file-mapping structure as our baseline, as Strata is a state-of-the-art PM-optimized file system [29, 39].

Filebench. We test our file mapping structures using filebench [45], a popular file-system-testing framework which has been used to evaluate many PM-optimized file systems [29, 52, 56]. We select the fileserver (write-heavy) and webproxy (read-heavy) workloads. These workloads test file mapping structure reads, updates, and deletions, and emulate the performance of mapping structures as they age under repeated modifications. We describe the characteristics of these workloads in Table 2.

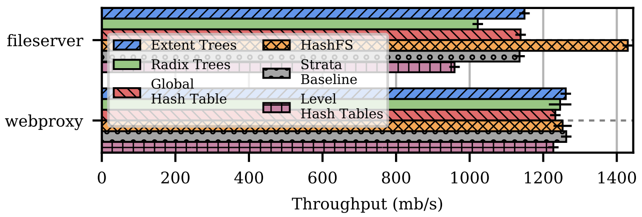


Figure 12: Filebench results for one write-heavy workload (fileserver) and one read-heavy workload (webproxy).

Workload	Characterization	Example
A	50% reads, 50% updates	Session activity store
B	95% reads, 5% updates	Photo tagging
C	100% reads	User profile cache
D	95% reads, 5% inserts	User status updates (read latest)
E	95% scans, 5% inserts	Threaded conversations
F	50% reads, 50% read-modify-write	User database

Table 3: Description of YCSB workload configurations.

We show the results of the filebench experiments in Fig. 12. In the fileserver workload, HashFS outperforms the baseline by 26%, while the other mapping structures perform similarly to the baseline. This result is explained by the IO size microbenchmark (§4.4). In this microbenchmark, HashFS performs the best for insertions, which is the predominant operation in this workload. In this workload, the radix trees perform the worst, experiencing a 10% throughput drop versus the baseline. Insert performance of radix trees is the worst among our file mapping approaches (Fig. 8). Extent trees and the global cuckoo hash table both perform similarly for large IO reads and smaller block insertions, so they perform similarly here and are not an improvement over the baseline.

The webproxy workload does not show any major difference in throughput across the file mapping approaches (all within ±2%, or within error). This is because this is a read-heavy workload on relatively small files with hot caches, and we show in §4.1 that the performance across file mapping structures is very similar in this case.

YCSB. We also evaluate the end-to-end performance on key-value workloads using YCSB [10] on LevelDB [17], a common benchmark to measure the performance of PM-optimized file systems [29]. We measure the throughput for all standard YCSB workloads (A–F). YCSB uses a Zipfian distribution to select keys for operations. We report the characteristics of these workloads in Table 3. We configure our YCSB tests to

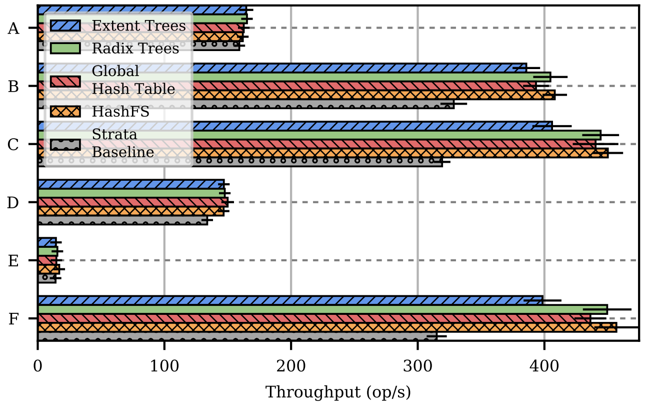


Figure 13: YCSB running on LevelDB. We report the average overall throughput (operations/second) on all workloads (A–F).

use a key-value database containing 1.6 million 1 KB key-value pairs (a 1.5 GB database), using a maximum file size of 128 MB for the LevelDB backend running on 4 threads. Our experiments perform 100,000 operations on the database, as dictated by the YCSB workload.

We show the results of our YCSB tests in Fig. 13. Workloads A and E are primarily bounded by the in-memory operations of LevelDB (e.g. performing read and scan operations), and not bounded by the file system, thus we see similar throughput across all mapping approaches (within 3% of the baseline). However, for the other workloads (B, C, D, F), HashFS provides the best performance, providing between 10–45% increase in throughput on workload F versus the other file mapping approaches. Radix trees and the global cuckoo hash tables are both slower by 2–4% on average than HashFS, but the PM-optimized extent trees perform worse than HashFS by 13% in workloads C and F. This is due to the generally poor performance of the extent tree structure for random reads, which dominate these workloads. In these workloads, the default file mapping structure in Strata spends 70% of the file IO path in file mapping—this provides ample opportunity for improvement, which is why HashFS is able to increase the overall throughput by up to 45% in these cases. We further discuss the performance of the baseline in our discussion on concurrency (§4.6).

We also show how concurrency impacts file mapping in real workloads by rerunning our YCSB benchmark using a single thread (Fig. 14).This experiment differs from the multi-threaded version (Fig. 13) other than the overall magnitude of the throughput. Additionally, the increase in throughput over the baseline is much higher (45%) in the multi-threaded experiment than the single threaded experiment (23%). Upon further investigation, we find that this is because the Strata *page cache* is not scalable. Strata's page cache maintains a global list of pages, with a single shared lock for consistency.

Summary. We draw two conclusions from these experiments: (1) HashFS results in the best overall throughput among our PM-optimized mapping structures; and (2) HashFS always matches or exceeds the performance of Strata's default mapping structure (page-cached extent trees). We therefore conclude that HashFS file mapping provides the best overall performance in real application workloads.

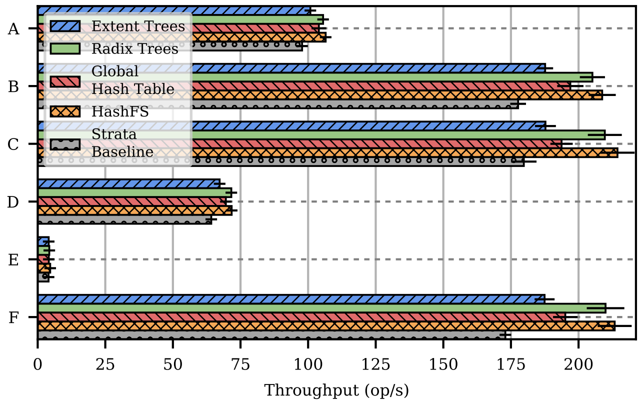


Figure 14: YCSB experiment using a single thread.

4.9 File Mapping via Level Hashing?

We examine whether PM storage structures can be efficiently used for file mapping, thus enabling the reuse of prior work. We select level hashing [7, 59], a state-of-the-art hash-table storage structure for PM, as a case study. Level hashing outperforms RECIPE-converted structures [32] and, as a hash table, is a good contender against our best-performing file mapping structures, which are also hash tables. The goal of this case study is to see if general-purpose PM data structures proposed in prior work can be used as file mapping structures, as-is. Hence, we do not apply any file-system specific optimizations to level hashing.

We evaluate how level hashing performs relative to our PM file mapping structure in our filebench workloads, shown in Fig. 12. In all cases, HashFS outperforms level hashing. In particular, for the fileserver workload (the most write heavy workload), level hashing underperforms even Strata's baseline. In particular, even though level hashing provides an efficient resizing operation, neither of our global hash table structures require resizing, as their total size is known at file system creation time. This experiment shows the importance of file-system specific optimizations for PM file-mapping structures and this suggests that PM storage structures should not be directly used for file mapping.

5 Discussion

Generalizability of results. In our microbenchmarks, we report the performance of file mapping operations in isolation from the rest of the file system; these results are applicable to other PM file systems that use persistent mapping structures (e.g., ext4-DAX, SplitFS [24], NVFS [40], and ZoFS [13]).

Strata batches file mapping updates via the application log in LibFS (SplitFS has a similar batching system), which is measured in our macrobenchmark results. Batching amortizes the update overhead of mapping structures. For this reason, we predict that HashFS would outperform other mapping structures by a larger margin on PM file systems that do not batch updates (e.g., ext4-DAX, NFVS, and ZoFS).

Resilience. Resilience to crashes and data corruption is not a challenge exclusive to file mapping structures and reliability concerns are usually handled at a file-system level, rather than specifically for file-mapping structures. As we use a log for non-idempotent file-mapping structure operations (§3.1.1) and Strata logs other file mapping operations, all of our file mapping structures are equivalently crash-consistent. For resilience to data corruption and other device failures, our mapping structures can use existing approaches (e.g., TickTock replication from NOVA-Fortis [53]).

6 Related Work

There is little prior work that specifically analyzes the performance of file mapping structures. BetrFS [23] finds that write-optimized, global directory and file mapping structures are effective at optimizing write-heavy workloads. However, this analysis is performed on SSDs.

File mapping in PM file systems. PMFS [15] uses B-trees, allocating file data blocks in sizes of 4KB, 2MB, and 1GB memory pages. The PMFS allocator is therefore similar to an OS virtual memory allocator, albeit with different consistency and durability requirements. PMFS contrasts itself with systems that use extents for file mapping, but provides no justification for its scheme other than the fact that it transparently supports large pages [21]. We therefore do not know if its file mapping scheme is adequate for PM file systems. This problem extends to DevFS, which re-uses the metadata structures present in PMFS [27]. Strata and ext4-DAX both use extent trees for file mapping, with Strata using extent trees at all levels of its storage device hierarchy [11, 29]. Both of these systems use extent trees based on the legacy of ext4, providing no analysis if extent trees are optimal for PM.

PM-optimized storage structures. Much work has proposed PM optimized storage structures, both generic [6, 12, 31, 32, 37, 46, 47, 54, 58, 59] and within the context of database applications, such as key-value stores [26, 28, 51]. These provide in-place atomic updates whenever possible to avoid having to keep a separate log. However, common file system operations typically require atomic update of *multiple* file-system structures—e.g., when allocating blocks, the block bitmap must also be modified. Enforcing consistency and atomicity for a single data structure alone is therefore insufficient—we need to analyze file mapping structures within PM file systems to achieve efficient metadata consistency and durability.

Memory mapping. Mapping virtual to physical memory locations is similar to file mapping. A large body of research has improved virtual memory for decades [3, 44, 55] and has devised similar structures; page tables are radix trees on many platforms and recent work proposes cuckoo hashing as a more scalable alternative [41]. The key differences are in caching and consistency. File mapping caches are optimized for sequential access via cursors and SIMD prefetching; they are shared across all threads, simplifying frequent concurrent updates. MMUs optimize for random read access via translation lookaside buffers (TLBs) that are not shared across CPU cores, requiring expensive TLB shootdowns for concurrent updates. Additionally, since file-mapping structures are maintained in software rather than hardware, they allow for a wider variety of designs which may be difficult to efficiently implement in hardware (i.e., extent trees or HashFS's linear probing).

7 Conclusion

File mapping is now a significant part of the IO path overhead on PM file systems that can no longer be mitigated by a page cache. We designed four different PM-optimized mapping structures to explore the different challenges associated with file mapping on PM. Our analysis of these mapping structures shows that our PM-optimized hash table structure, HashFS, performs the best on average, providing up to 45% improvement on real application workloads.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Rob Johnson, for their valuable feedback. This work is supported by Applications Driving Architectures (ADA) Research Center (a JUMP Center co-sponsored by SRC and DARPA), the National Science Foundation under grants CNS-1900457 and DGE-1256260, the Texas Systems Research Consortium, the Institute for Information and Communications Technology Planning and Evaluation (IITP) under a grant funded by the Korea government (MSIT) (No. 2019-0-00118) and Samsung Electronics. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] H. Akinaga and H. Shima. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proceedings of the IEEE*, 98(12):2237–2251, Dec 2010.

[2] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer Torque Magnetic Random Access Memory

(STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 9(2):13:1–13:35, May 2013.

[3] Thomas W Barr, Alan L Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). *ACM SIGARCH Computer Architecture News*, 38(3):48–59, 2010.

[4] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. Emerging NVM: A survey on architectural integration and research challenges. *ACM Trans. Design Autom. Electr. Syst.*, 23(2):14:1–14:32, 2018.

[5] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The next generation of ext2/3 filesystem. In *LSF*, 2007.

[6] Shimin Chen and Qin Jin. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[7] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812. USENIX Association, July 2020.

[8] JG Clerry. Compact hash tables using bidirectional linear probing. *IEEE Transactions on Computers*, 100(9):828–834, 1984.

[9] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. File Systems Fated for Senescence? Nonsense, Says Science! In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 45–58, Santa Clara, CA, 2017. USENIX Association.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[11] Jonathan Corbet. Supporting filesystems in persistent memory, September 2014.

[12] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. Revisiting hash table design for phase change memory. *Operating Systems Review*, 49(2):18–26, 2015.

[13] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 478–493. ACM, 2019.

[14] Mingkai Dong, Qianqian Yu, Xiaozhou Zhou, Yang Hong, Haibo Chen, and Binyu Zang. Rethinking benchmarking for nvm-based file systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, page 20. ACM, 2016.

[15] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[16] Jose M. Faleiro and Daniel J. Abadi. Latch-free synchronization in database systems: Silver bullet or fool's gold? In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, page 9, 2017.

[17] Sanjay Ghemawat and Jeff Dean. Leveldb. http://leveldb.org, 2011.

[18] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Software wear management for persistent memories. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 45–63, Boston, MA, 2019. USENIX Association.

[19] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 759–771, 2017.

[20] Intel. Intel® Optane™ DC Persistent Memory. http://www.intel.com/optanedcpersistentmemory, 2019.

[21] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2019.

[22] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[23] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 301–315, Santa Clara, CA, 2015. USENIX Association.

[24] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508. ACM, 2019.

[25] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, pages 691–704, 2018.

[26] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. SLM-DB: single-level key-value store with persistent memory. In Merchant and Weatherspoon [34], pages 191–205.

[27] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a true direct-access file system with devfs. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 241–256, 2018.

[28] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX-ATC 18)*, pages 993–1005, 2018.

[29] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. ACM.

[30] E. Lee, H. Bahn, S. Yoo, and S. H. Noh. Empirical study of nvm storage: An operating system's perspective and implications. In *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, pages 405–410, Sep. 2014.

[31] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, 2017.

[32] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Ontario, Canada, October 2019.

[33] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[34] Arif Merchant and Hakim Weatherspoon, editors. *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. USENIX Association, 2019.

[35] Micron. Battery-backed nvdimms, 2017.

[36] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, 2019.

[37] Moohyeon Nam, Hokeun Cha, Youngri Choi, Sam H Noh, and Beomseok Nam. Write-Optimized and dynamic-hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[38] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[39] Waleed Reda, Henry N. Schuh, Jongyul Kim, Youngjin Kwon, Marco Canini, Dejan Kostić, Simon Peter, Emmett Witchel, and Thomas Anderson. Assise: Performance and availability via NVM colocation in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Banff, Alberta, November 2020. USENIX Association.

[40] RedHat. NFVS documentation. https://people.redhat.com/~mpatocka/nvfs/INTERNALS, 2020.

[41] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1093–1108, 2020.

[42] Keith A Smith and Margo I Seltzer. File system aging—increasing the relevance of file system benchmarks. In *ACM SIGMETRICS Performance Evaluation Review*, volume 25, pages 203–213. ACM, 1997.

[43] Steven Swanson. Early measurements of intel's 3dx-point persistent memory dimms, Apr 2019.

[44] M. Talluri, M. D. Hill, and Y. A. Khalidi. A new page table for 64-bit address spaces. In *Proceedings of the*

Fifteenth ACM Symposium on Operating Systems Principles, SOSP '95, page 184–200, New York, NY, USA, 1995. Association for Computing Machinery.

[45] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41, 2016.

[46] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, pages 5–5. USENIX Association, February 2011.

[47] Chundong Wang, Qingsong Wei, Lingkun Wu, Sibo Wang, Cheng Chen, Xiaokui Xiao, Jun Yang, Mingdi Xue, and Yechao Yang. Persisting rb-tree into NVM in a consistency perspective. *TOS*, 14(1):6:1–6:27, 2018.

[48] Ying Wang, Dejun Jiang, and Jin Xiong. Caching or not: Rethinking virtual file system for non-volatile main memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[49] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 ACM International Conference on Management of Data*, SIGMOD '18, pages 473–488, 2018.

[50] 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.

[51] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 349–362, 2017.

[52] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[53] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 478–496, 2017.

[54] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.

[55] Idan Yaniv and Dan Tsafrir. Hash, don't cache (the page table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, page 337–350, New York, NY, USA, 2016. Association for Computing Machinery.

[56] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A tiered file system for non-volatile main memories and disks. In Merchant and Weatherspoon [34], pages 207–219.

[57] Shengan Zheng, Hao Liu, Linpeng Huang, Yanyan Shen, and Yanmin Zhu. HMVFS: A versioning file system on DRAM/NVM hybrid memory. *J. Parallel Distrib. Comput.*, 120:355–368, 2018.

[58] Jie Zhou, Yanyan Shen, Sumin Li, and Linpeng Huang. NVHT: An efficient key-value storage library for non-volatile memory. In *Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, pages 227–236. ACM, 2016.

[59] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.

pFSCK: Accelerating File System Checking and Repair for Modern Storage

David Domingo, Sudarsun Kannan
Rutgers University

Abstract

We propose and design pFSCK, a parallel file system checking and recovery (C/R) tool designed to exploit compute and storage parallelism in modern storage devices. pFSCK enables fine-grained parallelism at the granularity of inodes and directory blocks without impacting the C/R's correctness. pFSCK first employs data parallelism by identifying functional operations in each stage of the checking logic and then isolating dependent operations and shared data structures. However, full isolation of shared structures is infeasible and requires serialized updates. To reduce serialization bottlenecks, pFSCK introduces pipeline parallelism, allowing multiple stages of C/R to run concurrently without impacting correctness. Further, pFSCK provides per-thread I/O cache management, dynamic thread placement across C/R stages, and a resource-aware scheduler to reduce the impact of C/R on other applications sharing CPUs and the file system. Evaluation of pFSCK shows more than 2.6x gains over e2fsck (Ext file system C/R) and more than 1.8x over XFS's C/R that provides coarse-grained parallelism.

1 Introduction

Modern ultra-fast storage devices such as SSDs, NVMe, and byte-addressable NVM storage technologies offer higher bandwidth capabilities and lower latency than hard-disks providing better opportunities for exploiting CPU parallelism. While the I/O access performance has increased, storage hardware and software errors have continued to grow coupled with newer and exploratory high-performance designs impacting file system reliability [10, 12, 18, 21, 27, 46]. For decades, file system checking and repair tools (referred to as C/R henceforth) have played a pivotal role in increasing the reliability of software storage and availability of systems by identifying and correcting file system inconsistencies [41].

A significant body of prior work has shown that, in the event of a system crash or failure in data centers, C/Rs are typically used as the first remedial solution to system recovery. Prior work [21, 27] and discussions with file system maintainers and IT teams of organizations show that C/Rs are run across various scenarios. This includes problems during reboot due to hardware or software errors [11, 21, 27], periodic maintenance, or during mandatory security upgrades [37]. When C/Rs are run on a disk partition in an offline fashion, the partition's data is unavailable. Some C/Rs support online checking, but it is crucial that they do not interfere with other applications that use the same device. Thus, improving C/R performance and flexibility is critical for system availability and reducing performance impact on other applications.

File system C/R tools work by identifying and fixing the structural inconsistencies of file system metadata. The inconsistencies could be in inodes, data and inode bitmaps, links, or directory entry structures. Well-known and widely used tools such as e2fsck (file system checker for Ext4) [2] divide C/R across multiple stages (commonly referred to as passes), with each pass responsible for checking a file system structure (e.g., directories, files, links). However, C/Rs are known to be notoriously slow, showing a linear increase in C/R time with an increase in file and directory count [24, 38–41], at times lasting hours [37] or even weeks [11]. Although modern flash and NVM technologies provide lower latency and bandwidth, current C/R tools fail to exploit such hardware capabilities or multi-core CPU parallelism. While modern C/Rs have attempted to increase parallelism, they adopt coarse-grained approaches, such as parallelizing C/R across logical volumes or logical groups, which are insufficient to accelerate C/R on file systems with data imbalance across logical groups [20, 24, 39, 42].

To overcome such limitations, we propose **pFSCK**, a parallel C/R that exploits CPU parallelism and modern storage's high bandwidth to accelerate file system C/R in offline and online forms, thereby reducing system downtime and improving data (and system) reliability and availability [10, 20, 21, 39]. While pFSCK borrows ideas from prior task parallelism research [35, 45], it must solve several challenges specific to C/R, which includes increasing scalability in the presence of complex file system layouts and shared file system structures (e.g. universal bitmaps) without impacting correctness, adapting to various file system configurations, and reducing C/R impact on other applications. pFSCK introduces fine-grained parallelism, i.e., parallelism at the granularity of inodes and directory blocks, resulting in a significantly faster execution than traditional C/Rs. pFSCK first employs **data parallelism** by breaking up the work done at each pass, redesigning data structures for scalability, and allowing multiple threads to perform checks in parallel. Although data parallelism accelerates checking, updates to global data structures (e.g., bitmaps) within each pass are designed to match the file system's layout (e.g., block bitmap in an Ext4 file system) and must be synchronized to ensure checking correctness. As a result, with increasing thread counts, the cost of synchronization and serialization can quickly outweigh the performance gains. Hence, pFSCK introduces **pipeline parallelism** to parallelize C/R along with the logical flow (i.e. across multiple passes).

Supporting data and pipeline parallelism within pFSCK requires addressing several challenges. First, certain consistency checks must be ordered for correctness. For example, a directory cannot be certified to be error-free by the directory checking pass until all its files are verified as consistent by the inode checking pass. To address these ordering constraints, we take inspiration from modern hardware processors that support out-of-order execution but with in-order instruction commit. We isolate the global data structures and perform all necessary operations in parallel but certify correctness only when the results are merged. Second, static partitioning of CPU threads across different C/R passes is suboptimal because the time to process different metadata (e.g., file, directory, links) varies significantly (e.g., checking a directory can take substantially longer than a file). Hence, we propose **pFSCK scheduler**, a dynamic thread scheduler that monitors progress across different passes of pFSCK and uses the pending work ratio for thread assignment.

Third, I/O optimizations such as I/O caching and read-ahead mechanisms in current C/Rs are not designed for multi-threaded parallelism, which we address by designing thread-aware I/O caching, thereby substantially reducing I/O wait-times. Finally, to exploit multi-core parallelism in ways that do not affect the performance of other co-running applications that share CPUs or access the same disks checked by C/R (online checking), we design a **resource-aware pFSCK scheduler** that dynamically scales the C/R threads across passes by monitoring the total CPU utilization of the system.

The combination of pFSCK's above techniques significantly reduces C/R runtime. For example, using pFSCK's data parallelism and pipeline parallelism on a 1TB NVMe (and 2TB SSD) reduces C/R runtime for a file- and directory-intensive disk configurations by up to 2.6x and 1.6x, respectively compared to widely used e2fsck and by up to 1.8x over the XFS C/R tool. Further, pFSCK's scheduler increases gains by 1.1x. When sharing the CPUs between pFSCK and RocksDB, the resource-aware mechanism minimizes pFSCK performance degradation to 1.07x and limits RocksDB's performance overheads by 1.05x. The online C/R performance improves by up to 1.7x over the vanilla e2fsck. Finally, pFSCK improves I/O throughput by up to 2.7x with a nominal increase in memory use by 1.3x over e2fsck (from 2.7 GB in e2fsck to 3.5 GB) to manage task structures for threads.

2 Background and Motivation

We first give a brief background on current hardware trends, C/R tools, and then discuss prior approaches that accelerate C/R and their limitations.

2.1 Hardware and Software Trends

Modern ultra-fast storage devices such as SSDs and NVMe provide not only high bandwidth (8-16 GB/s) but also two orders of reduction in storage access latency (< 20µsec) compared to traditional hard drives [31, 49]. At the other end, fast storage class memories such as Intel's DC Optane [5] and other byte-addressable persistent memory technologies are evolving with access latency < 1µsec. In recent years, several new file systems have evolved to exploit these hardware benefits. A huge body of prior and ongoing research is developing optimized file systems to support fast storage hardware. This includes file systems for SSDs [34], NVMes [44], open-source efforts to optimize traditional Ext4 and XFS file systems for NVMs [48], and other research efforts [30, 33, 50]. However, reducing data corruption and errors with these file systems would require a few years of production use [9, 28]. While file system C/R tools will play a crucial role in these file systems, they are yet to be optimized to extract hardware storage benefits and multi-core parallelism.

2.2 File System Checking and Repair

Since the dawn of file systems, consistency has always been an issue. Though storage mechanisms such as journaling, copy-on-write, log-structured writes, and soft updates have been developed to mitigate potential file system inconsistencies, they are limited as they cannot fix errors that arise from software bugs or corruptions manifested in the past by events such as a failing disk, bit flips, overheating, or correlated crashes [13–15, 29, 51]. In these cases, popular C/R tools, such as e2fsck and xfs_repair [42], are used to detect and fix corruptions and errors by traversing the file system's layout and checking for inode consistency, directory consistency, file and directory connectivity, directory entry consistency, and consistent reference counts of inodes and blocks

C/R usage. The frequency and resulting runtime of file system C/Rs vary significantly in the real-world setting. While there is a lack of well-documented C/R best practices, our discussions with file system C/R maintainers, infrastructure teams, and other public discussions show that C/R tools such as e2fsck and xfs_repair remain critical for data reliability in current large-scale and personal computing systems as they are generally run during after system errors [7, 21, 27, 29], hardware or kernel upgrades, or even after mandatory security updates. Infrequent C/R and storage maintenance can increase system downtime to as high as three hours [37] and, in extreme cases, weeks on petabyte-scale file systems [11].

2.3 Related Work

Increasing disk capacities, overall file system size, and file counts have made C/Rs notoriously run longer, leading to longer downtime and forcing developers and users to reduce C/R usage at the risk of data loss [1, 7, 8, 25, 36]. We next discuss the state-of-the-art C/R optimizations for offline (unmounted file systems) and online C/Rs and their limitations.

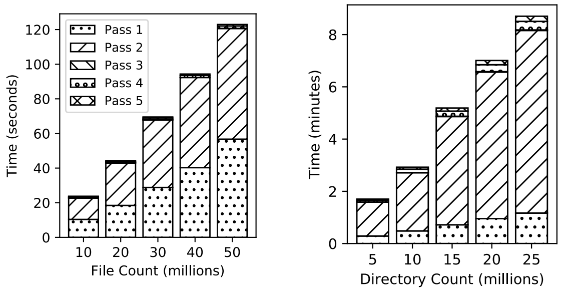
Offline C/Rs. Widely used open-source tools such as Ext4 file system's e2fsck parallelize C/R across multiple disks (e2fsck), where XFS file system's xfs_repair parallelizes C/R across disk and volume logical groups. Other approaches like Ffsck [41] and Chunkfs [26] have proposed accelerating C/R speed up by modifying file systems to provide a better balance across logical groups. For example, Chunkfs utilizes disk bandwidth by partitioning the file system into smaller, isolated groups that can be repaired individually and in parallel. In contrast, Ffsck [41] rearranges metadata blocks to reduce the seek cost and optimize file system traversal. SQCK [19] enhances C/R by utilizing declarative queries for consistency checking across file system structures. While prior approaches have advanced C/R innovations, they suffer from several weaknesses. First, most prior techniques fail to exploit multi-core parallelism and high storage bandwidths. Second, prior parallelization efforts are mostly coarse-grained (e2fsck, xfs_repair, Chunkfs). For instance, as we show in Section § 6, for an XFS file system that parallelizes across logical groups, imbalance in the number of files across logical groups and lack of parallelism for directory metadata checking leads to high overheads. Finally, techniques such as SQCK and Ffsck demand intrusive changes to file system metadata, block placement, or the need to rebuild C/R, hindering widespread adoption.

Online C/Rs. The last decade has seen an active push to develop online C/R techniques to identify and fix errors while applications concurrently use the file system in order to reduce system downtime and allow for the proactive identification of potentially harmful corruptions. Proprietary online C/Rs such as WAFL file system's Iron [32] (a NetApp-based C/R tool for WAFL file system) performs incremental live C/R by applying invariants such as checking all blocks before any software use and checking ancestor blocks (directory) before any data or metadata block (inode block). To scale C/R to petabytes, WAFL-Iron expects the presence of block-level checksums, RAID, and, most importantly, good storage practices by customers. Alternatively, Recon protects file system metadata from buggy operations by verifying metadata consistency at runtime [16]. Doing so allows Recon to detect metadata corruption before committing it to disk, preventing error propagation. Recon does not perform a global scan and cannot identify or fix errors originating from hardware failures. C/Rs such as e2fsck, traditionally meant for offline use, allow for partial online checking by utilizing LVM-based snapshotting and scanning for errors on the snapshot while the file system is still in use [3]. However, if errors need to be fixed, the C/R must be used offline. In this paper, we also study and evaluate open-source and widely-used online e2fsck against online pFSCK that exploit storage and compute parallelism.

C/R Correctness. To ensure the correctness and crash-consistency of C/Rs itself and recover more reliably in light of system faults, Rfsck-lib [17] provides C/Rs with robust undo logging. pFSCK's fine-grained parallelism goals are orthogonal to Rfsck-lib; however, incorporating Rfsck-lib can further improve pFSCK's reliability.

3 Motivation and Analysis

In the pursuit of accelerating C/Rs, we first decipher the performance bottlenecks of the widely-used Ext4 file system's e2fsck C/R tool. We first provide an overview of e2fsck and then examine e2fsck's runtime for different file system configurations. For brevity, we study xfs_repair in Section § 6.



(a) File Count Sensitivity: Runtime of e2fsck as total file count increases

(b) Directory Count Sensitivity: Runtime of e2fsck as total directory count increases

Figure 1: Runtime of C/R for an 1TB file system with varying counts of files or directories

3.1 e2fsck Overview

E2fsck uses five sequential passes for C/R: the first pass (referred to as Pass-1) checks the consistency of inode metadata; Pass-2 checks directory consistency; Pass-3 checks directory connectivity; Pass-4 checks reference counts; finally, Pass-5 checks data and metadata bitmap consistency.

3.2 Setup and Runtime Analysis

To analyze and decipher the cost of C/R runtime, we run e2fsck on file systems with varying configurations. We conduct our analysis on a 64-core Dual Intel® Xeon Gold 5218 running at 2.30GHz, 64GB of DDR memory, a 1TB NVMe, and a 2TB Micron 5200 SATA SSD running Ubuntu 18.04.1. We fill the file system using *fs_mark*, an open-source, file system benchmark tool [47]. We mainly focus on file systems without corruptions but study images with corruptions in Section § 6. To get a finer understanding of how e2fsck scales with file system configurations, we study the sensitivity of C/R's runtime for multiple file system variables such as file count and directory count.

File-intensive file systems. First, to understand how the file count affects runtime, we generate multiple file-intensive file system configurations with a 95 : 1 file to directory ratio. Pass-1, which checks the consistency of inodes structures, dominates e2fsck runtime, followed by Pass-2, which checks directory block consistency. Figure 2 shows the function-wise breakdown in Pass-1 that checks the consistency of file inodes as well as tracks directory blocks encountered to be examined in the next pass. We notice a function `dcigettext` (a seemingly innocuous) language translator used for error handling gets (incorrectly) used for every inode check, causing notable C/R slowdown. [1] Other steps such as `check_blocks` that checks blocks referenced by an inode, `next_inode` that reads next inode blocks from disk, `mark_bitmap` that updates global bitmaps to track the metadata encountered, and `icount store` that stores inode references also increase in runtime. Despite fewer directories, Pass-2's (directory checking) runtime increases because the number of directory blocks to

[1] We reported this to e2fsck developers, and the fix has been upstreamed.

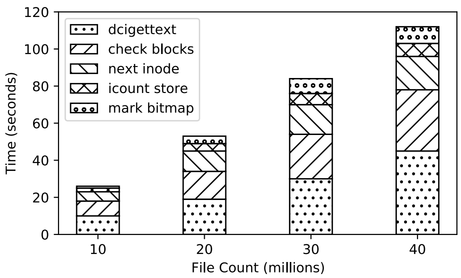


Figure 2: e2fsck Pass-1 Time Breakdown. Time spent within inode checking pass (Pass-1) as the total file count increases.

store directory entries increases. For all file counts, time spent in Pass-1 and Pass-2 account for over 95% of the runtime. Finally, for a small directory count, Pass-3, which checks connectivity and reachability of directories from the root, has lower runtime. *Increasing the file size when keeping file count constant does not increase C/R runtime significantly* (not shown for brevity).

Directory-intensive file system. In Figure 1b, we decipher the runtime of a directory-heavy file system configuration with 1 : 1 file to directory ratio. To ensure that each directory requires the same amount of work, we create a single file in each directory. First, the cost of fetching, identifying directory blocks, and adding them to a global block list (*db_list*) increases Pass-1's cost along with reference counting. As expected, with an increase in directory count, Pass-2's runtime significantly increases due to an increase in the number of directory blocks. Additionally, the directory blocks store checksums, and the checksum is recomputed and verified against the one stored in the directory blocks for consistency. Similar to the file-intensive file systems, for all directory counts, over 90% of the runtime is spent within Pass-1 and Pass-2.

3.3 Compute time and I/O utilization

To understand the computational vs. I/O bottlenecks, we analyzed the time spent by e2fsck on compute time vs. the time spent waiting for I/O to complete. Our analysis shows that in modern storage devices like NVMe, the I/O wait time for file-intensive and directory-intensive configurations are only 3% and 20% of the overall execution time. We also notice poor storage throughput utilization, which is just 270 MB/s on an NVMe device with 2 GB/s and 512 MB/s sequential and random read bandwidth, respectively. This clearly shows that (1) computation is the main bottleneck as it dominates overall execution time, and (2) C/Rs such as e2fsck fail to benefit from modern storage device bandwidths.

Summary. To summarize, our analysis of widely-used C/R tools such as e2fsck (and xfs_repair later in § 6) show high runtime overheads mainly due to single-threaded or lack of fine-grained parallelism. The linear complexity of C/R runtime is unsuitable as disk capacities and file system size trends upward, potentially taking hours, or even days, to check datacenter-scale file systems. Besides, C/R's repair during file system inconsistencies could further increase C/R runtime.

4 pFSCK Goals and Insights

pFSCK aims to address the limitations of current file system C/Rs by exploiting fine-grained multi-core parallelism and higher disk bandwidth. We first discuss our goals and insights, followed by pFSCK's design and implementation.

4.1 Goals

The main goal is to make the file system C/R faster. We want to increase the speed at which file system metadata is scanned and inconsistencies are identified without compromising repairing capabilities. In this pursuit, pFSCK strives to achieve the following goals: (1) adapt to different file system configurations, regardless of file system size, utilization, or configurations, such as a file-intensive or directory-intensive file system, (2) support efficient C/R for both online and offline forms, and finally, (3) adapt to varying system resource utilization to reduce the performance impact on any concurrently running applications.

4.2 Design Insights

Insight 1: Maximize potential bandwidth through multiple cores and data parallelism. To overcome the bottlenecks of current C/R tools that employ serial or coarse-grained parallelization techniques at the disk, volume, or logical group-level, pFSCK introduces fine-grained data parallelism. As shown in Section § 3.2, since Pass-1 and Pass-2 account for over 90% of the runtime for both file- and directory-intensive file systems, pFSCK focusses its efforts on these two passes. Our approach divides finer file system structures such as inodes, directory blocks, and dirents across a pool of worker threads in a single pass that performs C/R concurrently. While seemingly simple, achieving data parallelism requires data structure isolation across threads to reduce synchronization bottlenecks.

Insight 2: Enable pipeline parallelism by reducing inter-pass dependencies. Though data parallelism accelerates C/R, each pass (e.g., directory checking) must wait for the previous pass (e.g., inode check) to complete. Specifically, in C/R, several inter-pass global data structures are used to build a consistent view of the file system and identify inconsistencies (ex. bitmaps). As a consequence, updates to the shared global structures must be serialized, thereby increasing contention to shared structures with increasing thread count and limiting the CPU scalability. To reduce serialization overheads, pFSCK designs pipeline parallelism that breaks the rigid wall across passes allowing multiple passes to be executed concurrently. pFSCK manages per-pass thread pools, isolates inter-pass shared structures using divide and merge approaches, delineates checking from actual certification of an inode, and reduces I/O wait times.

Insight 3: Adapt to file system configurations with dynamic thread scheduling. Both data and pipeline parallelism requires assigning threads across different passes. Static or equal partitioning of CPU threads is suboptimal due to a lack of information about metadata types (files, directories, links) and work across passes. Simple checks such as information about the number of files vs. directory inodes are insufficient because directory processing is complex and time-consuming (see § 3). To overcome the above challenge, we design a C/R thread scheduler that dynamically assigns and migrates threads across passes to process different file system objects as they are discovered.

Insight 4: Reduce system impact through resource utilization awareness. File system C/Rs could potentially run with other applications sharing CPUs while performing checking on separate disks. Given pFSCK's goal is to exploit available CPUs, it could potentially impact other co-running applications. Similarly, C/R could run on disks that are also actively used by other applications to store data. To reduce the overall system impact on co-running applications as well as pFSCK, we equip pFSCK's scheduler with resource awareness to dynamically identify the number of cores to use at any single point in time to minimize the potential impact on other co-running applications and pFSCK's performance.

5 Design and Implementation

We discuss pFSCK's design and implementation of data parallelism, pipeline parallelism, dynamic thread scheduler, and resource-aware scheduling. We aim to extend the widely used e2fsck without requiring file system layout changes and reuse features such as snapshot-based online C/R.

5.1 Data Parallelism

pFSCK's data parallelism aims to divide work across worker threads in each pass at the granularity of inodes for parallelizing C/R. However, pFSCK must ensure efficient parallelism without compromising file system integrity or correctness through a functional separation of C/R passes and per-thread contexts that isolate global data structures for reducing synchronization cost.

Fine-grained Inode-level Parallelism. For fine-grained inode parallelism, pFSCK uses the superblock information to identify the total number of inodes in a file system and evenly divides the inodes across C/R workers. To reduce worker management costs, we use a thread-pool framework from our prior work to assign tasks across workers [31].

Functional Parallelism for Reducing Synchronization Overheads. Only dividing inodes for C/R across workers is insufficient. To benefit from fine-grained parallelism, reducing synchronization cost across workers in each pass without impacting correctness is critical.

pFSCK breaks each C/R pass into four main functional steps that comprise 95% of the work and reduces synchronization across these steps. These steps include (1) file system metadata checks, (2) global file system metadata update, (3) accounting, and finally, (4) intermediate result sharing. The metadata check verifies the integrity of metadata structures (for example, inode checksums and block references). Next, the global file system metadata updates include changes to file system-level bitmaps that maintain the checker's view of the file system in order to detect inconsistencies. For example, the block bitmap as shown in Table 1 is marked to track which block references have been seen to detect duplicate block references where more than one inode claims the same block(s). Third, C/R-level accounting involves updating counters that track statistics such as file types. Finally, intermediate result sharing involves creating data structures and lists that hold information to be processed by the next pass. Such structures include directory info lists and directory block lists that store directory information and the location of their blocks so it can be checked in the directories pass.

Structure	Role	pFSCK's access
dir_info_list	maintains directory information/relationships	splits into per-thread lists
db_list	maintains directory blocks for Pass-2 to check	splits into per-thread lists and merges
inode_used_bitmap	tracks valid inodes	using locks
inode_dir_bitmap	tracks directory inodes	using locks
inode_reg_bitmap	tracks regular file inodes	using locks
block_found_bitmap	tracks used data blocks	using locks
icount	tracks inode reference counts	using locks

Table 1: Global structures, their role, and access method.

While synchronization between file system metadata checks (step 1) and global metadata update steps (step 2) is essential, synchronization between the first two steps and the last two steps, C/R counter/statistics update (step 3) intermediate result sharing (step 4) can be avoided by allowing threads to maintain per-thread stats and generate data structures in isolation. The results of steps 3 and 4 can be aggregated before the next pass, reducing synchronization costs significantly.

Thread Contexts for Isolation. In current C/Rs such as e2fsck (and xfs_repair), we find significant use of global data structures inside and across passes. To reduce sharing and increase concurrency, we introduce per-thread contexts similar to OS thread contexts. These contexts store information that allow threads to operate in parallel. First, per-thread block caches, buffers (heap allocations), and iterators of file system objects allow for parallel file system traversal. Second, per-thread intermediate data structures and counters are used for gathering high-level file system state in parallel. For example, each inode pass (Pass-1) thread has its own `db_list` (directory block list) and `dir_info` list (directory information list) that gathers information about directories and exports these to the directory pass (Pass-2) threads for processing. Lastly, per-thread counters are used to track file type statistics in parallel. However, we observe that due to frequent access of global bitmaps across passes (for almost every operation), pFSCK is forced to use synchronization through locking. While disaggregating these bitmaps into per-thread structures is feasible, it would demand significant changes to the e2fsck framework. Table 1 shows the shared structures and their role in e2fsck.

Thread Colocation for Improved Locality. To increase locality and better utilize processor cache state, in each pass, pFSCK attempts to co-locate threads within the same pass to

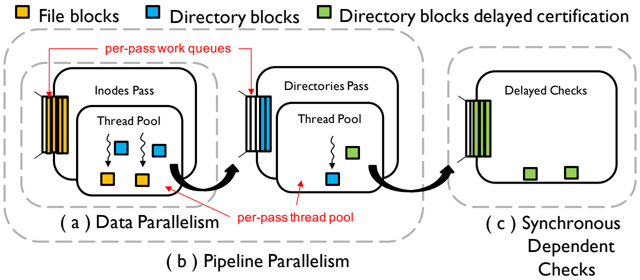


Figure 3: Parallelism in pFSCK. (a) Threads within each pass allows for data to be operate in parallel (*data parallelism*). (b) Multiple thread pools allows each pass of pFSCK to operate simultanously (*pipeline parallelism*). (c) Any dependent checks needed to be carried out synchronously is delayed within its own logical pass.

the same cores and memory sockets to avoid bouncing lock variables and shared structures across processor caches. To enable thread colocation, pFSCK maintains the CPU number each thread has used and a list of cores used by a particular pass. pFSCK first attempts to place the thread to the previously used core (if available), and if unavailable, uses other available cores which were used in the same pass.

Reducing I/O Wait Time with Per-thread Prefetchers and Cache. Though current C/Rs such as e2fsck cache and prefetch file system blocks, the caching and prefetching mechanisms are inflexible and lack thread awareness. First, e2fsck uses a small, static fully-associative LRU-based cache (with 8 blocks) and prefetches just inode blocks. E2fsck does not prefetch directory data blocks, which could be non-contiguous, unlike inode blocks. Second, because threads access blocks at different offsets, sharing a cache across threads results in conflicting evictions, increasing I/O overheads.

To overcome such limitations, we design and implement a per-thread caching mechanism to avoid false eviction of cache entries across threads. For avoiding the non-contiguity problem of directory blocks, we implement an adaptive prefetching mechanism (similar to Linux filesystem prefetching) – decrease prefetching window if previously prefetched directory blocks are not used due to lack of sequential access.

5.2 Pipeline Parallelism

While data parallelism achieves concurrency for processing file system objects within a pass, fully isolating per-pass shared data structures and global data structures is not feasible without substantial changes to either the file system layout or the C/R. As a result, data parallelism does not fully benefit from increasing the CPU count. As our results show, the benefits can considerably degrade performance at higher core counts due to increasing synchronization overheads.

To reduce synchronization time and increase CPU effectiveness, pipeline parallelism breaks the limitation that C/R passes must be sequentially executed. pFSCK's pipeline parallelism allows a subsequent C/R pass ($Pass_{i+1}$) to start even before the completion of an earlier pass ($Pass_i$) in a **pipelined** fashion (i.e., checking directories in directory checking pass even before the inode checking pass has completed).

5.2.1 Per-Pass Thread Pools and Work Queues.

First, to facilitate concurrent execution of passes, we use *per-pass thread pools*. As shown in Figure 3, the inode and directory checking passes maintain a separate thread pool and a dedicated work queue filled with file system objects needing to be checked. As each pass operates, any intermediate work generated is placed in the next pass's work queue. For example, the inode checking pass, when encountering an inode representing a directory, queues its blocks to the directory checking pass's work queue to enable concurrent C/R.

5.2.2 Delayed Certification for Concurrency.

Allowing multiple passes to run in parallel using pipeline parallelism requires reordering logical checks for correctness. For example, with pFSCK's pipeline parallelism, the directory data blocks can be checked by the directory checking pass (Pass-2) in parallel with inode checking pass (Pass-1) checking all the inodes (files and subdirectories) in the directory. While the two passes can proceed in parallel, a directory can be marked as consistent only after the inode checking pass verifies the consistency of its subdirectories and files. There are two main constraints for certifying a directory by the C/R: (1) all inodes referenced by the dirents of this directory are valid, and (2) the parent directory referenced by this directory is valid.

Providing Ordering Guarantee. To address the challenge of ordering guarantee, pFSCK delays certain checks until the prior pipeline pass is complete. For example, the inode checking pass within the pipeline is responsible for creating directory structures used in the directory C/R pass. The directory pass examines subdirectories and checks whether the subdirectory's parent (represented by double dot ..) maps back to the directory. However, because the inode and directory checking passes run in parallel, not all the inodes of the subdirectories would have been checked when the parent directories are checked. For handling the scenarios above, pFSCK delays certification by encapsulating the relationship needing to be verified into task structures that are added to a separate work queue. This task queue is then processed only after all inodes have been checked (e.g., after the inode checking pass completes) as shown in Figure 3. Delayed certifications are infrequent in file-intensive configurations and frequent in the directory-intensive configuration. pFSCK's delayed certification increases concurrency between Pass-1 and Pass-2, consequently improving performance.

To summarize, combining pipeline with data parallelism reduces I/O wait time and improves pFSCK's performance across different file system configurations, as our results show in Section § 6.

5.3 Dynamic Thread Scheduler

C/R runtime can vary significantly depending on the configuration of the file system. For example, C/R on a file system with a larger ratio of smaller files could result in a substantially longer runtime compared to a file system with few-but-larger files due to more metadata needing to be checked. Similarly, heterogeneity in terms of inode types (files, directories, links) can impact runtime, and the exact configuration remains unknown until the inodes are iterated over in the inode checking pass (Pass-1). Additionally, each pass within C/R has differing degrees of access to shared structures. Therefore, statically assigning threads across each pass could be ineffective. Hence, to adapt to file system configurations, pFSCK implements a C/R-aware scheduler, **pFSCK-sched**, supported by extending the thread pools to allow for migration of threads between the passes. Also, pFSCK-sched maintains an idle thread pool to hold any threads not scheduled to run for any of the passes.

Thread Assignment and Migration of Worker Threads. In pFSCK, we enable dynamic assignment of threads across each pass by implementing a scheduler that actively monitors progress and migrates threads across the passes. The scheduler periodically scans through the work queues of each pass to identify the work distribution ratio across the pipelined passes and uses this ratio to assign threads across them.

Figure 4 shows an example of pFSCK-sched across the first two passes. Initially, all the CPU threads are assigned to the first pass (inode checking pass) given that pFSCK only knows total inodes from the file system superblock and not the types of inodes. When an inode C/R thread identifies a group of directory inodes, it places the directory inodes and their corresponding directory blocks to the work queue of directory C/R pass. If no threads are present in the thread pool used for the directory pass, threads from the inode pass are migrated to the directory pass. To calculate the number of threads to be reassigned, a dedicated scheduler thread finds the total work to be done across all passes using the following model.

Let W_{total} be the amount of work needing to be done. Let q_i be the length of the work queue for pass i. Let n_i be the number of discrete elements needing to be processed for each entry in the work queue. Let w_i be some weight that normalizes the work to be done for each element in pass i. Let C be the core budget and t_i be the number of threads to assign for pass i.

$$W_{total} = \sum_{i=0}^{N} q_i n_i w_i \tag{1}$$

$$t_i = C \cdot q_i n_i w_i \cdot \frac{1}{W_{total}} \tag{2}$$

As shown in Equation (1), the total work to be done is a summation of outstanding work across each pass, which is a product of the work queue length (q_i), the number of objects encapsulated within each queue entry (n_i), and a normalizing weight (w_i). As shown in Equation (2), with the total amount of work needing to be done, the scheduler can determine the ideal number of threads to assign to a pass (t_i) based on the total core budget (C) and the relative amount of work calculated for each pass. Note that the normalizing weights are essential for accounting for the differences in the time to process different file types (directories vs. inodes). Our analysis of different file system configurations (discussed in § 3) shows that in pFSCK (and in e2fsck), the average CPU cycles spent on processing one directory is 1.8x - 2.3x higher than processing an inode, mainly due to directory checksum calculations. Overall, we find it is beneficial to use higher weights for directory checking queues.

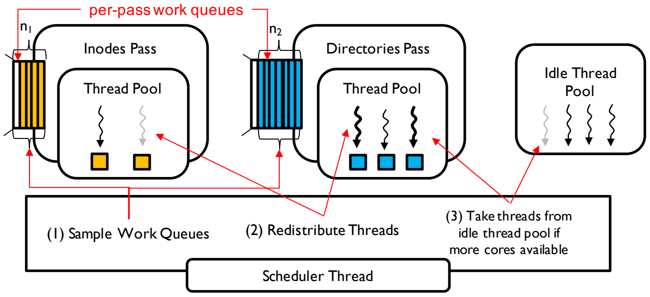


Figure 4: Dynamic Thread Scheduling. A dedicated scheduler thread periodically samples work queues among all the passes and redistributes threads based on the proportion of outstanding work.

5.4 System Resource-Aware Scheduling

File system C/Rs could potentially coexist or even share CPUs with other applications using the same or another file system (or disk). In the pursuit of exploiting parallelism, pFSCK must reduce the impact on other applications. To address this goal, we introduce **pFSCK-rsched**, a system resource-aware pFSCK scheduler.

5.4.1 Efficient CPU Sharing

First, we discuss a case where the C/R runs alongside other applications but performs C/R on a separate, unmounted disk. To reduce the impact of C/R overheads on other applications, pFSCK-rsched maintains a scheduler thread. Initially, the pFSCK-rsched workers are scheduled with SCHED_IDLE priority that mostly schedules a process on any idle CPUs [6]. As the scheduler periodically runs, pFSCK-rsched first determines a CPU core budget to identify the maximum number of threads it could use at any point in time by identifying the number of CPUs in active use across the system, the number of idle cores available, and the number of cores used by pFSCK-rsched. Based on the effective number of cores being used by pFSCK-rsched, pFSCK-rsched increases pFSCK's core budget if it was utilizing less than the available idle cores or shrinks pFSCK's core budget if pFSCK uses more than the idle cores, reducing contention with other applications. After determining the core budget, the scheduler identifies the work ratio across the passes using the per-pass work queues and redistributes an ideal number of threads across each pass. When adding threads to a pass, threads are taken from the idle thread pool and assigned to the per-pass thread pool. If threads need to be removed due to a decrease in the core budget, threads are signaled and reassigned to the idle thread pool. In § 6, we discuss the performance benefits and implications of pFSCK-rsched when co-running and sharing CPUs with another application (RocksDB).

5.4.2 Efficient CPU and File System Sharing

Given the renewed focus for supporting online C/R [32], C/R tools like e2fsck, originally intended for offline use, can be used online with the help of Linux's Logical Volume Manager (LVM). LVM's snapshot feature captures file system state by employing a copy-on-write approach to preserve the original version of modified blocks. [23]. This enables C/R tools to be used in a proactive manner, scanning for pre-existing errors without having to bring the system down.

Towards online C/R with LVM, an empty snapshot volume is first initialized. If any blocks are modified by another application, LVM copies the original blocks to the snapshot before updating the blocks in place on the original volume. When C/R reads blocks that are found to have been modified, the reads are redirected to the snapshot which holds the original blocks. Reads of unmodified blocks are redirected to the original volume. While snapshot initialization is inexpensive, applications incur extra overhead of synchronous data copying and I/O redirection reducing the available storage bandwidth for C/R. This is especially the case when file system blocks are being frequently modified the other application.

In the case of pFSCK, fine-grained parallelism accelerates online C/R even when applications share the same file system (and disks). Futher, pFSCK's resource awareness reduces the impact on co-running applications by reducing CPU (and I/O contention), allowing the application to run faster. We further discuss the benefits of pFSCK for online C/R in § 6.

5.5 Verifying Correctness and Optimizations

Correctness. To ensure the correctness of the C/R, pFSCK with fine-grained parallelism employs a series of steps. First, although the checks are done in parallel, an inode is not marked complete unless prior passes in the pipeline are complete (e.g., a directory inode is marked complete only after all the child inodes (directory entries) are checked. Second, the C/R threads synchronize upon detecting errors. The thread that detects an inconsistency notifies other threads to stall and attempts to fix errors with (e.g., incorrect inode, blocks claimed by multiple inodes) or without user input (e.g., inconsistent bitmap), after which parallel execution is resumed. While tools such as C/Rs allow partial and full checks and checkpoint intermediate states, more robust tools could be added to increase C/R crash-consistency [17].

Optimizations. As additional optimizations to both e2fsck and pFSCK, we restrict the overheads of language localization as discussed in § 3.2, utilize Intel's hardware acceleration for checksum calculations, as well as improve the cache-readahead mechanism. We evaluate the benefits of these optimizations in § 6 (referred to as e2fsck-opt in graphs).

pFSCK support for other file systems and C/R tools. While pFSCK currently extends e2fsck (on Ext file system), the fine-grained inode-level data and pipeline parallelism and efficient scheduling can be applied to other C/R tools, such as xfs_repair and fsck.f2fs that implement multiple passes to check on files, directories, links, and others. We will explore designing a generic C/R in our future work.

Name	Description
e2fsck	original FSCK for EXT file systems
e2fsck-opt	optimized e2fsck
xfs_repair	XFS file system checker
pFSCK	proposed file system checker

Table 2: C/R systems evaluated.

Name	Description
datapara	Only data parallelism enabled
datapara+pipeline-split-equal	Pipeline + data parallelism equally distributing threads across passes
datapara+pipeline-split-optimal	Same as above but manually selects optimal thread assignment
sched	Pipeline + data parallelism with dynamic thread assignment
rsched	Sched configuration with system-level resource-awareness

Table 3: pFSCK incremental system design

6 Evaluation

We evaluate pFSCK to answer the following questions:

- Does pFSCK's data parallelism reduce C/R runtime by increasing CPU parallelism?
- How effective is pFSCK's pipeline parallelism in achieving concurrent execution of C/R passes?
- How effective is pFSCK's dynamic thread placement for different file system configurations?
- Can pFSCK's resource-aware scheduler effectively minimize the performance impact on other applications?
- How does pFSCK perform for online C/R?
- How does pFSCK perform in light of file system errors?

6.1 Experimental Setup

We use a machine equipped with a 64-core Dual Intel® Xeon Gold 5218 running at 2.30GHz, 64GB of DDR memory, a 1TB NVMe, and a 2TB Micron 5200 SATA SSD running Ubuntu 18.04.1. We run pFSCK on various file system configurations with varying thread counts. As seen in Table 2, we compare against vanilla e2fsck, e2fsck-opt (optimized e2fsck with reduced language localization overheads and Intel hardware-accelerated checksumming), and xfs_repair. Table 3 shows pFSCK's incremental design approaches.

File system configurations. The number of files and directories in a file system can be variable and dependent on applications, and there is no publicly available data. Our analysis of workloads like RocksDB (a key-value store), video server, web server, and mail server (using filebench), and two shared servers in our organization show that the file count dominate (99% files to 1% directories). To understand the impact of pFSCK's design on file and directory-intensive configurations, we use a 1TB NVMe with an 840GB file system utilizing 50 million inodes and a 2TB file system on SATA SSD utilizing 100 million inodes. We evaluate pFSCK's impact on a file-intensive (99% files), a medium directory-intensive (25%

directories), and an extreme directory-intensive (50% directories) configuration.

6.2 Data Parallelism

To understand the performance improvements and implications of pFSCK's fine-grained inode-level data parallelism that partitions inodes across threads in each pass but running the passes serially, we evaluate a file-intensive configuration (in Figure 5a), a directory-intensive configuration with 25% directories (in Figure 5b), and an extreme directory-intensive configuration with 50% directories (in Figure 5c). The x-axis shows four C/R approaches: the vanilla e2fsck, our optimized e2fsck (e2fsck-opt), xfs_repair with coarse-grained parallelism, and finally, our proposed pFSCK with data parallelism (pFSCK[datapara]). For xfs_repair and pFSCK[datapara], we also vary the thread counts from 2 to 16 threads.

File-intensive configuration. First, as shown in Figure 5a, our optimized e2fsck-opt outperforms the vanilla e2fsck by optimizing the CRC mechanism and avoiding language localization overheads. Next, xfs_repair parallelizes C/R in the granularity of coarse-grained allocation groups, which is ineffective. There are 16 allocation groups for our XFS filesystem configuration (by default). While xfs_repair checks the sanity of allocation groups in parallel, the directory metadata within the allocation groups are not checked in parallel. Specifically, when files are small, xfs_repair cannot check directory entries and link counts in parallel, with a substantial increase in C/R time. Besides, varying inode counts across allocations groups further impact performance (increasing allocation groups did not improve performance). Both e2fsck and pFSCK outperform xfs_repair for all cases. Finally, pFSCK[datapara] with fine-grained inode-level parallelism, reduces the runtime of the first pass (inode checking) by 2.1x and directory checking pass (Pass-2) by 1.8x, resulting in an overall C/R speedup of 1.9x for four threads over the vanilla e2fsck and 1.52x over e2fsck-opt. Beyond four threads, pFSCK's data parallelism scaling is hindered by high serialization and lock contention to update shared structures such as the used/free block bitmap.

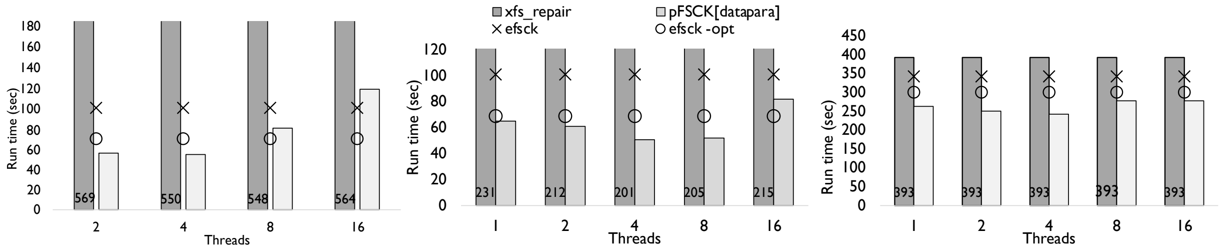
Directory-intensive Configurations. As shown in Figure 5b and 5c, the trends are similar for the directory-intensive file system. Even with 50% directories, pFSCK's Pass-1 and Pass-2 runtime reduces by 1.8x and 1.3x, respectively. pFSCK achieves an overall C/R speedup of 1.4x, 1.24x, and 1.8x over e2fsck, e2fsck-opt, and xfs_repair that does not parallelize directory metadata checking. For the 25% directory-intensive configuration, the gains are 2x over e2fsck. However, the synchronization overheads of global structures prevents pFSCK's data parallelism from scaling beyond 4 cores.

6.3 Pipeline Parallelism and Scheduling.

Next, we evaluate the benefits of combining data and pipeline parallelism and the need for a dynamic thread placement for a file-intensive configuration in Figure 6a and two directory-intensive configurations in Figures 6b and 6c. With pipeline parallelism, C/R passes run concurrently, and the threads of each pass add work for the next pass in a producer-consumer fashion. The x-axis shows the increase in the number of threads used for the C/R. We compare four cases: (1) *pFSCK[datapara]*, which only uses data parallelism running one pass at a time; (2) *pFSCK[pipeline-split-equal]*, which statically divides an equal number of threads for each of the simultaneously executing passes (e.g., two threads are assigned to the inode checking pass (Pass-1) and two threads to directory checking pass (Pass-2) in a 4-thread configuration); (3) *pFSCK[pipeline-split-optimal]*, which represents the best manually selected thread configuration; and (4) *pFSCK-sched*, which employs pFSCK's dynamic thread scheduler to dynamically assign threads based on the amount of outstanding work done within each pass. Single-threaded e2fsck and e2fsck-opt are marked as a baselines. Because e2fsck and pFSCK outperform xfs_repair in all cases, we do not show xfs_repair.

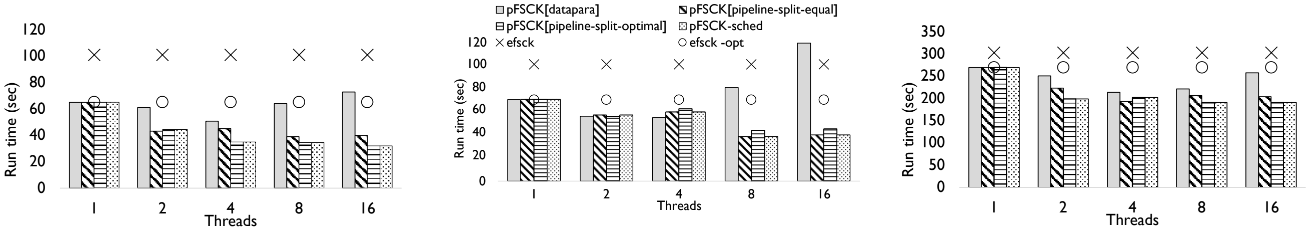
File-intensive Configuration. First, unlike the data parallelism-only approach, the pipeline parallelism approach improves performance when increasing thread count. Next, *pipeline-split-equal* approach splits threads equally across passes. For low thread counts (2 and 4 threads), the performance gains over data parallelism approach is minimal. This is because, for a file-intensive configuration, most work is done in the inode checking pass (Pass-1), static and equal division of threads across passes under-utilizes threads assigned in the directory checking pass. Increasing the thread count (along the x-axis) only marginally improves performance by increasing parallelism in the inode checking pass. In contrast, when employing a manually selected thread configuration using pFSCK's *pipeline-split-optimal* and assigning three-fourth of the threads to the inode checking pass, performance increases by up to 1.3x compared to data parallelism only. The concurrent work across passes also reduces the synchronization cost of data parallelism scaling beyond four threads. Finally, pFSCK's scheduler (*pFSCK-sched*) avoids the tedious manual process of optimal thread placement for different file system configurations by automatically migrating threads based on the relative amount of outstanding work to be completed across each pass. In fact, the dynamic thread placement improves performance by 1.1x compared to *pipeline-split-optimal*, resulting in an overall speedup of 2.6x compared to vanilla e2fsck.

Directory-intensive Configurations. Unlike the file-intensive configuration, for the extreme directory-intensive configuration (50% directories), both inode and directory checking passes demand substantial work, resulting in the need to frequently coordinate across the passes. We observe that automatic thread placement with *pFSCK-sched* controls the number of threads across passes, reducing contention within each pass while thread migration helps in accelerating inode checking (Pass-1) without accumulating a substantial number of directory inodes to be checked (in Pass-2). pFSCK employs delayed certification of directories (directory with subdirectory) as discussed in 5.2.2 which limits scalability.



(a) File-intensive FS. **(b)** Directory-intensive FS with 25% directory. **(c)** Directory-intensive FS with 50% directory.

Figure 5: Data Parallelism impact. The configurations use a total of 50 million inodes.



(a) File-intensive FS. **(b)** Directory-intensive FS with 25% directory. **(c)** Directory-intensive FS with 50% directory.

Figure 6: Comparison of pipeline parallelism and scheduler

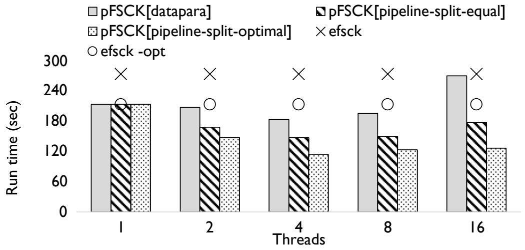
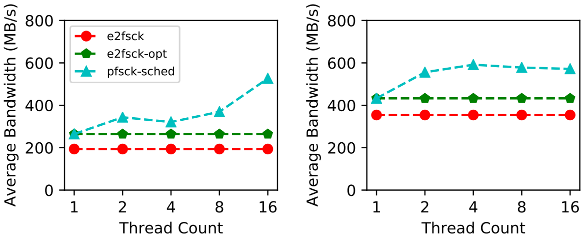


Figure 7: C/R on 2TB SSD with File-intensive FS.

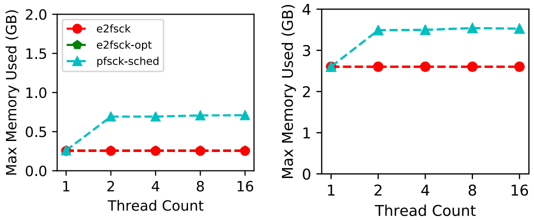


(a) File-intensive FS **(b)** Directory-intensive FS

Figure 8: I/O Bandwidth

Overall, the performance improves by up to 2.7x and 1.6x over e2fsck for 25% and 50% directory-intensive configurations.

Low-bandwidth SSD. Lastly, to understand gains on slower SATA-based SSDs (350MB/s sequential bandwidth) for a large 2TB configuration, Figure 7 shows pFSCK performance on a file-intensive configuration. pFSCK shows speedups of up to 2.1x and 1.73x over vanilla e2fsck and e2fsck-opt despite the lower bandwidth of SSDs compared to NVMe. *In summary, pFSCK's pipeline parallelism reduces serialization bottlenecks of data parallelism, and the dynamic thread placement reduces work imbalance, leading to significant performance gains in fast NVMe and SSD devices.*



(a) File-intensive FS **(b)** Directory-intensive FS

Figure 9: Memory Usage.

6.4 Storage Throughput and Memory Usage

We analyze the effective storage bandwidth use and increase in memory capacity with pFSCK for the file-intensive and extreme directory-intensive (50% directories) configurations. For brevity, we compare single-threaded e2fsck, e2fsck-opt, and multi-threaded pFSCK-sched.

6.4.1 Storage Throughput

Figures 8a and 8b show the storage bandwidth utilization for file-intensive and directory-intensive configurations in MB/s. First, our optimized e2fsck (e2fsck-opt) reduces the overhead between synchronous reads improving bandwidth utilization by 1.3x over e2fsck. In contrast, pFSCK increases I/O throughput for the file-intensive configuration by 1.9x and 2.7x for 8 and 16 threads, respectively, over e2fsck. I/O throughput utilization for directory-intensive file system improves by 1.7x, showing the benefits of pFSCK to utilize available disk bandwidth effectively.

The improvement in I/O bandwidth utilization comes from a combination of an pFSCK's threading, ability to prefetch directory blocks, better caching, and scheduling, which can dynamically migrate threads across passes based on the pending work. For the file-intensive configuration in Figure 8a, the scheduler allows threads to read inode blocks in Pass-1 and migrates extra threads to Pass-2 to read directory blocks in parallel, bumping up the bandwidth utilization. In Figure 8b, the I/O bandwidth utilization is better with most threads operating in the directory checking pass (Pass-2). However, due to serializing access to global shared structures (e.g., `db_list`) listed in Table 1, I/O bandwidth increase does not translate to higher performance with increasing thread count.

6.4.2 Memory Usage

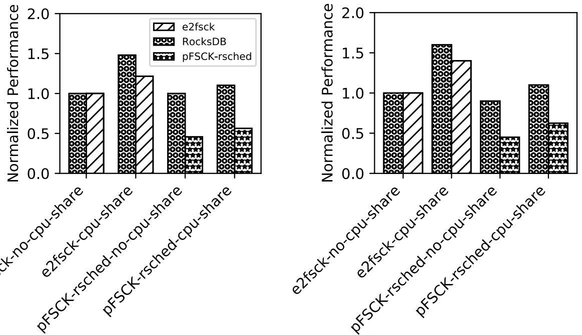
In Figure 9, we compare the memory (DRAM) capacity use. First, for file-intensive configuration in Figure 9a, the overall memory utilization is below 1GB for all approaches. Both e2fsck and e2fsck-opt show the same memory usage. Regarding e2fsck and e2fsck-opt memory utilization, which also applies to pFSCK, the memory use stems from data structures used for tracking directory information such as a `db_list` which hold a list of all directory data blocks, `dirinfo_list`, which tracks relationships between directories, `dx_dirinfo` list which keeps track of all directory HTREE blocks, as well as a dictionary structure used to verify consistency among the dirents within a directory. For *pFSCK-sched*, the memory usage increases by a nominal 300MB (2.1x). pFSCK's memory increase is mainly due to maintaining task queues. Apart from inode checking tasks for Pass-1, the threads discover directories and create directory block tasks for Pass-2 threads to process. Note that each task structure (in the queue entry) represents a fixed-size range of blocks to process. The queues are currently dynamically allocated and unrestricted but can be restricted to reduce memory increase. The range of blocks each task is assigned can also be increased to reduce the number of tasks being generated. Consequently, increasing the thread count does not or marginally increases memory use.

Next, for the directory-intensive configuration in Figure 9b, e2fsck and e2fsck-opt uses 2.6GB of memory. Similar to the file-intensive configuration, the main source of memory consumption is from data structures used for tracking directory information. With an extreme increase in directory count, the memory consumption of these data structures is significantly amplified. pFSCK's memory usage is comparable, only using 3.5GB, resulting in a 1.3x increase. Similar to the file-intensive configuration, the increase in memory usage is due to task structures being generated and added to task queues. With an extreme increase in directory count, the memory overheads of task structures also increase.

In general, memory usage is a function of file system utilization/configuration and not thread count. We argue that pFSCK's performance gains in today's system outweigh the nominal memory increase in today's systems with large memory capacity. Further, we believe memory use can be reduced through pFSCK's code optimizations.

6.5 System Resource-Aware Scheduler

File system C/Rs could run concurrently with other applications, where the C/R and applications can either operate on the same or separate file systems while sharing the same CPUs. To understand the effectiveness of pFSCK's resource-aware scheduler (pFSCK-rsched) in reducing the impact on other applications, we pick a popular multi-threaded and persistent I/O-intensive key-value store, RocksDB [4], which is used as a backend for several real-world applications [22,43]. We evaluate pFSCK-rsched in an offline setting, where C/R is performed on a file system separate from the file system RocksDB is using, and an online setting, where online (live) C/R is performed on a file system that is concurrently being updated by RocksDB. For both offline and online settings, we evaluate the performance of the following cases: (1) *e2fsck-no-cpu-sharing*, where RocksDB and the vanilla e2fsck do not share CPU cores; (2) *e2fsck-cpu-sharing*, where CPUs are shared between RocksDB and the vanilla e2fsck; (3) *pFSCK-rsched-no-cpu-sharing*, which employs pFSCK-rsched without sharing CPUs with RocksDB; and finally, (4) *pFSCK-rsched-cpu-sharing* which employs pFSCK-rsched sharing CPUs with RocksDB. We run RocksDB with 12 threads and facilitate CPU sharing by running pFSCK-rsched with 12 threads and restricting the affinity of all threads to 16 cores, resulting in the overlapping of 8 cores. Similarly, for e2fsck, we restrict the affinity of all threads to 12 cores, resulting in an overlap of 1 core. Due to space constraints, we show only the results for checking a file-intensive file system configuration.



(a) Offline C/R. **(b)** Online C/R.

Figure 10: Impact of resource-aware pFSCK for offline and online C/R. Results shown for file-intensive configuration.

6.5.1 Offline C/R with CPU Sharing.

Figure 10a shows the offline approach performance using separate file systems for each C/R and RocksDB. The x-axis shows e2fsck and pFSCK-rsched approaches without and with CPU sharing. In the y-axis, the results are normalized to the performance of e2fsck running with RocksDB without sharing CPUs (*e2fsck-no-cpu-sharing*).

First, when sharing CPUs, the runtime of vanilla e2fsck and RocksDB is significantly impacted (shown as *e2fsck-cpu-share*) compared to *e2fsck-no-cpu-share* due to frequent context switches which take away effective CPU time from RockDB; e2fsck's performance degrades by 1.2x and RocksDB's performance degrades by 1.5x compared to the no-sharing approach. In contrast, with pFSCK's resource-aware scheduler, CPU sharing between the pFSCK-rsched and RocksDB (*pFSCK-rsched-cpu-sharing*) has minimal impact for both pFSCK and RocksDB. The resource-awareness

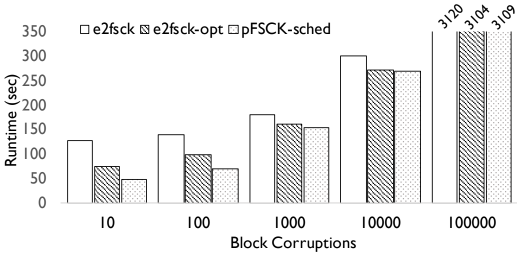


Figure 11: Repair runtime for varying corruption count.

capability adaptively downscales the number of threads being utilized to carry out C/R, reducing CPU context switches away from RocksDB and minimizing related overheads. Athough pFSCK-rsched and RocksDB intially overlap 8 out of the 16 cores, pFSCK-rsched is able to downscale threading to around 4-6 threads, allowing RocksDB to consistently utilize 12 out of the total 16 cores. As a result, pFSCK-rsched and RocksDB show minimal performance degradation of 1.07x and 1.05x, respectively, compared to the no CPU sharing case.

6.5.2 Online C/R with CPU Sharing.

Figure 10b shows the results when each C/R and RocksDB share the CPU as well as the file system. As discussed earlier in 5.4.2, pFSCK utilizes the LVM-based snapshots to capture file system changes and perform C/R on a stable version of the file system represented by the snapshot. Similar to offline C/R evaluation, we normalize the results to the performance of e2fsck running with RocksDB without sharing CPUs.

First, when overlapping e2fsck and RocksDB, performance significantly degrades by 1.4x and 1.6x, respectively. The degradation is mainly due to frequent CPU context switching between e2fsck and RocksDB. However, this main source of performance degradation increases the time the snapshot must remain active, resulting in further performance degradation due to LVM snapshot overheads. Next, with pFSCK-rsched, the performance degradation when co-running pFSCK-rsched with RocksDB is minimal. This is due to pFSCK-rsched's resource-aware thread assignment (similar to offline setting) which mitigates performance impact and context switching overheads by scaling the number of threads pFSCK uses. Because performance impact from context switching is minimized, the amount of time the snapshot must active is minimized, mitigating any further performance degradation due to LVM snapshot overheads. The performance degradation compared to the baseline (no CPU or file system sharing) for both pFSCK-rsched and RocksDB is 1.2x. Although higher than the offline approach, most of it is due to disk sharing and the resulting LVM snapshotting overheads.

Summary. pFSCK-rsched's resource awareness effectively adapts to the number of available CPU cores (and threads) for C/R and maximizes their utilization for better performance in both an offline and online setting. The performance impact on co-running application is also minimized.

6.6 Performance with Errors

To evaluate pFSCK's performance with file system errors, we use e2fsprogs's fuzzing tool, *e2fuzz*, to introduce random block corruptions to a file-intensive configuration. In Figure 11, we introduce up to 100K corruptions in the x-axis and compare e2fsck, e2fsck-opt, and pFSCK-sched that uses eight threads. Note that prior studies real-world systems show that the scale of corruptions can be just a few bits or bytes, and the hardware and software corruptions could significantly vary, ranging from silent bit corruptions to FTL metadata corruptions and shorn or incomplete writes [12, 18, 27].

First, even for 100 corruptions, pFSCK-sched speeds up C/R by up to 2.7x and 1.6x over e2fsck and e2fsck-opt, respectively. However, for 10K corruptions, pFSCK-sched performs similarly to e2fsck-opt and speeds up C/R by only 1.1x compared to e2fsck. pFSCK's speedup reduces with increasing corruption counts because long and serially executed error-fixing operations start to dominate the overall runtime for all C/Rs including pFSCK. Further, for pFSCK-sched, we observe that if the corruption count is greater than 10K, synchronization overheads start to further diminish performance gains from parallelism. To mitigate diminishing returns from thread synchronization, pFSCK tracks the number of errors encountered and reverts to serial checking after discovering a 10K errors. This allows pFSCK to perform similarly to e2fsck-opt for higher error counts, experiencing only slight performance deterioration due to initial thread synchronization. Our future work will focus on exploring ways to parallelize fixes to accelerate C/R for highly corrupted file systems.

7 Conclusion & Future Work

With a goal of accelerating file system checking and repair tools, we propose pFSCK, a parallel C/R tool that exploits CPU parallelism and the high bandwidth of modern storage devices to accelerate C/R time without compromising correctness. pFSCK explores fine-grained parallelism by assigning threads to inodes, blocks, or directories and efficiently performing C/R using data parallelism within each pass and pipeline parallelism across multiple passes. In addition, pFSCK enables efficient thread management techniques to adapt to varying file system configurations as well as minimize performance impact on other applications. As a result, pFSCK shows more than 2.6x gains over e2fsck and 1.8x over xfs_repair that provides coarse-grained parallelism. In light of pFSCK's limitations, future work will explore accelerating pFSCK for hard disks while mitigating costly seeks due to random accesses, reducing memory overheads through more efficient data structures and rate-limiting, and finally accelerating fixes for disks with higher corruption counts.

Acknowledgements

We thank the anonymous reviewers and Dean Hildebrand (our shepherd) for their insightful comments and feedback. We thank the members of Rutgers Systems Lab for their valuable input. This material was partially supported by funding from NSF grant CNS-1910593. We also thank Rutgers Panic Lab for helping with the storage infrastructure.

References

[1] Disk check takes too long to check. linuxquestions.org. https://www.linuxquestions.org/questions/linux-hardware-18/disk-check-takes-too-long-to-check-510584/.

[2] e2fsck: fsck for ext4. https://linux.die.net/man/8/e2fsck.

[3] e2scrub: online fsck for ext4. https://lwn.net/Articles/749106/.

[4] Facebook RocksDB. http://rocksdb.org/.

[5] Intel-Micron Memory 3D XPoint. http://intel.ly/1eICR0a.

[6] Linux sched() man page. http://man7.org/linux/man-pages/man7/sched.7.html.

[7] StackExchange - Extremely long time for an ext4 fsck. https://unix.stackexchange.com/questions/78785/extremely-long-time-for-an-ext4-fsck, Mar 2013.

[8] File system check (fsck) is slow and running for a very long time. https://access.redhat.com/solutions/2210281, Sep 2016.

[9] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 353–369, 2019.

[10] 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, pages 151–167, Berkeley, CA, USA, 2016. USENIX Association.

[11] William (Bill) E. Allcock. Parallel File Systems at HPC Centers: Usage,Experiences, and Recommendations. https://www.nersc.gov/assets/Uploads/W01-DataIntensiveComputingPanel.pdf.

[12] 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. *ACM Trans. Storage*, 4(3), November 2008.

[13] 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. *ACM Transactions on Storage (TOS)*, 4(3):8, 2008.

[14] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems - SIGMETRICS 07*, 2007.

[15] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243. ACM, 2013.

[16] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. *ACM Transactions on Storage (TOS)*, 8(4):1–29, 2012.

[17] Om Rameshwar Gatla, Muhammad Hameed, Mai Zheng, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojević, Cyril Guyot, and Robert Mateescu. Towards robust file system checkers. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 105–122, Oakland, CA, February 2018. USENIX Association.

[18] John Goerzen. Silent data corruption is real. https://changelog.complete.org/archives/9769-silent-data-corruption-is-real/.

[19] Haryadi S Gunawi, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Sqck: A declarative file system checker.

[20] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Sqck: A declarative file system checker. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 131–146, Berkeley, CA, USA, 2008. USENIX Association.

[21] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 1–14, Oakland, CA, 2018. USENIX Association.

[22] Ethan Hamilton. Rocksdb is eating the database world. https://rockset.com/blog/rocksdb-is-eating-the-database-world/.

[23] Michael Hasenstein. The logical volume manager (lvm). *White paper*, 2001.

[24] Val Henson, Zach Brown, and Arjan van de Ven. Reducing fsck time for ext2 file systems. 04 2019.

[25] Val Henson, Amit Gud, Arjan van de Ven, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the Second Conference on Hot Topics in System Dependability*, HotDep'06, pages 7–7, Berkeley, CA, USA, 2006. USENIX Association.

[26] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *HotDep*, 2006.

[27] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating file system reliability on solid state drives. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 783–798, Renton, WA, July 2019. USENIX Association.

[28] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating file system reliability on solid state drives. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 783–798, 2019.

[29] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating file system reliability on solid state drives. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 783–798, Renton, WA, July 2019. USENIX Association.

[30] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[31] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redesigning lsms for non-volatile memory with novelsm. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 993–1005. USENIX Association, 2018.

[32] Ram Kesavan, Harendra Kumar, and Sushrut Bhowmik. WAFL iron: Repairing live enterprise file systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 33–48, Oakland, CA, February 2018. USENIX Association.

[33] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[34] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, Santa Clara, CA, 2015.

[35] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. OOPSLA ’09, page 227–242, New York, NY, USA, 2009. Association for Computing Machinery.

[36] W. Li, Y. Yang, J. Chen, and D. Yuan. A cost-effective mechanism for cloud data reliability management based on proactive replica checking. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 564–571, 2012.

[37] HPC-Users Mailing List. Outages in HPC Systems. https://maillists.uci.edu/pipermail/hpc-users/2019-December/000095.html.

[38] M. Lu, T. Chiueh, and S. Lin. An incremental file system consistency checker for block-level cdp systems. In *2008 Symposium on Reliable Distributed Systems*, pages 157–162, Oct 2008.

[39] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Marshall Kirk Mckusick. Ffsck: The fast file-system checker. *Trans. Storage*, 10(1):2:1–2:28, January 2014.

[40] Marshall K. McKusick. Improving the performance of fsck in freebsd. *;login:*, 38(2), 2013.

[41] Marshall Kirk McKusick, Willian N Joy, Samuel J Leffler, and Robert S Fabry. Fsck- the unix† file system check program. *Unix System Manager’s Manual-4.3 BSD Virtual VAX-11 Version*, 1986.

[42] Mtanski. mtanski/xfsprogs github.com/mtanski/xfsprogs/preadv2/repair. https://github.com/mtanski/xfsprogs/tree/preadv2/repair, Feb 2015.

[43] Arjun Narayan and Peter Mattis. Why we built cockroachdb on top of rocksdb. https://www.cockroachlabs.com/blog/cockroachdb-on-rocksd/.

[44] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[45] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI’18, pages 145–160, Berkeley, CA, USA, 2018. USENIX Association.

[46] Omar Sandoval. A survey of bugs in the Btrfs filesystem. https://courses.cs.washington.edu/courses/cse551/15sp/projects/osandov.pdf.

[47] Ric Wheeler. fs_mark. https://sourceforge.net/projects/fsmark/.

[48] Matthew Wilcox and Ross Zwisler. Linux DAX. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

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

[50] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST’16, 2016.

[51] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W. Zhao, and Elizabeth S. Yang. Reliability analysis of ssds under power fault. *ACM Trans. Comput. Syst.*, 34(4):10:1–10:28, November 2016.

Pattern-Guided File Compression with User-Experience Enhancement for Log-Structured File System on Mobile Devices

Cheng Ji[1], Li-Pin Chang[2,3], Riwei Pan[4], Chao Wu[4], Congming Gao[5], Liang Shi[6], Tei-Wei Kuo,[4] and Chun Jason Xue[4]

[1]*Nanjing University of Science and Technology* [2]*National Chiao Tung University* [3]*National Yang Ming Chiao Tung University*
[4]*City University of Hong Kong* [5]*Tsinghua University* [6]*East China Normal University*

Abstract

Mobile applications exhibit unique file access patterns, often involving random accesses of write-mostly files and read-only files. The high write stress of mobile applications significantly impacts on the lifespan of flash-based mobile storage. To reduce write stress and save space without sacrificing user-perceived latency, this study introduces FPC, file access pattern guided compression. FPC is optimized for the random-writes and fragmented-reads of mobile applications. It features dual-mode compression: Foreground compression handles write-mostly files for write stress reduction, while background compression packs random-reading file blocks for boosted read performance. FPC exploits the out-of-place updating design in F2FS, a log-structured file system for mobile devices, for the best effect of the proposed dual-mode compression. Experimental results showed that FPC reduced the volume of total write traffic and executable file size by 26.1% and 23.7% on average, respectively, and improved the application launching time by up to 14.8%.

1 Introduction

Mobile devices including smartphones, tablets, and wearable devices are now a necessity in everyone's daily life. Recent researches reported that the number of smartphone shipments surpassed 1.37 billion in 2019 [1] and 86% of them were based on the Android [2]. Mobile devices employ flash memory for persistent data storage. While the performance of mobile processors is improving drastically, the improvement of mobile storage performance is, however, relatively slow. Recent studies report that I/O operations on mobile storage are write-dominant [3–7], and the write pattern is highly random and synchronous. These write operations are identified closely related to user-perceived latencies due to the relatively high write latency of flash memory [8, 9]. In addition, as flash memory technology is evolving toward high cell-bit-density at the cost of degraded endurance, the high write stress negatively impacts on the flash-storage lifespan.

Android-based mobile devices exhibit very distinct file usage patterns compared with desktop systems: First, mobile applications heavily rely on an embedded database layer, SQLite, for transactional data management; Second, Android packs various runtime resources such as executable binaries and compiled resources into large executable files. We examined the contents of these files and found that they are highly compressible. While the database files contribute to a large portion of the write traffic, executable files are large in size. Intuitively, existing file compression techniques, such as those reported in [10–13], can be adopted to reduce write stress and to save space. However, the existing designs might not be effective or risk degraded user experience in mobile devices.

Manipulating SQLite databases generates many small file overwrite and append operations [6, 14]. These small operations are highly fragmented in the storage space and the cause has been identified related to the file fragmentation problem [8]. With the fragmented updates, file compression on top of a conventional in-place-updating file system, like Ext4 [15], may create many holes in the storage space because a compressed file block may not fit in its original space after an update. With the small append operations, file compression cannot use a large compression window on new data for a better compression result. Regarding Android executable files, although they are sequentially written upon installation, they are subject to small, random read operations during application launching. Decompressing file blocks from random file offsets significantly amplifies the I/O read overhead because of the larger unit size of block I/O [12, 13]. These unique file access patterns of database files and executable files in Android mobile devices are, however, not well studied in prior file compression work.

We believe that file compression should be judiciously applied to files based on their access patterns. This study presents ***File Pattern-guided Compression (FPC)*** for mobile devices. FPC features foreground compression and background compression. Considering the timing overhead of compression, foreground compression is applied only on the write-intensive, highly compressible SQLite files. In particular, SQLite journal files are barely read (write-ahead logging journal) or never read (roll-back journal). For the journal files, FPC further applies deep compression by packing file-system metadata with user data and compressing them using a larger compression window. Executable files, which are also highly compressible, are subject to small, random reads upon application launching. Hence they are not suitable for sequential compression in the foreground. Instead, this paper proposes to apply infrequent background compression to re-organize read-critical blocks of executable files through compression. As a result, both user-perceived application launching latency and storage space utilization can be improved.

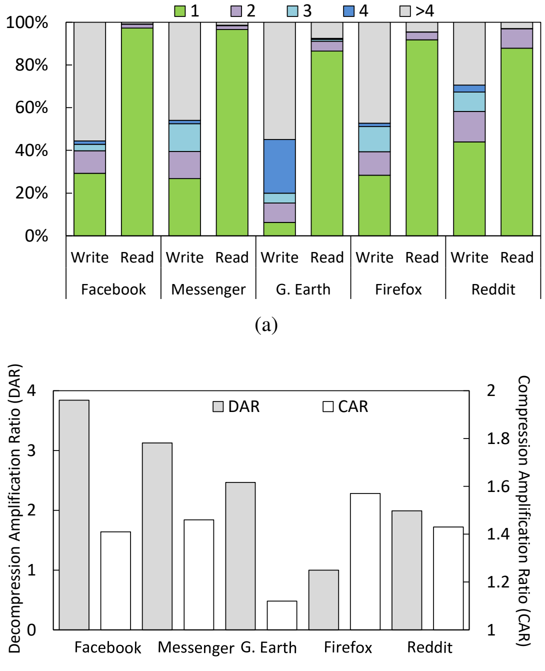
The effect of the proposed FPC is best achieved by an implementation on top of a log-structured file system, e.g., F2FS [16] for mobile storage. FPC exploits out-of-place updating and reverse mapping, which are existing mechanisms in F2FS, for foreground compression and critical block reorganization, respectively. With out-of-place updating, small, fragmented writes and appends to SQLite files and their ancillary file-system metadata can be combined as sequential, deep compression with a large compression window. With the reverse mapping of storage addresses to file block offsets, it is possible to load multiple compressed read-critical blocks of executable files through a single block I/O request to accelerate application launching. In summary, this work makes the following contributions:

- Proposing a foreground compression method for write mostly, highly compressible files to improve write stress and energy consumption of mobile storage;
- Proposing a background compression method that identifies and re-organizes read-critical blocks in executable files for fast application launching and space saving;
- Exploiting out-of-place updating and reverse mapping of F2FS for the best effect of deep, metadata-level file compression and fast application launching.

2 Background and Motivation

2.1 I/O System and Storage of Mobile Devices

Android is the dominant operating system for mobile devices today. The Android I/O system consists of host system software and a flash storage device. The host software includes a lightweight database layer, SQLite, for transactional data management. SQLite operates on top of the file system layer, where two major options are provided: Ext4 [15], an in-place updating file system, and F2FS [16], a log-structured file system. Because random file writes may unexpectedly increase the cost of garbage collection of flash storage, F2FS benefits flash storage by converting random updates into sequential, out-of-place updates. As out-of-place updates create outdated data in the storage space, F2FS needs to timely compact valid data to provide contiguous free space for future sequential writes.



(a)

(b)

Figure 1: (a) Characterization of file write and read sizes for mobile applications (unit: page). (b) Low compression efficacy of traditional sequential compression approaches. *Compression Amplification Ratio* (CAR) and *Decompression Amplification Ratio* (DAR) are presented, respectively.

High Write Stress. Applications heavily rely on the SQLite journaling mechanism for data integrity guarantee, producing enormous synchronous, random block writes [6]. The write traffic is further amplified by other components of the I/O system. Specifically, free-space defragmentation for F2FS involves many extra data migrations [16], and flash garbage collection inside mobile storage requires data movements before memory erasing [17]. The multiple levels of write amplification become even worse when the level of file system fullness is high. The amplified write traffic noticeably degrades user-perceived latency [9]. In addition, as modern flash technology is evolving toward high bit-cell density at the cost of reduced endurance, e.g., a TLC flash block can only withstand about 1,000 P/E (program-erase) cycles [18], the excessive write traffic also poses concerns to the storage lifespan.

2.2 Pitfalls of File Compression

File compression is expected to save storage space and reduce the amount of I/O. However, it may not be the case for mobile storage because of the highly random nature of file reads and writes of mobile applications. In this section, we demonstrate the high randomness of read and write under selected popular applications and show that existing file compression designs could be harmful to space utilization and read performance.

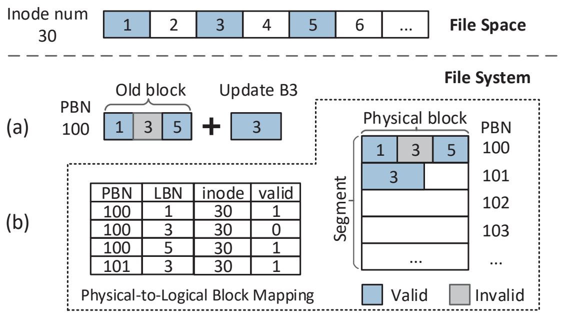
In Android devices, read traffic and write traffic are mainly contributed by executable files and SQLite files, respectively [3, 14]. Figure 1(a) reports the size distribution of file read operations on *.apk executable files and that of file write operations on SQLite files. Results indicate that roughly more than one half of the write operations to SQLite files were not larger than 4 pages (16KB). Most of these small writes were bound for random file offsets. For read operations, nearly 90% of all read operations were not larger than one page in all applications. The file offsets of these reads were also highly fragmented.

Many existing compression file systems, e.g., Btrfs [10], JFFS2 [11] and EROFS [12], allow only compressed file blocks of consecutive offsets to be stored in the same storage block. A new storage block is allocated for a compressed file block if it does not continue the file offset of the last compressed file block. This design, referred as the *sequential compression method*, can seriously degrade the space utilization in mobile storage. To assess the severity of the problems, in Fig 1(b) we report *Compression Amplification Ratio* (CAR), which is the ratio of the total number of physical blocks [1] required to store a series of compressed logical blocks with the sequential compression method to the minimal number of physical blocks required to store all the compressed logical blocks. CAR reflects how the sequential compression method degrades space efficiency on random write through internal fragmentation. The CAR values indicated that the sequential compression method incurred more than 40% extra space requirement for 4 out of the 5 applications.

We then show how application launching suffered from an amplified read overhead with the sequential compression method. We employed *Decompression Amplification Ratio* (DAR) to show the ratio of the total number of compressed logical blocks stored in the physical blocks that are read to launch an application to the total number of compressed logical blocks that are actually required to launch the application. For example, DAR is 4 if a physical block stores four compressed logical blocks and only one of them is actually used. Fig 1(b) shows the DAR values were even higher than the CAR values because one-page reads dominated the overall read traffic. The CAR and DAR results showed that the existing sequential compression method unexpectedly degrades space utilization and amplifies the read overhead for application launching on mobile devices. This underlines the need for a new space management strategy to cope with the unique random I/O pattern of mobile applications.

2.3 Benefits of File Compression with LFS

File system compression should achieve a high compression efficiency (space saving) with a reduced decompression penalty. It is possible to achieve both by taking advantage of the file system structure and application behaviors of file access. This study is based on the log-structured file system for mobile storage. Compared with conventional in-place-updating file systems, e.g., Ext4, LFS is highly friendly to file compression, as discussed below:



PBN	LBN	inode	valid
100	1	30	1
100	3	30	0
100	5	30	1
101	3	30	1

Figure 2: Updating compressed data with (a) in-place-updating file system and (b) a log-structured file system.

Out-of-Place Updating. Conventional in-place-updating file systems have a disadvantage of handling compression. In Figure 2(a), three data blocks are compressed and packed together into the same physical block (PBN 100). Consider that B3 is updated. If the new compressed size of B3 is larger than the old version, it is impossible to overwrite B3 in place. One solution is to re-write B1, B3, and B5 and repack them tightly in another free space. Another option is to align compressed data to predefined boundaries to allow future size changes of compressed data. However, with both options above, file compression suffers, from either amplified write traffic or internal fragmentation [19, 20]. By contrast, as shown in Figure 2(b), LFS appends the compressed B3 to a new block, avoiding rewriting of existing data and wasting of free space.

Reverse (Physical-to-Logical) Mapping. To support data migration of space cleaning, LFS, e.g., F2FS, maintains physical-to-logical block (P2L) mapping. The P2L mapping is necessary to determine whether a piece of data is valid (and requires migration) and to update new locations of data during cleaning. As Figure 2(b) shows, the P2L mapping provides the `inode` number and file offset of a compressed logical block (§ 4.3). Interestingly, file compression can leverage the existing P2L mapping mechanism for efficient decompression. We discerned that the launching of mobile applications, whose latency affects user experience the most [8], generated small, random reads on executable files (Section 3.3.1). With P2L mapping, it is possible to compress the file blocks necessary to application launching and pack them into physical blocks. This way, fewer block read requests are required to launch an application, improving the user-perceived latency.

Although the log-structured file system is friendly to compression, prior compression studies paid little attention to the read/write patterns of mobile systems. This study proposes to exploit the file access behaviors of mobile applications and the structure of F2FS to address two major design challenges: 1) Efficient file compression for write stress reduction and space saving and 2) Efficient re-organization and decompression of

[1] In the rest of this paper, we refer to blocks in the storage space as physical blocks and blocks in the file address space as logical blocks.

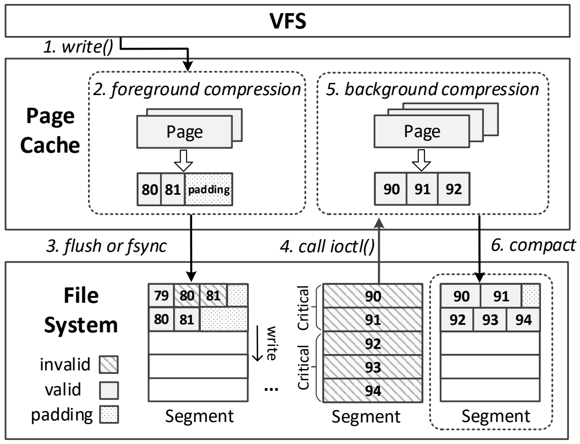


Figure 3: FPC architecture. Steps 1 to 3 show foreground compression on write-intensive files and Steps 4 to 6 show background compression/re-arrangement of read-critical data.

executable files for improved user experience.

3 Pattern-Guided File Compression

This section presents the proposed design principle and the details of foreground and background compression.

3.1 File Access Behaviors of Mobile Apps

We propose categorizing files according to their types of access (read or write) and hotness (access frequency): 1) Write-hot, read-cold files: Roll-back journals (*.db-journal) are a good fit in this category because they are frequently written [21]. However, they are rarely read (except during crash recovery). SQLite database files are also a good fit because many mobile applications frequently write SQLite database files but barely read them [22]. 2) Write-cold, read-hot files: Executable files fall in this category because they are immutable after installation or update. Android executable files are large and highly compressible [13]. However, the read latency of executable files is critical to user experience, so it is crucial to optimize the decompression overhead.

Both SQLite files and executable files are subject to random access: SQLite database files are prone to random updates, while executable files are subject to random reads. Although random updates will be converted into bulk writes through out-of-place updating, random reads, however, should be optimized through rearrangement of file blocks. Figure 3 shows the architecture of the proposed *File Pattern-guided Compression approach (FPC)*. FPC performs foreground compression on SQLite files for write stress reduction. On the other hand, executable files are left to background activities of block re-arrangement and compression for improved user experience and space saving.

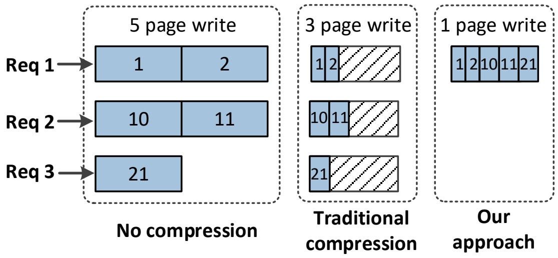


Figure 4: Comparison of traditional *sequential compression* approaches and the proposed compression approach. The Logical Block Number (LBN) is shown inside each data block.

3.2 Foreground Compression

This section presents the proposed foreground compression (FC) solution for write stress reduction. FC is focused on the compression of write-hot but read-cold files.

3.2.1 Non-Sequential File Block Compression

On-line file compression is a feature of existing file systems such as Btrfs [10] and JFFS2 [11]. In the prior designs, a physical block can only store compressed logical blocks of contiguous file offsets. This is because, first, file writes in desktop computers and servers are large in size (compared with mobile devices), so a sequential burst of compressed data sufficiently utilizes a physical block. Second, file compression is a separate layer in file system, and therefore compression of sequential file blocks minimally affects the existing index scheme of file blocks. This *sequential compression method*, which is adopted by Btrfs and JFFS2, results in poor space utilization in mobile storage, because the major write traffic contributor SQLite [4, 14] produces many small writes bound for random file offsets. In this case, a compressed file block usually demands a new physical block due to the irrelevant file offset from the prior compressed file block.

Fig. 4 shows the problem of the sequential compression method. Consider the three pending file write operations on the left-hand side. Three physical blocks are required for compression because the three file writes do not have sequential file offsets. Because compression reduces the file block sizes, these three physical blocks suffer from poor space utilization. On the contrary, we propose allowing file blocks of irrelevant file offsets to share the same physical block. As the right-hand side of Fig. 4 shows, only one physical block is used and the space utilization is high. However, compression of non-sequential file blocks is challenging because it increases the index resolution of file blocks to the sub-block level.

3.2.2 Selective Foreground File Compression

Foreground compression (FC) is performed in real time. Since unconditional compression risks poor write latency and extra energy consumption, foreground compression is highly selective to avoid such a drawback.

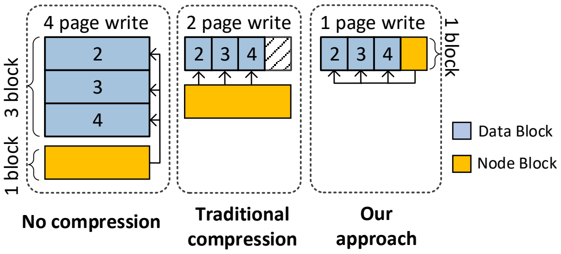


Figure 5: Comparison of traditional compression approaches and the proposed metadata-level file compression approach.

Selection of File Types. Since FC is part of the file system, it can simply ignore writes associated with the file types that are known to be incompressible, e.g., files with multimedia extensions *.jpg, *.mp4 and so on. Among all the other file types, SQLite files have been identified write-intensive and highly compressible [23]. In this study, *compression ratio* is defined as the ratio of the size after compression to that before compression. The smaller the better. We measured that the compression ratio of many SQLite files was better (lower) than 0.2, and compressing these SQLite files could benefit the write latency. On the other hand, although executable files are also highly compressible, sequential compression of executable files will significantly amplify the read overhead because such files are subject to small, random reads during application launching. FC leaves executable files to background compression.

Selection of Page Writes. Although FC can simply compress all writes associated with files having a *.db extension, it is possible that SQLite files are embedded with incompressible contents. For example, the Google Earth app stores map image tiles in *.db files using the BLOB (binary large object) format. A prior study reports that real-time identification of data compressibility is feasible [24]. Here, FC employs a sampling technique to quickly identify such incompressible contents: FC always compresses the first cached file page of an SQLite file write operation. If the first cached page is highly compressible, i.e., its data size can be reduced by at least one half, then FC compresses the rest cached pages of the write. Otherwise, FC forwards all the cached pages of the write to the original F2FS write logic.

3.2.3 Metadata-Level File Compression

F2FS stores user data in data blocks and file-system metadata (e.g., inodes) in node blocks. These two types of blocks are written to separate free spaces because node blocks are considered being updated more often. Small, synchronously-written files will experience a high metadata overhead.

The left-hand side of Fig. 5 shows the space allocation without file compression. Now, let the file undergo compression, the middle of Fig. 5 shows that the three data blocks are packed into a physical block while the uncompressed node block still occupies a second physical block. In this study, we propose writing data blocks of a *.db-journal file and their associated node block to the same segment through compression. The right-hand side of Fig. 5 shows that only one physical block is written with this method, while the traditional compression in the middle requires two. There are several rationales behind this design: First, the node block and data blocks of a *.db-journal file share the same lifetime because a rollback journal is discarded upon a successful SQLite transaction. Second, rollback journals are write-only (except during crash recovery) so packing metadata and user data together would have little impact on read performance. Although the design described above is based on rollback journaling in DELETE mode, it is also applicable to PERSIST mode. In PERSIST mode, rollback journals are reused. Like in DELETE mode, data blocks and the node block of reused journals are updated through F2FS out-of-place writing, and they are packed together for compression.

F2FS flushes data blocks before writing node blocks to avoid reference to uninitialized data. In our method, physical blocks containing all compressed data blocks are still flushed first. Let a mixed block be a physical block storing a few compressed data blocks and their associated, compressed node block. It is possible that upon crash recovery, the compressed node block in a mixed block is valid but its associated compressed data blocks in the same mixed block contain uninitialized data. To deal with this problem, we propose inserting a checksum to every mixed block. If a checksum fail is detected on a mixed block during crash recovery, the compressed node block in the mixed block is discarded.

3.3 Background Compression

Foreground compression on sequentially-written, random-read executable files risks degraded application launching performance. This part investigates the access patterns of executable files and proposes background compression (BC).

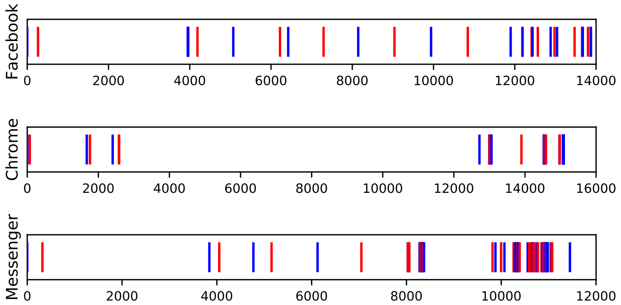


Figure 6: Distribution of read addresses within executable files during application launching. The X-axis shows the block offsets relative to the begin of files. Read requests having consecutive LBAs are marked in the same color (red and blue are used alternatively for clear presentation).

3.3.1 Highly Random Reads of Executable Files

Executable files, including .apk, .dex, .odex and .oat files, contribute to a large proportion of storage space [14], and they are highly compressible. The latencies of reading executable during application launching are crucial to user experience, because users have to wait until all necessary executable data are decompressed and loaded into memory.

We inspected how executable files were read with typical mobile applications. In order to accurately identify the required file data, we inserted routines to the Virtual File System (VFS) to extract related system calls, e.g., instrumenting `do_mmap` function to record reads on memory-mapped executable files and instrumenting `do_generic_file_read` function to extract reads on non-memory-mapped files. The duration of launching ended when the executable file of the application did not receive any read for one second.

Figure 6 shows the distributions of read addresses within the `base.apk` file, which is the main executable file for Facebook, Chrome, and Messenger. Results show that read requests of the inspected applications were small and random. While the executable files were large, only a small portion of executable file data was actually fetched for launching. For example, the executable file of Facebook was 80.6 MB, but only 632 KB of the file was read to launch Facebook. File pages were fetched through run-time demand paging, but the address distribution shows that these required pages were not well organized based on their correlation.

Apk files are actually a package of resource files. These files contribute a significant portion of read traffic during application launching, e.g., 49%, 27% and 21% of total read requests for Facebook, Chrome, Messenger, respectively. The random reads of apk files had great impacts on application launching latencies, which will be shown in Section 5. We de-compiled the Facebook's base.apk and analyzed which resource files were actually read for launching. The accessed resource files were identified by matching the read addresses and the file offsets of the resource files within the apk. The base.apk contained 17,439 resource files. Table 1 shows that only a small subset of the resource files (110 out of 17,439) were read. These required resource files (`*.xml`, `*.png`, etc.) were dispersed to random locations within the `apk` and were all accessed through single-block read requests. These small, random reads increased the block I/O count and degraded the user-perceivable application-launching latency.

Table 1: Resource files read from Facebook's *base.apk*.

Type	.xml	.arsc	.png	.dex	others	apk metadata
File count	51	1	41	9	8	1
Blocks read	1	32	1	1	1	17

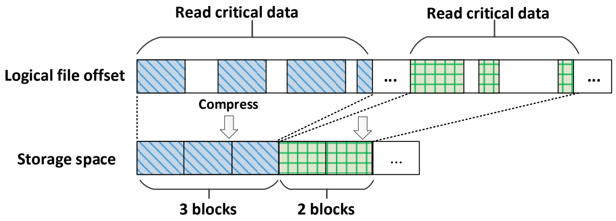


Figure 7: Compression of read-critical data in executable files.

3.3.2 Read-Guided File Compression

Decompressing small pieces of data from random file offsets negatively impacts on read performance. Compression provides an opportunity of reorganizing necessary file blocks to reshape the read patterns for better decompression efficiency.

Read-Critical Data Compression: We refer to a piece of data in an executable file as *read-critical data* if the data is required to launch an application. Because of the random read pattern of application launching, sequential compression of executable files risks the mixture of read-critical and non-read-critical data in a physical block. The mixture of data significantly amplifies application launching time because non-read-critical data are loaded and decompressed. The upper half of Figure 7 illustrates that critical data are scattered in an executable file. We propose monitoring the read-critical data set during application launching. Later on, upon requests, the file system compacts these critical data and compresses them into file blocks, as shown in the bottom half of Figure 7. Notice that the compaction changes the storage layout of file blocks but *not* the logical order of file blocks.

Compacting and compressing read-critical data avoids to load and decompress non-read-critical data and thus prevents decompression from degrading application launching time. It also complements the existing file pre-fetching mechanism, which could unexpectedly load non-read-critical data from sequential file offsets. When reading and decompressing a piece of read-critical data, the file system also brings the other read-critical data of the same physical block into the page cache. The prefetching of read-critical data requires the physical-to-logical mapping of F2FS (see Section 4.3), which is a unique feature of log-structure file systems.

Read-Critical Data Identification: The efficacy of the proposed read-critical data compression is subject to the I/O pattern of executable files. It has been reported that application launching exhibits highly predictable I/O patterns on desktop computers [25]. This phenomenon is also true for mobile applications: a cold start process of launching an application was tested through the `am start` command of the `adb shell` with the page cache cleared beforehand. We monitored the file read operations on the file blocks of the base.apk file of Facebook, Chrome, Messenger, Twitter, Google Earth, and Firefox for 5 rounds of cold starts. Results show that the set of file blocks read during the multiple cold starts barely changed (difference was between 1% to 3%). The read performance of these file blocks affects the user experience the most because the application screen is not fully rendered yet. As the applications continued to launch after the `am start` command returned, a higher degree of variation in the start-up file blocks was observed as they began to display random advertisements and splash screens.

Based on the results above, we propose capturing the core set of the read-critical data, i.e., those shared among different rounds of application cold starts, for fast application launch. When an executable file is opened (or memory-mapped), our method records the offsets of file reads. The proposed design collects the read offsets for the first three rounds of cold starts of a new application. After this, the core read-critical data will be compacted and compressed into a set of physical blocks. Non-core read-critical data will be compressed to another set of physical blocks. When the system is lightly loaded, a resident user-level process invokes an `ioctl()` with an argument of an `inode` number of an executable file, and the file system begins to compress the read-critical data of the file in the background. The compression of executable file blocks is conducted during system idle periods (see Section 4.4).

4 Implementation

This section discusses how to implement the proposed FPC in a log-structured file system, F2FS, for mobile devices.

4.1 Dynamic Compression Window

A large compression window sufficiently populates a large dictionary for effective data compression. However, partial reads in a large chunk of compressed data would induce a high overhead because the compressed data must be read and decompressed as a whole. The selection of the compression window size is based on the following rules: The default size of the compression window is set to 4 KB to avoid an amplified read overhead. Exceptions are as follows: First, for foreground compression on SQLite journals, the compression window is large as the block write request. Because these files are barely read, using a large compression window has little effect on read performance. Second, background compression uses 32 pages (128 KB) as the compression window size for *.dex and *.odex, as mobile applications often performed bulk, sequential read on such executable files through *mmap()*. Since a page fault in Linux is handled by fetching a set of 32 pages into the page cache, using a compression window as large as the pre-paging size effectively improves the compression ratio without sacrificing the read performance. Due to the bulk reading, a chunk of compressed data of *.dex and *.odex files is allowed to be stored across physical block boundaries.

4.2 Sub-Block L2P Mapping

To better accommodate small file reads and writes of mobile applications, we enhance the mapping process to improve the read-write data compression efficacy. Because a compressed file system stores multiple compressed blocks in a physical block, it requires logical-to-physical (L2P) mapping at the sub-block level [2]. The existing L2P mapping for F2FS is managed at the block level using special node blocks `inode`. However, the current F2FS inode structure is 4 KB and it cannot accommodate all the extra metadata for sub-block indexing. To deal with this problem, our design appends extra bits to each mapping entry of an inode block: 3 bits for sub-block indexing and 1 bit as a compression flag. In the storage space, if a physical block contains compressed blocks, then the physical block is formatted into a header area and a compressed block area. A header entry is indexed by the sub-block number of an inode mapping entry, and the header entry contains a starting offset (16 bits) of the corresponding compressed block within the physical block.

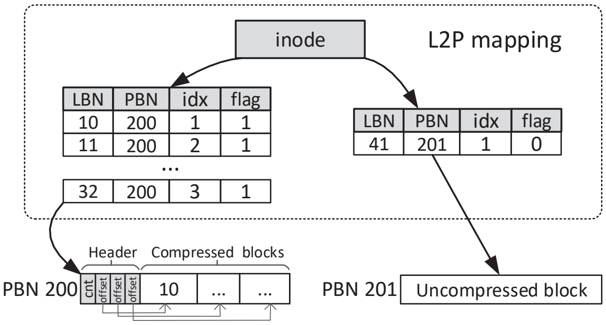
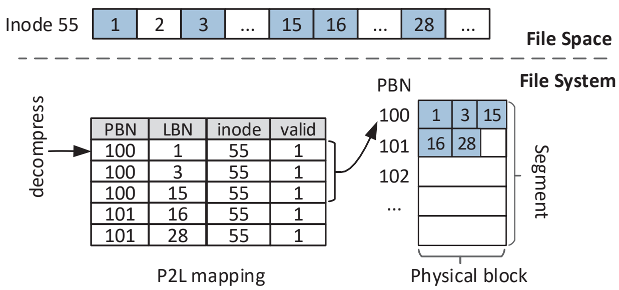


Figure 8: The extended L2P mapping.

Figure 8 shows an example of the extended L2P mapping and physical block layout. Suppose that a read of the tenth logical block (LBN 10) of a file will be served. The file system first locates the inode of the file and reads the LBN-to-PBN mapping entry for LBN 10. It identifies that the corresponding physical block PBN 200 is compressed (flag=1) and LBN 100 is the first compressed block (idx=1) in the physical block. After reading PBN 200, the file system loads the decompressed data of LBN 10 into the page cache and completes the read. It is possible that the header entries of multiple logical blocks refer to the same compressed block. To distinguish between logical blocks in the same compressed block, each header entry contains a logical block sequence number.

Since a physical block contains multiple compressed blocks, a physical block may be partially invalidated after write operations. We adopt a Block State Table (BST) to keep track of the valid/invalid status of compressed blocks in a physical block. The existing F2FS SIT (Segment Information Table) stores the valid/invalid status of data at the block level. Our BST is an extension to the SIT and is protected by the checkpoint mechanism. Let a physical block contain up to N compressed blocks. The BST extension uses a counter and a validity bitmap of the compressed blocks, which require $\log_2 N$ and N bits, respectively.

[2] In this paper, L2P mapping refers to the mapping of a file block offset (LBN within a file) to a storage address (PBN in storage).



PBN	LBN	inode	valid
100	1	55	1
100	3	55	1
100	15	55	1
101	16	55	1
101	28	55	1

Figure 9: Decompression for read-critical blocks with physical-to-logical (P2L) mapping.

4.3 Decompression with P2L Mapping

When decompressing a logical block, the file system requires the inode number and file offset of the block to properly load the block into the page cache address space. For a file block that is explicitly requested by a read operation, this information is known to the file system upon the request. However, as shown in Figure 8, when reading the physical block at PBN 200, the file system cannot find such information for the other two compressed blocks in the physical block. Therefore, the file system will have to ignore these two compressed blocks, although they have been brought into memory. Thanks to the reverse (physical-to-logical) mapping, a unique feature of log-structured file systems, our design can identify the inode and file offset of the other two compressed blocks and opportunistically load them into memory. The reverse mapping for F2FS is provided by the Segment Summary Area (SSA) [16]. The original purpose of SSA is for the space cleaning procedure to identify the valid/invalid status and logical block address of each block in a victim segment.

Figure 9 shows an example of decompressing blocks based on the P2L mapping. When the first logical block of the file (LBN 1) is requested, the file system reads the physical block at PBN 100, decompress the logical block of LBN 1, and loads it into the page cache. By consulting the reverse mapping, the file system also decompresses and loads the other two compressed blocks along with their inode number (55) and LBNs (3 and 15) into the page cache. Decompression with P2L mapping is essential to the read-critical data compression strategy because BC compacts and compresses correlated executable file blocks into the same physical block.

4.4 Logging and Cleaning

Data Separation. F2FS writes data of different hotness (write frequency) to separate segments through six logging heads to improve cleaning efficacy. FPC inherits these logging heads and employs three new logging heads: The first new logging head is for FC to write compressed read-write files, i.e., SQLite files. A node-data-combined compressed block (Section 3.2.3) is also written to this segment because the node block and data blocks of a rollback journal share the same lifetime. The second is for new, uncompressed executable files, and the third is used by BC to write compressed executable files. New executable files are written to the second new logging head without compression. During background compression, executable files are re-organized for read-critical data and then compressed into the third new logging head. FPC inherits the original F2FS victim selection policy with a slight enhancement. A segment of the largest number of invalid compressed blocks is selected as the victim for space cleaning. Once a victim segment is selected, only valid compressed blocks in the segment are copied for space cleaning.

On-Demand Background Compression. Since executable files do not change after installation or update, background compression can be executed on demand: As described in Section 3.3, a process running in the user space is responsible for collecting the information of read-critical blocks. The process begins to profile an application upon installation or update. For an application under profiling, the file read operations on its executable file are collected for at least 5 rounds of launching. When done profiling an application, the user process issues an `ioctl` call with the name of the executable file of the application to the file system for background compression. The background compression procedure is largely based on the data migration method of the existing segment cleaning procedure of F2FS, but it selects read-critical data for migration and compression. As reported in [26], the typical update period of mobile applications is half a month. In other words, the frequency of background compression will be very low, minimally impacting the file system performance and write traffic volume.

4.5 Design Summary

F2FS Modification. Our implementation of FPC requires enhancements of the F2FS core data structures. The enhancements are described below: 1) File indexing (L2P mapping) requires to augment each direct pointer in the inode and direct node with additional information, which now may refer to a compressed block in a physical block. An original direct pointer is of 32 bits, while a new direct pointer adds 1 bit for compression indication and 3 bits as an offset in a physical block (the compressed block number in a physical block is no larger than 8). The original inode structure has an array of 923 direct pointers, and our design replaces them with an array of 820 upgraded direct pointers. This design slightly reduces the largest file size from 3.94 TB to 3.50 TB. 2) Reverse (P2L) mapping requires to modify the Segment Summary Area (SSA). A physical block can contain multiple compressed blocks, so the structure `f2fs_summary` is extended to represent the reverse mapping information of each compressed block in a physical block. 3) Metadata-level compression requires one additional bit for each Node Address Table (NAT) entry to indicate whether or not a node block is compressed with data blocks into a physical block. Our design adds a separate bitmap to NAT for this purpose.

R/W pattern	File type	Compression policy	Compression type
write-intensive, random-write	db	compression on non-sequential blocks	FC for reduced write stress
(almost) write-only	db-journal	large compression window, metadata-level compression	
write-once, random-read	apk	critical data compaction and compression	BC for fast launching and space saving
write-once, seq-read	dex, odex	large compression window, across physical block boundary	

Figure 10: Summary of compression policies for different types of files and their read/write patterns.

4) The logging of compressed data requires three extra types of segment (Section 4.4). The segment numbers of the three active logging heads (along with those of existing logging heads) are kept in Checkpoint (CP). 5) Segment cleaning requires the valid/invalid status of data. This information is provided by the original Segment Information Table (SIT). If a physical block contains compressed blocks, our Block State Table (BST, Section 4.2) uses five bits to represent the invalid/invalid status of its compressed blocks and three bits for counting invalid compressed blocks.

Crash Consistency. F2FS employs the checkpoint scheme [16, 27] to maintain the data consistency in case of system crashes. The proposed compression solutions extend a set of core data structures including the summary blocks of the added compression logging heads, and *f2fs_inode* and *direct_node* node blocks. Since all the involved metadata extended in the compression designs belong to the data types that have to be protected by existing F2FS checkpoints, system consistency can be maintained by calling the recovery procedure of F2FS.

Fig. 10 is a summary of the access patterns of file types and their associated compression policy.

4.6 Overhead Analysis and Discussions

Space overhead. The primary space overhead of FC approach is related to the Block State Table (BST) for block state tracking. Besides in the storage space, a copy of the BST is made in memory for efficient access. For FC, the largest number of compressed blocks that a physical block can store is empirically set to 5 for a good balance between the metadata overhead and compression space reduction. Each BST entry for a physical block requires 8 extra bits (3 bits for a counter of invalid compressed blocks and 5 bits for a valid/invalid bitmap of compressed blocks). For a 16 GB storage space, the extra DRAM overhead for the counters and bitmaps is 4 MB, which is affordable to modern smartphones. On the other hand, for the metadata-level file compression, the Node Address Table (NAT) is enhanced to locate the compressed node block stored together with the compressed file blocks by adding a 1-bit flag for each NAT entry. As mentioned in Section 4.5, a new bitmap of the flag bits is added to the NAT, and the bitmap requires no more than 0.5 MB storage overhead for a 16 GB storage device. At last, the checksum stored in each physical block costs 4 bytes, and 16 GB storage space introduces a maximal 16 MB space overhead.

General Applicability. It is possible to generalize our pattern-guided compression for unknown file types. This is because the proposed method employs only a few simple properties of file access, including access sequentiality and read-write tendency, as shown in Fig. 10. The file system can adopt a profiling module to observe how files are accessed and apply proper compression strategies accordingly. In particular, the access pattern discovery mechanism is already part of the proposed BC design (Section 3.3.2), and the read-write tendency of new files can be monitored using counters.

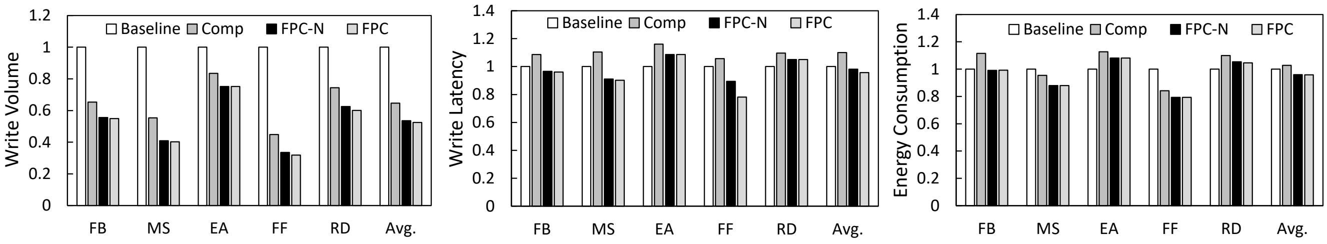
Compression Deployment. While compression of user data is feasible at the application level, this study focuses on a file-system approach because it enables deep integration with system-level mechanisms, e.g., prefetching of compressed blocks in the storage and compressing of file-system metadata, which are difficult at the application level.

5 Performance Evaluation

5.1 Experimental Setup

We implemented and evaluated the proposed FPC based on F2FS on a real platform Hikey 960 [28], which is an embedded development board for AOSP. The platform was equipped with a Kirin 960 8-core ARM processor, 4GB of RAM, and a 32GB UFS. The Android and Linux kernel versions were 9.0 and 4.9, respectively. FPC employed LZO [29] as the data compression algorithm. We configured F2FS in the `lfs` mode for using out-of-place updating only. This is because, in the current implementation, F2FS switches to in-place update when the space utilization is extremely high or when performing `fdatasync()` on small files. As mentioned previously, in-place updates risk severe internal fragmentation and thus we turned it off for the best effect of FPC. The system-level energy consumption was measured using the Monsoon power monitor [30].

FPC was evaluated using a set of popular mobile applications, including Facebook (FB), FB Messenger (MS), Google Earth (GE), Firefox(FF), Reddit (RD), Line (LN), Twitter (TW), Instagram (IG), Wechat (WC) and Chrome (CR). The evaluation of foreground compression (FC) was based on the first five applications as they produced a high volume of SQLite-related writes. The evaluation of background compression (BC) involved all the applications. The following methods were evaluated for performance comparison: 1) **Baseline**: The original F2FS without any compression. 2) **Comp**: F2FS with unconditional compression of incoming data. It employed a fixed compression window size of 4 KB and allowed only file blocks of sequential file offsets to share the same physical block. 3) **FPC-N**: The proposed FPC but without metadata-level compression. 4) **FPC**: The full-fledged version of the proposed approach (see Fig. 10). Currently,



(a) (b) (c)

Figure 11: Results of normalized (a) SQLite write volume (b) SQLite write latency and (c) system energy consumption using different compression approaches.

there is little choice of read-write compression file system readily available for Android devices. We implemented the core idea of existing compression file systems in F2FS for performance comparison. Specifically, Comp employed the sequential compression method (Section 3.2.1), which is adopted by Btrfs and JFFS2.

Table 2: Workload characteristics. The percentage reflects the contribution of SQLite files to the total write traffic.

	write count	avg. size	write contribution	compression ratio
FB	3215	39.2 KB	31.6%	0.39
MS	2597	32.4 KB	21.4%	0.25
EA	19919	55.2 KB	99.5%	0.65
FF	17695	33.2 KB	57.8%	0.12
RD	2658	30.5 KB	39.2%	0.40

Because the efficacy of foreground compression (FC) is primarily concerned with the compression ratio of incoming data, we employed content-accurate trace replay for performance evaluation of FC: First, we used each mobile application for 30 minutes and recorded their file operations on SQLite files at the VFS layer. The traces reflected highly common user scenarios, including viewing online news feeds (FB, FF, RD), viewing online satellite maps (EA), and sending/receiving text messages (MG). Each of the recorded operation consisted of an inode number, a file name, a write time, a file block offset, and the data content. The characteristics of the collected traces can be found in Table 2. Second, a user-level process was created to replay the file writes on a set of files with their original SQLite file names, and these writes were captured and compressed by foreground compression inside of F2FS.

We conducted experiments on background compression (BC) with the following steps: For each of the listed applications, we installed an application and performed five times of cold-start launching (with the page cache cleared). The read-critical data set of the application's executable files were identified by BC during the launching. After this, when the system was idle, an `ioctl` request was sent to BC to explicitly request compression of read-critical data in executable files.

5.2 Evaluation Results

This section presents the evaluation results of 1) foreground compression, including write volume, write latency, and energy consumption and 2) background compression, involving space requirement and application launching time.

Write Volume: Figure 11(a) shows the write-traffic volumes bound for SQLite files of Baseline, Comp, FPC-N, and FPC. All results are normalized to Baseline. Compared with Baseline, FPC greatly reduced the SQLite write volume by 47.5%, indicating that foreground compression was highly effective in terms of write stress reduction. In particular, during the 30-minute execution of Facebook, FPC reduced the SQLite write volume from 123.1 MB to 67.5 MB through foreground compression. FPC also outperformed Comp thanks to the use of a larger compression window on SQLite files and the storage of compressed file blocks having random file offsets in the same physical block. The improvement of FPC upon FPC-N depends is highly subject to the application scenario. Under the FF workload, FPC reduced the write volume by 5.3% by appending compressed node blocks of *.db-journal files to their associated compressed data blocks. Overall, the reduction in SQLite write volume contributed to a 26.1% reduction in the entire system write volume (from 3044.1 MB to 2250.9 MB). This large reduction is beneficial to the flash storage lifespan because the wear degree in flash memory is proportional to the volume of inbound write traffic.

Write Latency: Figure 11(b) shows that FPC reduced the write latency of SQLite files by 7.1% on average compared with Baseline. In other words, the benefit of a reduced write I/O count was larger than the cost of data compression. The only exception is Google Earth, whose SQLite files contain a large amount of incompressible multimedia data. Fortunately, many of the incompressible data were not selected for compression thanks to the compression ratio sampling technique mentioned in 3.2.2. By contrast, Comp suffered from the highest write latency because it compressed all incoming data, including those incompressible ones. To observe how high is the time overhead of data compression on the CPU, we measured the CPU utilization of all methods using the `TOP` utility during the trace replay. In our experiment, data compression was affiliated with a specific core. We observed that the average utilization of the involved core was between 3% and 8% under FPC, while that was between 1% and 2% under Baseline. In other words, data compression of FPC only marginally increased the CPU utilization.

Energy Consumption: Energy consumption is a critical

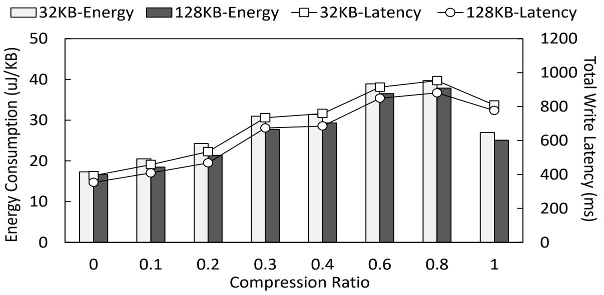


Figure 12: Sensitive study of write latency and energy consumption as compression ratio improves. Data are not compressed for compression ratio of 1.

concern for battery-powered mobile devices [31]. File compression takes extra CPU cycles but reduces flash storage writes. Although file compression induced an extra energy overhead, Figure 11(c) shows that FPC still achieved a lower energy consumption for most of the applications compared with Baseline. In particular, the energy-saving was larger when SQLite files were highly compressible, which was a common case among all applications. In particular, the largest write volume reduction for Firefox also led to the largest energy saving. FPC slightly increased the energy consumption of Google Earth. This is again because Google Earth stored incompressible multimedia contents in SQLite files and a small portion of them underwent ineffective data compression.

SQLite Journaling: In our experiments, although the system default for SQLite journaling was WAL, we found that the five applications (in Table 2) operated 23 out of 33 SQLite files in DELETE mode. In practice, many applications explicitly specify a journal mode to override the system default.

A rollback journal in DELETE mode is write-only except during crash recovery. However, rollback journals in PERSIST mode were reused, and reusing a rollback journal required to read the journal header block. However, due to page caching, the read barely reached the storage during successive transactions. For example, RD read only two blocks from its journal files in a 30-minute session. A similar result was also reported in [22]. The observation confirmed that rollback journals are (almost) write only.

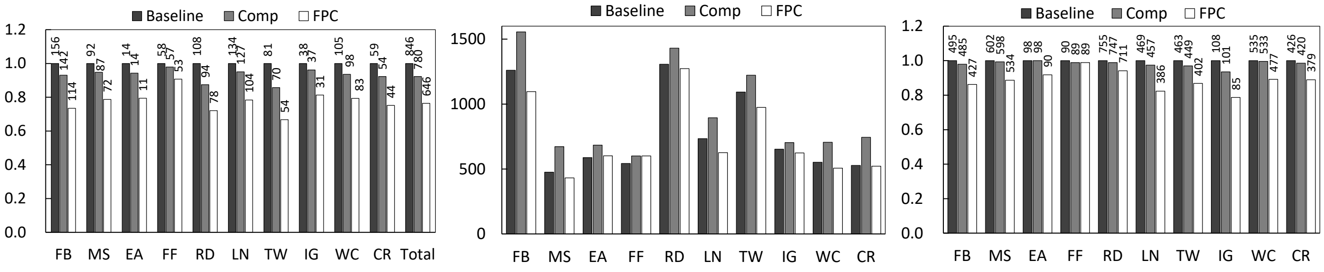
Effect of Compression Ratio: An experiment was conducted to understand the trade-off between the cost and benefit of compression as the compression ratio changes. We created a 100 MB file beforehand and then sequentially overwrote the entire file with large (128 KB) and small (32 KB) file writes. Each file write was followed by an `fsync()` operation. For each test, the data pattern was pre-generated to match the desired compression ratio. As shown in Figure 12, compared with the results without compression (compression ratio=1), the write latency and total energy consumption became lower when the compression ratios were not lower than 0.4 and 0.2, respectively. In other words, when data were highly compressible, compression benefited both write latency and energy consumption. The result indicates when data compress reasonably well, the benefit of our foreground compression design outweighs the cost of compression.

Space Requirement: Typically, executable files require more storage space than SQLite files. Figure 13(a) shows the executable file size of each application with the three methods. With FPC, the total executable file size of all applications was noticeably reduced from 846 MB to 646 MB (reduced by 23.7%). Interestingly, the size reduction of Facebook with FPC was 24%, much better than the reduction 8% achieved by unconditional compression Comp. The reason for the greater reduction of FPC is that background compression BC used a larger compression window on read-critical file blocks and allowed the compressed data to be stored across the physical block boundaries. By contrast, Comp always used a 4KB compression window and did not allow the physical block straddling, incurring a poor space efficacy.

App Launching Time: The application launching time shown here is the value reported by the activity manager `am` called by an `adb` command from a remote PC [32]. As shown in Figure 13(b), compared with Baseline, FPC improved the launching time of applications (except EA and FF, to be explained later), and the reduction was 5.2% on average. By contrast, applications with Comp launched even slower than with Baseline because Comp incurred a high cost of decompression under small, random reads of executable files. FPC significantly outperformed Comp by 22.3% on average in terms of the launching time. This is because our BC method identified read-critical data and then compacted/compressed them into a few physical blocks, leading to a fewer number of block read operations to launch applications. As Figure 13(c) shows, while Comp and Baseline required comparable numbers of block read requests to launch an application, FPC produced much fewer block read requests. In particular, compared with Baseline, FPC reduced the total block read count from 469 to 386, thus speeding up launching LN by 14.8%.

FPC marginally increased the launching time of EA and FF (2 out of the 10 applications). As Figure 13(c) shows, EA required a small number of block reads to launch. We also discerned that the launching of EA was CPU-intensive, and therefore the launching of EA did not much benefit from a reduced I/O count. FF is another case, for which the launching involved sequential reads mostly. Nevertheless, in summary, FPC successfully reduced the I/O cost through read-critical data compression, and the reduction concealed the time overhead of decompression and accelerated application launching.

We use *Decompression Amplification Ratio* (DAR), which has been defined in Section 2.2, to show how much non-read-critical share the same physical block with read-critical data. Table 3 shows the DAR values of the 10 applications. The DAR values of 8 out of the 10 applications were large than 2, indicating that more than one half of a physical block was occupied by compressed blocks of non-read-critical data. The DAR values of Firefox (FF) were very low, explaining why our FPC did not improve the launching time of FF in Fig-



(a) (b) (c)

Figure 13: Results of (a) executable file sizes (unit: MB). (b) application launching time (unit: ms). (c) block read number during application launching. The values of block read numbers are marked above each bar.

ure 13(b). Overall, most of the profiled applications had high DAR values, and therefore we believe that small, random reads on executable files are a common problem of mobile applications. Provided that the launching process of an application is not CPU-intensive, the proposed FPC approach can help accelerate the launching process.

Table 3: DAR values of 10 mobile apps.

	FB	MS	EA	FF	RD	LN	TW	CR	IG	WC
DAR	3.8	3.1	2.5	1	2.1	2.6	2.7	3.2	3.7	1.4

6 Related Work

Host Level Compression: In-place update file systems with compression support, such as JFFS2 [11], NTFS [33], and compressed ext2 [34], could suffer from a low space efficacy if the new compressed data size was larger than the old version. Burrows et al. [19] proposed an on-line data compression approach by leveraging the out-of-place write behaviors of a log-structured file system to avoid the above problem. Compression techniques were also recommended in log-structured systems, e.g., enterprise storage system Purity [20] and the database engine Rose [35]. Btrfs [10] was a B-tree file system with compression support. It sequentially compressed incoming data in the fixed granularity upon file updates and then wrote the compressed blocks to a new extent. However, the above studies paid little attention to the compression optimization for small file writes, which could incur a poor compression efficacy for mobile systems.

To reduce read amplification for mobile applications, Zhang et al. [13] proposed a compression framework based on the FUSE file system which compressed read-only files only and required additional decompression hardware. A compressed read-only file system (EROFS) was introduced to save storage space and improve read performance [12]. It leveraged the fixed-sized output compression to reduce read amplification, but only sequentially compressed data without paying attention to the unfriendly random reads of mobile applications that could degrade the decompression efficacy. FPC exploited the compression-friendly structures of log-structured file systems. More importantly, FPC took advantage of the unique file access behaviors of mobile devices and addressed the impact of decompression on user-perceived latencies.

Device Level Compression: There are several solutions for device compression. Zhang et al. [36] proposed a device-side in-place delta compression technique to reduce write stress on SLC-mode flash blocks. Ji et al. [23] proposed a firmware-based compression that selectively compressed data in eMMC devices. Several enterprise storage system vendors including Nimble [37] and Pure Storage [38] had announced their compression-enabled enterprise storage devices. Nevertheless, unconditional device-side data compression is not aware of much useful host information, e.g., the critical reads associated with executable files, and hence it is difficult to optimize decompression latency and improve user experience. Another critical problem of device compression is that recent Android versions have been equipped with block encryption [39], which renders the encrypted data uncompressible.

7 Conclusion

This paper proposed FPC, a file access pattern guided compression framework, to reduce write stress and save storage space for mobile devices. First, the compression was performed at the foreground to selectively compress write-mostly, highly compressible files that produced many small data updates to reduce write stress. Second, background compression re-grouped and compressed critical blocks in executable files to reduce the application launching latency and improve space utilization. The proposed FPC approach was implemented on a real mobile device and experimental results showed both the write traffic and the executable file size was substantially reduced. FPC also reduced the application launching time.

Acknowledgement

We would like to thank our shepherd Youjip Won and the anonymous reviewers for their valuable comments and guidance. This work is partially supported by the Natural Science Foundation of Jiangsu Province (BK20200462), Ministry of Science and Technology of Taiwan (107-2628-E-009-002-MY3 and 109-2221-E-009-075), Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CityU 11204718 and 11218720), NSFC 62072177, and Shanghai S&T Project (20ZR1417200).

References

[1] Smartphone os market share. https://www.idc.com/promo/smartphone-market-share/, 2018.

[2] Umar Farooq and Zhijia Zhao. Runtimedroid: Restarting-free runtime change handling for android apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'18)*, pages 110–122. ACM, 2018.

[3] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8(4), 2012.

[4] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. WALDIO: eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proceedings of USENIX ATC*, pages 235–247, 2015.

[5] Cheng Ji, Riwei Pan, Li-Pin Chang, Liang Shi, Zongwei Zhu, Yu Liang, Tei-Wei Kuo, and Jason Chun Xue. Inspection and characterization of app file usage in mobile devices. *ACM Transactions on Storage (TOS)*, 16(4), 2020.

[6] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *Proceedings of ATC, 2013*, pages 309–320, 2013.

[7] Younghwan Go, Nitin Agrawal, Akshat Aranya, and Cristian Ungureanu. Reliable, consistent, and efficient data sync for mobile apps. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST 15)*, pages 359–372, 2015.

[8] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of Annual Technical Conference (USENIX ATC 17)*, pages 759–771. USENIX Association, 2017.

[9] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. Fasttrack: Foreground app-aware I/O management for improving user experience of android smartphones. In *Proceedings of Annual Technical Conference (USENIX ATC 18)*, pages 15–28. USENIX Association, 2018.

[10] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[11] Jffs2. http://www.linux-mtd.infradead.org/doc/jffs2.html.

[12] Xiang Gao, Mingkai Dong, Xie Miao, Wei Du, Chao Yu, and Haibo Chen. EROFS: A compression-friendly read-only file system for resource-scarce device. In *Proceedings of Annual Technical Conference (USENIX ATC)*. USENIX Association, 2019.

[13] Xuebin Zhang, Jiangpeng Li, Hao Wang, Danni Xiong, Jerry Qu, Hyunsuk Shin, Jung Pill Kim, and Tong Zhang. Realizing transparent os/apps compression in mobile devices at zero latency overhead. *IEEE Transactions on Computers*, 66(7):1188–1199, 2017.

[14] Kisung Lee and Youjip Won. Smart layers and dumb result: IO characterization of an android-based smartphone. In *Proceedings of the tenth ACM international conference on Embedded software (EMSOFT)*, pages 23–32. ACM, 2012.

[15] S. Bhattacharya A. Dilger A. Tomas A. Mathur, M. Cao and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, pages 21–33. Citeseer, 2007.

[16] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[17] Ming-Chang Yang, Yuan-Hao Chang, Chei-Wei Tsao, and Chung-Yu Liu. Utilization-aware self-tuning design for TLC flash storage devices. *IEEE Trans. VLSI Syst.*, 24(10):3132–3144, 2016.

[18] Samsung Semiconductors. 3D TLC NAND to beat MLC as top flash storage. EETimes, 2015.

[19] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In *ASPLOS*, pages 2–9. Citeseer, 1992.

[20] 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*, pages 1683–1694. ACM, 2015.

[21] M. Son, J. Ahn, and S. Yoo. Nonvolatile write buffer-based journaling bypass for storage write reduction in mobile devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(9):1747–1759, 2018.

[22] Taeho Hwang, Myungsik Kim, Seongjin Lee, and Youjip Won. On the I/O characteristics of the mobile web browsers. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC'18)*, pages 964–966, 2018.

[23] Cheng Ji, Li-Pin Chang, Liang Shi, Congming Gao, Chao Wu, Yuangang Wang, and Chun Jason Xue. Lightweight data compression for mobile flash storage. *ACM Trans. Embed. Comput. Syst.*, (5s):183:1–183:18, 2017.

[24] Danny Harnik, Ronen Kat, Dmitry Sotnikov, Avishay Traeger, and Oded Margalit. To zip or not to zip: Effective resource usage for real-time compression. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST 13)*, pages 229–241, 2013.

[25] Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G Shin. Fast: Quick application launch on solid-state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 259–272, 2011.

[26] Dmitry Garbar. How often should you update your mobile app? https://www.apptentive.com/blog/2018/12/27/how-often-should-you-update-your-mobile-app/, 2018.

[27] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 83–98, 2016.

[28] Hikey 960. https://www.96boards.org/product/hikey960/.

[29] LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo/.

[30] Monsoon power monitor. http://www.msoon.com/LabEquipment/PowerMonitor/, 2016.

[31] Jayashree Mohan, Dhathri Purohith, Matthew Halpern, Vijay Chidambaram, and Vijay Janapa Reddi. Storage on your smartphone uses more energy than you think. In *Proceedings of HotStorage*. USENIX Association, 2017.

[32] Android debug bridge (adb). https://developer.android.com/studio/command-line/adb.html.

[33] Ntfs compressed files. http://www.ntfs.com/ntfs-compressed.htm.

[34] e2compr. http://e2compr.sourceforge.net/.

[35] Russell Sears, Mark Callaghan, and Eric Brewer. Rose: Compressed, log-structured replication. *Proceedings of the VLDB Endowment*, 1(1):526–537, 2008.

[36] Xuebin Zhang, Jiangpeng Li, Hao Wang, Kai Zhao, and Tong Zhang. Reducing solid-state storage device write stress through opportunistic in-place delta compression. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST 16)*, pages 111–124, 2016.

[37] Casl architecture in nimble storage. http://www.nimblestorage.com/products/architecture.

[38] Flashreduce data reduction in pure storage. http://www.purestorage.com/flash-array/flashreduce.html.

[39] Android Encryption. https://source.android.com/security/encryption/.

ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory

Kai Wu
kwu42@ucmerced.edu

Jie Ren
jren6@ucmerced.edu

Ivy Peng†
peng8@llnl.gov

Dong Li
dli35@ucmerced.edu

University of California, Merced *Lawrence Livermore National Laboratory*†

Abstract

Failure-atomic transactions are a critical mechanism for accessing and manipulating data on persistent memory (PM) with crash consistency. We identify that small random writes in metadata modifications and locality-oblivious memory allocation in traditional PM transaction systems mismatch PM architecture. We present ArchTM, a PM transaction system based on two design principles: avoiding small writes and encouraging sequential writes. ArchTM is a variant of copy-on-write (CoW) system to reduce write traffic to PM. Unlike conventional CoW schemes, ArchTM reduces metadata modifications through a scalable lookup table on DRAM. ArchTM introduces an annotation mechanism to ensure crash consistency and a locality-aware data path in memory allocation to increases coalesable writes inside PM devices. We evaluate ArchTM against four state-of-the-art transaction systems (one in PMDK [30], Romulus [21], DUDETM [46], and one from Oracle [50]). ArchTM outperforms the competitor systems by 58x, 5x, 3x and 7x on average, using micro-benchmarks and real-world workloads on real PM.

1 Introduction

Byte-addressable persistent memory (PM) can provide DRAM-like performance and storage-class capacity. The state-of-the-art Intel Optane DC PM could implement up to nine terabytes memory capacity on a single machine with latency in hundreds of *ns* [31, 32, 34, 57, 72]. Such high-performance PM is emerging in datacenters and clouds to boost performance-critical data-intensive applications, such as database [9,17,22,28,41,65] and graph workloads [18,25].

Crash consistency is a primary challenge in using PM. With PM, programs can recover their persistent data on PM even in the event of crashes. However, such a recovery requires a guarantee that persistent data is in a consistent state, a requirement referred as the crash consistency guarantee. Failure-atomic transactions are a popular mechanism to ensure crash consistency. Extensive studies [16,21,27,30,39,40,49–51,61,67,69,70,73] have proposed various transaction mechanisms that generally employ logging-based (undo or redo logging) or Copy-on-Write (CoW)-based designs.

Existing works optimize PM transactions by reducing data copying [11,20,51,68] or persistence overhead [20,35,38,43, 56,62]. They emulate PM based on DRAM with increased memory latency or reduced bandwidth, but miss PM architecture details. In this study, we focus on the implications of PM architecture on transaction performance. Our performance analysis on state-of-the-art PM transaction systems identifies that the PM micro-architecture, such as internal buffers and data block size, has significant impacts on transaction performance. The mismatch between the transaction implementation and PM architecture can cause 3x-58x slowdown, compared to an architecture-aware implementation.

Performance characterization of PM architecture leads us to rethink the design of PM transactions. Logging-based transactions have a double write problem because of creating logs and updating data in-place. The excessive writes to PM mismatch with poor write performance on PM. CoW-based transactions avoid this problem, but suffers from performance overhead due to metadata updates, which causes many small writes misaligned with PM internal block size.

Therefore, high-performance PM transactions call for new design principles tailored to the characteristics of the emerging PM architecture, which is distinctive from conventional block devices and more than just a slower DRAM. We introduce two design principles customized to PM architecture.

- Avoid small (less than 256 bytes) writes to PM. Small writes in PM suffer from write amplification because data in a small write must be aligned with the internal write block size (256 bytes) in PM, which wastes memory bandwidth and delays transactions. Our characterization study reveals that in state-of-the-art PM transaction systems (one in PMDK [30], Romulus [21], DUDETM [46], and an Oracle transaction system [50]), more than 78% of data objects are smaller than 64 bytes, when the transaction systems perform write operations on 512-byte persistent objects. The main source of those small data objects comes from metadata for transaction runtime state, memory allocation and object mapping.

- Encourage coalescable writes. Sequential write performs much faster than random write on PM (e.g., for 64-byte writes, sequential write is 3.7x faster than random write). Multiple sequential writes can be coalesced in an internal buffer of Optane, enabling high performance.

We follow the above principles in ArchTM. ArchTM uses a CoW-like design to avoid the double write problem in logging-based transactions. To avoid small writes, ArchTM stores metadata of memory allocator and data objects on DRAM to reduce frequent small random writes to PM. However, such a design suffers from a fundamental tradeoff between performance and crash consistency. In particular, metadata on DRAM, although leading to high transaction performance can be lost when a crash happens, leading to a problem of identifying crash consistency of data objects.

The above problem is caused by the fact that metadata is the only connection between the transaction state and data objects for crash recovery. Such a connection is not PM-oriented. Removing it causes isolation between transaction state and data objects. To address this challenge, ArchTM introduces a lightweight annotation mechanism. This mechanism adds data object metadata (object ID and size) and transaction ID into the data object, and adds transaction ID into the transaction metadata (i.e., the transaction state variable). The transaction ID is persistent and sets up an alternative connection between data objects and the transaction state. Using the transaction ID, the data object ID and size, ArchTM can easily locate data objects and identify their crash consistency after a crash.

To encourage coalescable writes, ArchTM makes best efforts to allow consecutive memory allocation requests to get contiguous memory allocations. This strategy is based on the observation that in a transaction, data objects that are allocated consecutively are likely to be updated together. For example, in a key-value store system, memory allocation requests for a key data object and a value data object associated with the key often happen together. Writes to the key and value data objects happen in sequential and continuous order. Hence, allocating the key and value contiguously in the address space likely results in coalescable write.

However, to implement the above strategy, we must re-examine the traditional wisdom for memory allocation. The existing memory allocators typically use multiple free lists for each thread. Each free list supports allocation requests for specific sizes. Such size-class-based memory allocation is used to reduce memory fragmentation. However, it allocates noncontiguous memory blocks to consecutive memory allocation requests if they are fulfilled by multiple free lists. Hence, there is a fundamental tradeoff between *allocation locality* and memory fragmentation.

To break this tradeoff and encourage coalescable writes, ArchTM uses a single free list and a lightweight online defragmentation mechanism. In particular, ArchTM supports locality-aware data path using the single free list for allocation and uses a recycle list to collect and merge freed memory blocks. For defragmentation, ArchTM aggregates data objects in highly fragmented memory regions to create large and contiguous memory blocks.

In summary, the paper makes the following contributions:

- We reveal the performance characterization of realistic PM hardware and pinpoint the performance problems in the representative PM transactions. Such problems are caused by the negligence of the characteristics of PM architecture in traditional PM transaction designs.
- We identify two fundamental tradeoffs to enable high performance PM transactions. We introduce a new PM transaction design, ArchTM, customized to the PM architecture and breaking the tradeoffs.
- ArchTM beats state-of-art PM transaction systems PMDK, Romulus, DUDETM and the Oracle system by 58x, 5x, 3x and 7x on average, using micro-benchmarks and real-world workloads on PM hardware.

2 Background

2.1 Persistent Memory Transactions

Failure-atomic transactions are a common solution to ensure crash consistency on PM [16, 21, 27, 30, 39, 40, 49–51, 61, 67, 69, 70, 73]. Updates in a failure-atomic transaction either all succeed or fail, leaving the data on PM in a consistent state. We refer to data objects accessed in a transaction as *persistent objects*. PM transactions are implemented in two major paradigms – logging and copy-on-write (CoW).

Logging-based transactions can use either *undo-logging* or *redo-logging*. Both logging approaches must write twice to update a persistent object, i.e., update the log and then the data (Figure 1 a and b). This in-place update to the data could cause concurrent random writes because transactional workloads could update arbitrary persistent objects.

CoW-based transactions create a new copy of a persistent object before modifying it (Figure 1 c). All updates are captured in the new copy, i.e., out-place updates. After persisting updates in the new copy, the system updates the pointer to the persistent object to the new copy and discards the old copy. Hence, CoW transactions write to PM only once. Even when random persistent objects are updated, persisting their new copies laid out sequentially still result in sequential writes.

2.2 Memory Management in PM Transactions

In logging or CoW paradigms, logs are inserted and removed, or copies of persistent objects are created and deleted in each transaction. Frequent memory allocation and deallocation in concurrent transactions require **scalable** solutions. Also, the

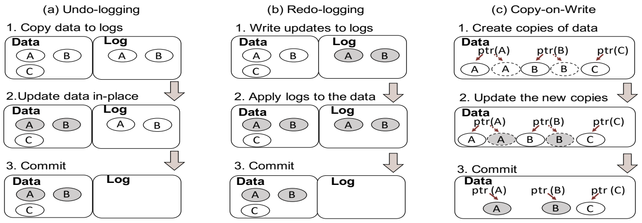


Figure 1: Three transaction implementations: undo-logging, redo-logging, and copy-on-write.

persistence in PM imposes unique requirements of **consistency** and **low fragmentation** on memory management.

Scalable memory allocators [14, 23, 26, 64], including state-of-the-art PM allocators [15, 30, 66], typically implement thread-local free lists and global free lists. An allocation request is first tried on the requester thread's local free list before being forwarded to the global free list. For a deallocation request, the freed memory block is added to the requestor thread's local free list to avoid synchronization on the global free list. The existing memory allocators usually predefine a set of object size classes. For each size class, the allocator maintains a list of free memory blocks of that size. An allocation request is fulfilled by the list in the nearest size class. Memory fragmentation occurs when the selected size class is larger than the requested size. Unlike volatile memory, fragmentation on PM has a longer-lasting impact. Volatile memory may restart the program to diminish fragmentation while fragmentation on PM persists through restarts. Besides, a PM allocator needs to ensure its metadata in a consistent state to avoid data loss and memory leakage after crash.

2.3 Emerging PM Architecture

Emerging persistent memories are byte-addressable, and PM DIMMs are attached directly to the memory bus, like conventional DRAM DIMMs. Processors can access PM through `load` and `store` instructions. The Intel Optane DC PM represents state-of-the-art PM hardware [34, 57, 58, 71]. The data transfer between the processor and PM occurs at the cache line granularity (64 bytes). The Optane internal transactions, however, have a granularity of 256 bytes. *Write amplification* occurs as a result of the two mismatched transaction sizes. For instance, updating a cache line (64 bytes) could result in a 256-byte write inside the Optane media. A *combining buffer* of 16KB [34] sits inside each NVDIMM to coalesce writes. Multiple writes from the processor could be combined into a single transaction if they occupy a contiguous 256-byte block.

3 Performance Characterization

We study the performance of PM transactions and Optane PM to gain insights for our design.

3.1 Transaction Performance Study

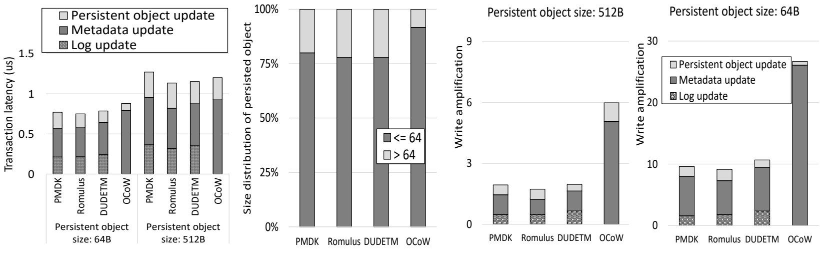
We study four representative PM transaction systems: PMDK [30], Romulus [21], DUDETM [46], and one from Oracle [50]. PMDK uses undo-logging, Romulus and DUDETM use redo-logging, and the Oracle system (denoted as *OCoW*) uses CoW. The specification of our Optane platform is in Section 6. We focus on write operations because they are the most expensive transaction operation, and writes to PM are expensive. A write operation in a transaction needs to update persistent object, log (if logging-based), and metadata. Figure 2a shows the latency breakdown of a write operation in PM transactions. We report the performance on small (64-byte) and large (512-byte) persistent objects. The figure shows that most time is spent on log updates or metadata updates.

We instrument the APIs used to persist data objects (e.g., *pmemobj_persist*() in PMDK) to study the performance of write operations. The APIs use the starting address and size of the data objects as input. Figure 2b reports the distribution of the *persisted* data size in transactions that perform write operations on 512-byte persistent objects. The figure reveals that more than 78% of persisted objects are smaller than 64 bytes, i.e., a lot of small writes on PM. Furthermore, we study write amplification, quantified as the ratio between write traffic in PM measured by performance counters and the number of bytes modified by transactions. Figure 2c reports the write amplification in transactions that perform write operations on 64- and 512-byte persistent objects. All systems exhibit write amplification, inflating PM write traffic by 1.8x - 27x.

Performance analysis. *We find that the metadata updates are the primary source of small writes.* In general, transaction systems have four types of metadata: metadata for transaction runtime, metadata for memory allocation, log metadata, and metadata for persistent objects. Metadata for transaction runtime records transaction status, e.g., COMMIT or ABORT, and transaction IDs. Metadata for memory allocation has information about memory consumption. Log metadata has information on logs (e.g., the indexing of log records), and is unique in logging-based transactions. Metadata for persistent objects store pointers to the new or old copy of persistent objects, and is unique in CoW-based transactions. By design, CoW-based systems have more metadata updates than logging-based ones. For instance, OCoW has about 270% more metadata updates than the other three logging-based systems. For each update, a CoW-based transaction must allocate a new data copy, remap pointers to the data, and deallocate the old data copy. This process generates frequent small writes to metadata for memory allocation and persistent objects.

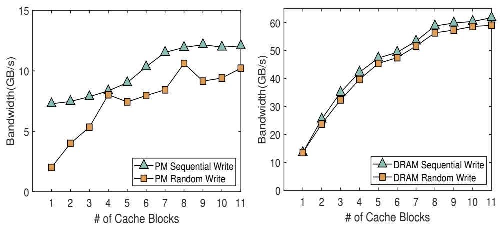
3.2 Performance Study of PM Writes

We study the write performance on Optane DC PM using a microbenchmark that performs random and sequential writes. Each write is followed by cache line flushes to persist to PM.



(a) Latency breakdown **(b)** Size distribution **(c)** Write amplification

Figure 2: Performance characterization of write operations in PM transactions.



(a) PM **(b)** DRAM

Figure 3: Sequential and random write bandwidth at different write sizes on PM and DRAM.

Various write sizes, ranging from one to 11 cache lines, are tested. Figure 3 reports the bandwidth of performing 100M writes using 24 threads on PM and DRAM. We have the following observations and insights for high-performance PM transactions.

Figures 3a and 3b show that write bandwidth of PM is significantly lower than that of DRAM. On our system, write bandwidth to DRAM reaches 80 GB/s but only 13 GB/s to Optane PM. Furthermore, on Optane PM, the peak write bandwidth is 13 GB/s, 3x lower than the peak read bandwidth. These results are consistent with the existing work [34]. Hence, **reducing write traffic on PM is critical** for high-performance transactions. The logging-based transaction systems need to write data twice to update a persistent object, which causes excessive write traffic.

Figure 3a shows that small random writes on PM perform worse than sequential writes. When writing only 64 bytes (Figure 3a), random write merely achieves 25% of the bandwidth of sequential write. This performance gap is caused by the 256-byte Optane internal granularity and write amplification, and the gap reduces when the write size increases. The logging-based transactions update persistent objects in-place. This could result in random writes, because persistent objects in a transaction can be randomly distributed on PM. Using out-place updates, as in CoW-based transactions, can enable sequential writes because the new copies of persistent objects are manageable and can be laid out contiguously in PM.

Figure 3a shows that the random writes on PM have performance spikes at write sizes that are a multiple of 256 bytes, e.g., four and eight cache lines. In contrast, random writes on DRAM (Figure 3b) exhibits no such pattern. Such performance on PM is due to the effect of the write combining buffer. It buffers and combines 64-bytes stores into a 256-byte internal store. Small simultaneous writes to contiguous address space are more likely to be combined into one internal store than small writes to arbitrary addresses. Therefore, increasing the probability of concurrent writes to contiguous address space can increase the opportunity to **leverage the combining buffer** hardware to coalesce writes inside the PM.

4 Design Principles and Major Techniques

Driven by the performance characterization and analysis of existing PM transactions and PM, we introduce two design principles and five techniques in ArchTM for high-performance architecture-aware transactions.

- **Avoid small writes on PM.**
  (1) Logless. ArchTM favors the CoW mechanism to reduce write traffic to PM.
  (2) Minimize metadata modifications on PM with guaranteed crash consistency. ArchTM keeps transient metadata on DRAM to avoid frequent metadata modifications on PM. Also, ArchTM introduces an annotation mechanism to connect the persistent transaction state with data objects. From the transaction state of data objects, ArchTM can detect the consistency of data on PM and recover from a crash.
  (3) Scalable persistent object referencing. ArchTM uses a scalable object lookup table on DRAM to quickly locate the latest copies of persistent objects in concurrent transactions.
- **Encourage coalescable writes**.
  (4) Consecutive allocation requests get contiguous memory blocks. ArchTM supports a locality-aware data path for small memory allocations to encourage sequential writes in transactions.
  (5) Avoid memory fragmentation. ArchTM employs a lightweight online memory defragmentation technique that examines memory usage by regions and reduces fragmentation on PM.

4.1 Logless

ArchTM employs a CoW-like mechanism to reduce write traffic to PM. Upon an update request, ArchTM creates a new copy of the persistent object and applies updates to the new copy. The out-of-place update in CoW reduces the number of PM writes. When committing the new copy to PM, consecutive writes into contiguous memory addresses increase the possibility of writes coalesced at the combining buffer. However, naively adopting CoW incurs excessive metadata updates on PM due to object remapping and allocation management (Section 3.1). We address this challenge by maintaining metadata on DRAM.

4.2 Minimize Metadata Modification on PM

ArchTM places the memory allocation metadata on DRAM. It does not record memory allocation and reclamation into logs on PM as in previous PM transaction systems [19, 21, 30, 66, 69]. Also, ArchTM avoids modifying the persistent object metadata on PM by using an *object lookup table* on DRAM. This lookup table is used to locate the latest copy of a persistent object quickly. Existing CoW-based implementations [50] must modify the persistent object metadata on PM to update the pointer to the object to the new copy (Figure 1.c). With these metadata in DRAM, ArchTM reduces small PM writes and accelerates the lookup, but cannot ensure crash consistency. ArchTM introduces an annotation mechanism to guarantee crash consistency.

Annotation. ArchTM annotates a transaction by adding a transaction ID into the transaction metadata (the transaction state variable). The embedded transaction ID is persisted immediately when the transaction state changes to *start*. ArchTM also annotates a persistent object by adding the object information, i.e., object ID, object size, and transaction ID, into the object header on PM when the object is created. During the recovery from a crash, ArchTM uses the object ID and size to identify each persistent object on PM. Then, ArchTM uses the annotated transaction ID to identify the most recent copy of a persistent object, recycle the stale copies, and discard uncommitted modifications.

4.3 Scalable Object Referencing

ArchTM uses an object lookup table to find the critical information, such as the location of the latest copy of a persistent object. The table is indexed by persistent object IDs. When a persistent object is allocated, the allocator thread gets an object ID and populates the corresponding entry in the lookup table. Multiple threads can reference persistent objects from the table concurrently and efficiently because DRAM supports higher bandwidth than PM.

The object lookup table is essential for high-performance transactions. Compared to decentralized object referencing [40, 50], the object lookup table in ArchTM resides on a contiguous DRAM space, which brings convenience for management (e.g., checkpointing) and migration. If the DRAM space is insufficient to store the whole lookup table, the spilling part of the table is placed on PM. Compared with general concurrent index data structures, such as hash tables, our object lookup table is easy to implement and has no synchronization overhead. The competition between threads to get an entry from the lookup table cannot happen, because threads are assigned with disjoint sets of object IDs and hence update disjoint sets of table entries. The object lookup table can find the object metadata in one step because it uses the object ID as the index of the table, which differs from other indexes (e.g., hash table and B-trees) that require additional calculations or queries to find object metadata.

4.4 Contiguous Memory Allocations

ArchTM customizes memory allocation and reclamation for transactional workloads on PM to maximize the possibility of sequential writes. Small allocations are the main optimization focus because sequential writes benefit small objects more than large objects (See Figure 3a). In ArchTM, there are two data paths for persistent object allocation and reclamation: (1) a regular data path for large allocations and reclamations, similar to existing allocators like JEMalloc [23]; and (2) a locality-aware data path for small allocations. The latter optimizes through a single free list and global recycling procedure.

A single free list is used in ArchTM for allocating objects of various sizes. Existing approaches [14, 15, 23, 26, 30, 64, 66] use multiple free lists, each for a different allocation size. Multiple free lists could cause consecutive allocation requests of different sizes to go to different free lists. Consequently, those requests get noncontiguous memory allocations, and writing to them leads to nonsequential writes to PM. Instead, using a single list of freed segments in sorted order would encourage consecutive requests to get sequential allocations. To maximize concurrency, ArchTM assigns each thread with a dedicated portion from the global free list (Section 5.1).

Recycle and merge memory blocks globally. Current approaches [14, 15, 23, 30, 64, 66] return freed memory blocks to thread-local free lists directly. This procedure avoids synchronization on managing a global free list but may harm the locality of freed memory blocks. Free memory blocks in a free list may be noncontiguous so that consecutive allocation requests get noncontiguous allocations. ArchTM runs a helper thread to collect and merge freed blocks from threads. These freed blocks are sorted and merged into a global recycle list before returning them to the global free list. The global recycling procedure does not happen in the critical path and does not affect the efficiency of memory deallocation.

4.5 Reduce Memory Fragmentation

Using a single free list for various allocation sizes could result in memory fragmentation. ArchTM uses a 64-byte size class in the memory allocator. An allocation smaller than the size class gets rounded up. We choose this size class to avoid false sharing in cache lines.

ArchTM introduces an online defragmentation mechanism to reduce memory fragmentation. The mechanism monitors the memory usage of the persistent object pool in the background to identify underutilized memory regions. During the memory allocation, this mechanism dynamically aggregates

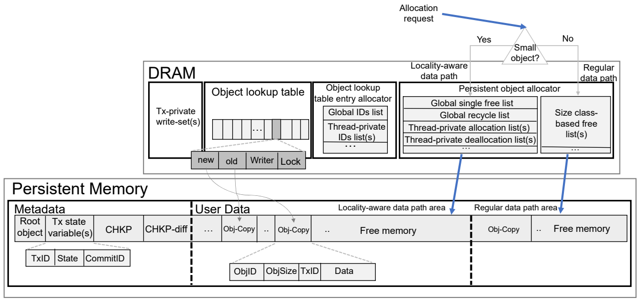


Figure 4: Major data structures in ArchTM

persistent objects distributed in the underutilized memory regions to improve memory usage. The online defragmentation mechanism is a user-space solution that can be enabled or disabled. It requires no modifications to operating systems as required by existing solutions [52]. Also, the user-space solution is more flexible than offline static solutions [59] and can react to changes in the application during execution.

5 ArchTM Implementation

We describe our implementation based on Section 4.

5.1 Data Structures

Persistent Data Structures on PM. ArchTM maintains a persistent memory pool partitioned into metadata and user data areas. As depicted in Figure 4, the metadata area stores a root object, a list of transaction state variables, a checkpoint field (*CHKP*), and a checkpoint-diff field (*CHKP-diff*).

The list of transaction state variables records the state of each ongoing transaction. Each variable encodes transaction state and ID, and commit ID. We use the transaction start timestamp as transaction ID, and the transaction commit timestamp as commit ID. They are global timestamps captured at the beginning and end of a transaction. ArchTM uses hardware clock (*rdtscp* in x86 architectures [6, 36]) and prevents the constant skew of the hardware clock among processors by the ORDO primitive [36] to ensure correct ordering of transactions. The transaction state indicates the progress of a transaction, e.g., BEGIN, COMMITTED, END or ABORT.

CHKP stores a persistent checkpoint of the object lookup table to speedup recovery (Section 5.6). *CHKP-diff* records the list of memory blocks (named *memory segments*) pre-allocated to each thread (Section 5.4-Allocation). *CHKP-diff* is useful to track working objects before the next checkpoint. It is implemented as an array of elements containing three fields: ID of ongoing transactions for which the segment is fetched, the segment start address and size.

The user data area stores persistent objects. Each object has an object header and data. The header contains object ID and size, and transaction ID. The user data area is divided into a regular data path area for large object allocations and a locality-aware data path area for small object allocations.

Transient Data Structures on DRAM ArchTM maintains an object lookup table and a hash set per transaction. The object lookup table is a one-dimensional array mapping a persistent object ID to a persistent object on PM. Each PM object has an entry in the table. An entry has four fields, i.e., a pointer to the latest copy (*new*), a pointer to the old copy (*old*), a variable (named *writer*) storing the pointer of the transaction state variable of the ongoing transaction that modifies the latest copy, and a write lock associated with the *writer* to coordinate parallel transactions. The hash set (named *write-set*) is used to collect the IDs of all persistent objects modified by a thread in an active transaction. Before committing a transaction, all objects in the hash set must be persisted.

ArchTM manages metadata for two allocators on DRAM. The first allocator allocates an entry in the object lookup table when a persistent object is created. This allocator maintains a list of free IDs for persistent objects (named *ID list*) per core. A persistent object ID is the index of an entry in the object lookup table. When the allocator allocates an entry, it gets an object ID from the ID list. When a persistent object is freed, its object ID is returned to the ID list. We reuse IDs for persistent objects to avoid the explosion of IDs. New IDs are created only when the ID list is empty.

The second allocator allocates persistent objects. It reuses the metadata structures in JEMalloc [23] for the regular data path but adds significant extensions to optimize small writes to PM (Section 4). For the locality-aware data path, ArchTM maintains a global free list and a global recycle list. The global free list contains memory blocks available for allocations. To ensure sequentially when multiple threads access the global free list, ArchTM uses a write lock on the global free list. To mitigate contention on the global free list, each thread maintains a thread-private allocation list, which is a portion from the global free list. Only when a thread exhausts its allocation list will the thread access the global free list to get a new portion. Therefore, synchronization on the global free list is infrequent. The global recycle list collects memory blocks freed by all threads. The allocator manages a deallocation list per thread to collect deallocated memory blocks. Blocks from these thread-local deallocation lists are gathered, sorted, and merged into the global recycle list. Memory management is described in detail in Section 5.4.

5.2 Background Threads

Background threads are helper threads transparent to the application. ArchTM uses two background threads to manage the PM pool at runtime – the *garbage collection (GC) manager* and the *fragmentation manager*. The GC manager recycles freed persistent objects. The fragmentation manager examines memory usage by regions and aggregates memory blocks for defragmentation (see Section 5.4).

Algorithm 1 Start, read, and write operations.

```
 1: function APT_TX_BEGIN
 2:     volatile  $TxID = $ GLOBALTIMESTAMP()
 3:     $TxState$.ATOMIC_STORE($TxID, BEGIN$)
 4:     $Fence()$
 5: end function
 6:
 7: function APT_TX_READ(TxState, objID)
 8:     $obj = objLookupTable[objID]$
 9:     if $obj.new == NULL$ then   return $obj.old$
10:     end if
11:     if $obj.writer \rightarrow TxID == TxState.TxID$ then   return $obj.new$
12:     end if
13:     if $obj.writer \rightarrow State == COMMITTED$ and $obj.writer \rightarrow CommitID <=$
        $TxState.TxID$ then   return $obj.new$
14:     end if
15:     return $obj.old$
16: end function
17:
18: function APT_TX_WRITE(TxState, objID)
19:     $obj \leftarrow objLookupTable[objID]$
20:     if $obj.new! = NULL$ and $obj.writer \rightarrow TxID == TxState.TxID$ then
21:         return $obj.new$
22:     end if
23:     if LOCK($obj.writer$) then
24:         $obj.writer = \&TxState$
25:         $obj.new = $ ALLOC($obj.old$)
26:     else   ABORT_AND_RETRY( )
27:     end if
28:     $obj.new = $ DUPLICATE($obj.old$)
29:     $obj.new.header.txID = TxState.TxID$
30:     # append the object to write-set
31:     $write\_set.insert(objID)$
32:     return $obj.new$
33: end function
```

Algorithm 2 Commit and post-commit operations.

```
 1: function APT_TX_ON_COMMIT(TxState)
 2:     if EMPTY(write_set) then   return
 3:     end if   # read-only tx
 4:     for each $obj \in write\_set$ do   FLUSH($obj.new$)
 5:     end for   $Fence()$   # persist all modified objects
 6:     volatile   $CommitID = $ GLOBALTIMESTAMP()
 7:     $TxState$.ATOMIC_STORE($COMMITTED, CommitID$)
 8:     $Fence()$
 9:     APT_TX_POST_COMMIT(TxState)
10: end function
11:
12: function APT_TX_POST_COMMIT(TxState)
13:     for each $tx \in Ongoing\_TXs$ do
14:         if $tx.TxID < TxState.CommitID$ then   WAIT_FOR($tx$)
15:         end if
16:     end for
17:     while $obj \leftarrow write\_set.pop()$ do
18:         FREE($obj.old$)   # append to the reclaim list
19:         $obj.old = obj.new$
20:         $obj.new = NULL$
21:         $obj.writer = NULL$
22:         UNLOCK($obj.writer$)
23:     end while
24:     $TxState$.ATOMIC_STORE($END, INF$)
25:     $Fence()$
26: end function
```

5.3 Transaction Operations

ArchTM supports five core operations to begin, read, write, commit, and postcommit in a transaction. ArchTM provides snapshot isolation [8, 13] similar to existing work [12, 27, 45, 48, 55, 63] and industrial production database systems [2–5, 7, 53]. We illustrate the operations in Algorithms 1 and 2.

APT_TX_BEGIN starts a transaction and assigns a unique ID ($TxID$) based on the global timestamp (Alg. 1 Line 2) to the transaction. A transaction state variable ($TxState$) is created and stored in the metadata area on PM. $TxState$ is a combination of the $TxID$, state and transaction commit ID ($CommitID$). At the transaction beginning, ArchTM adds $TxID$ and the state BEGIN into $TxState$ by an atomic write.

APT_TX_READ returns a pointer to a copy of the persistent object with ($objID$). If the object is not being updated by any transactions (Alg. 1 Line 9), the pointer to the old copy is returned. If the object is being updated by the current transaction (Alg. 1 Line 11) or a transaction committed before the current transaction starts (Alg. 1 Line 13), the pointer to the new copy is returned. Otherwise, ArchTM returns the pointer of the old copy. The whole process is lock-free.

APT_TX_WRITE returns a pointer to the persistent object $objID$ ready for update. If the persistent object already has a new copy and the most recent update to the copy is performed by the current transaction, the pointer to the new copy is returned (Alg. 1 Lines 20-22). If the persistent object does not have a new copy, the application thread allocates a one, acquires the write lock of the *writer* of the object (Alg. 1 Line 23), duplicates the old copy to the new one, and then updates the new copy. The application thread also inserts the object ID into the *write-set*. If the application thread fails to obtain the write lock of the object, *APT_TX_WRITE* aborts and retries in a new transaction.

APT_TX_ON_COMMIT commits a transaction. If the transaction is read-only, no persistent operations are performed. Otherwise, ArchTM persists the modified objects recorded in the *write-set* to PM (Alg. 2 Lines 4-5). After that, ArchTM gets a global timestamp as CommitID and updates the state to COMMITTED with CommitID in the transaction state variable by an atomic write.

APT_TX_POST_COMMIT cleans up a committed transaction. First, it checks whether there is any ongoing transaction that starts before the current transaction is fully committed (Alg. 2 Lines 13-16). It reclaims the old copy (i.e., putting the old copy in the thread-private deallocation list) after the earlier transactions are fully committed. This ensures that the old copy of the persistent object is no longer required in any ongoing transaction. Afterwards, ArchTM sets the new copy as the old copy and sets the new copy as $NULL$. Finally, it resets and unlocks the writer of modified objects. ArchTM also updates and persists the transaction state to END and CommitID to INF.

5.4 Memory Management for Transactions

ArchTM uses a customized persistent object allocator. Depending on the size of an allocation request, ArchTM chooses the locality-aware data path for small allocations and use the regular data path for the others. We describe the locality-aware data path in this Section.

Allocation. When a thread attempts to allocate a persistent object, ArchTM searches through the thread's private allocation list to locate the first memory block larger than the requested size. If no block is found, ArchTM fetches freed memory blocks from the global free list to refill the allocation list. Each fetch takes a large and fixed-size memory segment to avoid frequent contention on the global free list. The fetching history is stored and persisted in *CHKP-diff*. Each fetching event in *CHKP-diff* contains the IDs of ongoing transactions, where the segment is fetched from, the segment start address, and the segment size. If ArchTM cannot find free memory blocks from the global free list, ArchTM replenishes memory blocks from the global recycle list to the global free list.

Deallocation (garbage collection). When a thread deallocates a persistent object, the object is ready for GC because no other transactions are accessing the object (Alg. 2 Lines 13-16). The deallocated object is added to the thread's private deallocation list. In the background, the GC manager periodically collects freed objects from threads to the global recycle list, during which freed blocks are zeroed. Synchronization between application threads and the GC manager is rare because an application thread only updates the head while the GC manager only updates the tail of a deallocation list. The global recycle list is sorted to speed up search during allocation and fragmentation ratio computation during defragmentation. Sorting is inexpensive because when freed memory blocks are added to the global recycle list, they are already mostly sorted.

Defragmentation. ArchTM implements an online defragmentation mechanism to improve the memory usage of the global recycle list. The mechanism works at the granularity of memory regions (4KB). The defragmentation manager monitors the fragmentation ratio (defined as the ratio of used memory to 4KB) of each memory region in the global recycle list. A memory region with a fragmentation ratio greater than f (f is 50% in our evaluation) is deemed underutilized. ArchTM aggregates persistent objects in underutilized regions and migrates them to a newly allocated memory region. For migration, the defragmentation manager internally creates a "mock" write transaction to ensure the atomicity of data migration and correctness. At the end of the "mock" write transaction, the migrated objects in the original location will be reclaimed through the deallocation process.

5.5 Recovery Management

ArchTM follows a two-step recovery process to resume the program from a crash.

1) Detect uncommitted transactions: This is implemented by checking the state of each transaction state variable on PM. If a state is neither COMMITTED nor END, ArchTM inserts the transaction ID of the uncommitted transaction into a temporary buffer (named $uncommittedTxIDs$).

2) Rebuild object lookup table: ArchTM creates a new object lookup table on DRAM (described in Section 5.1) and loads the object information to the new table. The loading process is similar to processing write operations, with the difference that the object information is retrieved from PM instead of the user request. In particular, ArchTM scans the user data area on PM to find persistent objects and inserts their location information (i.e., pointers to the objects on PM) into the lookup table. ArchTM puts the location information of each persistent object in the lookup table based on the object ID which indicates where the location information is in the original lookup table. To identify an object on PM, ArchTM relies on the object header annotated in each persistent object. The header contains the object ID and object size, which is used to isolate persistent objects from each other on PM.

ArchTM must eliminate object copies in uncommitted transactions. If the transaction ID of an object copy is found in $uncommittedTxIDs$, the object copy is discarded, and its memory space is reclaimed.

Since ArchTM does not invalidate the memory blocks of a freed object copy until the memory manager recycles them to the global recycle list, a persistent object may have multiple copies in the PM pool. Therefore, ArchTM must identify the latest copy and discards the others. When ArchTM reads a persistent object from PM and finds that the object already exists in the object lookup table, ArchTM compares the transaction IDs annotated in these two copies and only keeps the latest one. The mapping information in the object lookup table is then updated, and the old copy is reclaimed.

Crash consistency is ensured because (1) all modifications in uncommitted transactions are discarded, (2) all modifications in a committed transaction are persisted, and (3) only the latest committed copy of a persistent object is retained. All uncommitted transactions are captured in the transaction state variables stored in PM, and all object copies with a transaction ID in these uncommitted transactions are discarded during recovery. A transaction is only marked committed after all modified persistent objects in this transaction (collected in *write-set*, (Alg. 1 Line 31)) are persisted (Alg. 2 Lines 4-5). ArchTM identifies the latest committed copy of an object by transaction IDs, which by design guarantees that a transaction ID is no earlier than the commit ID of another transaction if they update the same object (Alg. 1 Lines 23-27).

5.6 Reduction of Recovery Time

The recovery process may take a long time if a large number of persistent objects exist on PM because ArchTM must scan the entire user data area to locate objects and rebuild the object lookup table. The recovery can take as long as tens of minutes on PM with TBs of capacity.

We reduce the recovery time by incorporating an incremental checkpoint technique into ArchTM. In particular, ArchTM periodically copies the modifications of the object lookup table since the last checkpoint to PM, such that ArchTM builds a checkpoint of the object lookup table on PM. When restarting from a crash, ArchTM uses the checkpoint to resume the object lookup table, instead of building it from scratch.

ArchTM uses the following method to detect modifications of the lookup table since the last checkpoint. After taking an incremental checkpoint, ArchTM temporarily blocks all transactions, sets all pages of the object lookup table on DRAM as read-only by enabling write protection, and then resumes the transactions. Any following writes to those pages will trigger a write-protection page fault, indicating that the page is modified. ArchTM records the faulted pages for the next incremental checkpoint. After a page fault is triggered, the page is not write-protected, and there will be no more page faults. At the time of incremental checkpoint, only those modified pages are copied from DRAM to PM.

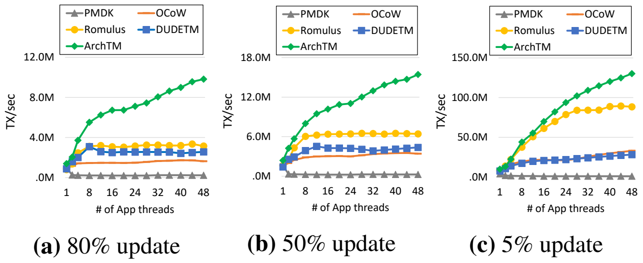
Using a persistent checkpoint of the object lookup table for recovery is not enough to reduce recovery time, because after a crash, the updates on object metadata since the last checkpoint are lost. To solve this problem, the persistent object allocator in ArchTM records the fetching history of memory segments in *CHCP-diff* (Section 5.4-Allocation), and those PM segments contain the modifications of persistent objects since the last checkpoint. ArchTM scans those modified segments to find missing updates as Section 5.5. Note that page information collected from the above page fault mechanism cannot be used to locate missing segments, because it is on DRAM and gets lost after crash. The page information is only used to implement incremental checkpoint. Overall, ArchTM uses a combination of the checkpoint of the object lookup table and the fetching history of memory segments in *CHCP-diff* to quickly restore the object lookup table.

6 Evaluation

We use an Intel Purley platform that has 2nd Gen Intel® Xeon® Scalable processor, 32KB L1 caches, 1MB L2 caches, and a shared 35MB L3 cache. The memory subsystem consists of 12 DRAM DIMMs and PM DIMMs, providing a total of 192 GB DRAM and 1.5 TB PM. We compare ArchTM with four state-of-the-art transaction systems: PMDK [30], Romulus [21], DudeTM [46], and OCoW [50]. PMDK uses libpmemobj v1.7. Libpmemobj does not support isolation, so we use a readers-writer lock to protect a transaction from concurrent accesses. Romulus uses RomulusLR for the best performance, and DudeTM uses the default persistent scheduler. We set the checkpoint frequency in ArchTM to 30 seconds, and the size of the pre-allocated PM segment to two GB. The granularity of memory regions for defragmentation (Section 5.4) is 4KB.

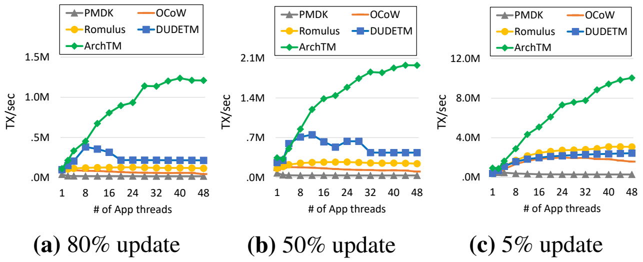
6.1 Micro-benchmarks

Hash tables and red-black trees are two important concurrent data structures widely used in database workloads [24, 37, 44, 60]. We evaluate hash tables and red-black trees with three update rates (5%, 50%, and 80%) similar to [21,27,40,61,74]. Each transaction operation randomly accesses a key-value pair to read or update. Each key-value pair uses an 8-byte key and 16-byte value. Figure 5 and 6 present the performance and scalability results.



(a) 80% update **(b)** 50% update **(c)** 5% update

Figure 5: Performance and scalability of hash table.



(a) 80% update **(b)** 50% update **(c)** 5% update

Figure 6: Performance and scalability of red-black trees.

Hash table. The experiments use a hash table of 10K buckets, each as a single linked list. The hash table is initialized with 100K key-value pairs. ArchTM outperforms the other systems by 10x, 12x and 22x on average at 80%, 50%, and 5% update rates respectively (Figure 5). ArchTM demonstrates high scalability as the concurrency in applications increases to the maximum. In contrast, Romulus stops scaling, and DUDETM and OCoW have performance degradation when the application uses more than 16 threads.

In write-intensive workloads (Figures 5a and 5b), the sequential write technique contributes significant improvement at low application concurrency. When the number of application threads continues increasing, contention on the Optane media outweighs the write amplification. Other optimizations in ArchTM, such as the transient metadata on DRAM, start coping with this new bottleneck, and sustain performance scaling. In a read-intensive workload (Figure 5c), ArchTM achieves nearly linear speedup through scalable object referencing on DRAM and lock-free read operations.

Romulus scales well when the concurrency is low (i.e., 1-8 threads) for write-intensive workloads. At high concurrency, its single-threaded write operations become a performance bottleneck. DUDETM cannot consume volatile logs from DRAM to PM in time, causing long delays. OCoW has frequent metadata updates on PM for object remapping, allocation, reclamation, thereby reducing the overall throughput. PMDK shows the worst performance because it uses read-write locks extensively for logging and memory allocation.

Red-black tree. In this experiment, the red-black trees are initialized with one million key-value pairs. ArchTM outperforms the other systems by 7x-13x on average. It exhibits near-linear scalability as the number of threads increases for the read-intensive workloads (Figure 6c).

We notice that all three workloads have performance fluctuation at about 28 application threads, likely caused by the high contention on the Optane media. This contention point arrives later than that in the hash table, because each update in the red-black tree needs to search longer than in the hash table, reducing its write intensity.

PMDK, OCoW, Romulus, and DUDETM have lower scalability in the red-black tree than in the hash table. In write-intensive workloads (Figures 6a and 6b), the performance in these systems either fails to scale or even degrade when the concurrency increases. They suffer from the expensive synchronization [27, 40]. The lock-free operations and scalable object referencing in ArchTM avoid this contention and enables high performance at high concurrency.

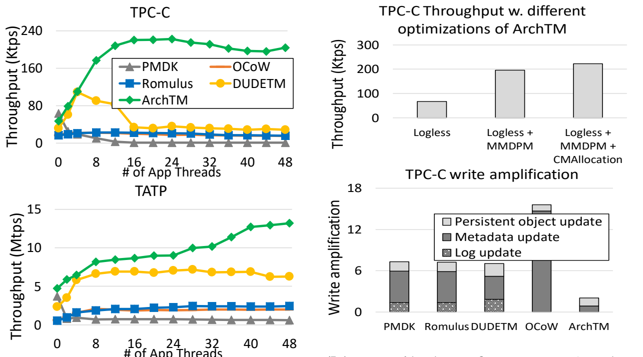
6.2 Real World Workloads

We run TPC-C [42] and TATP [54]) against PMEMKV [1]. PMEMKV is a in-memory key-value store developed by Intel. In this experiment, we use its *cmap* storage engine.

TPC-C. We run the *new-order* transaction test, where each application thread works on its corresponding warehouse and executes new order transactions. This workload has a 100% update rate. On average, each transaction inserts more than ten new objects into different tables and modifies more than ten existing objects. ArchTM significantly outperforms others by 10x, 9x, and 5x on average (Figure 7a). PMDK is more than 100 times slower than others when more than 12 threads are used. The performance of ArchTM scales up quickly to 24 application threads and then slightly declines due to write contention on the Optane media. DUDETM only scales up to eight threads because its performance is limited by centralized persistent logs. Once the background thread cannot flush the log buffer to PM in time, the application threads are delayed.

TATP. TATP is widely used for online transaction processing. ArchTM outperforms DUDETM, Romulus, OCoW and PMDK by 2x, 6x, 5x, and 13x, respectively. For evaluation, we implement three read-only and three read-write transactions similar to [27, 46]. The transactions in TATP are less write-intensive than the TPC-C test. Therefore, ArchTM achieves performance scaling up to the maximum application threads. Since TATP has less write traffic than TPC-C, DUDETM sustains performance at 16 threads and beyond.

We quantify the contribution from our design techniques to performance improvement. We separate techniques into logless, minimized metadata modification on PM (*MMDPM*), and contiguous memory allocation (CMAlloction). Figure 7b compares the performance using different techniques when running TPC-C with 24 application threads. In this test, We use DUDETM as the baseline, and its throughput is 37 Ktps. Minimized metadata modification on PM contributes the most (66%) performance improvement. The logless design and the contiguous memory allocation technique contribute 18% and 16% performance improvement, respectively. Using the same test configuration (Figure 7b-bottom), we quantify the write amplification in the five systems. The write amplification in ArchTM is only 2.03. ArchTM has 3x to 8x lower write amplification than the other systems.



(a) Throughput

(b) Detailed Performance Analysis on TPC-C

Figure 7: Real-world workloads with PMEMKV.

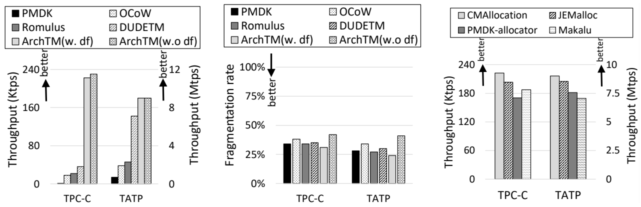
6.3 Performance Analysis

Online defragmentation. We evaluate the online defragmentation technique by quantifying the system throughput and memory fragmentation rate in TPC-C and TATP against PMEMKV. Each test uses 24 application threads. We compare the performance of ArchTM with and without online defragmentation (denoted as w.df and w.o.df in Figure 8), with four other PM systems.

The two ArchTM-based systems outperform other systems by 12x and 3x on average on TPC-C and TATP, respectively. The online defragmentation in ArchTM reduces memory fragmentation from 58% to 69% with only 3% overhead on system throughput on TPC-C. TATP is less write-intensive than TPC-C, and therefore no noticeable performance loss is observed from the online defragmentation. Figure 8b reports the memory fragmentation rate of all systems. The memory fragmentation rate of the ArchTM with online defragmentation is 4%, 9%, 3%, and 5% lower than PMDK, OCOW, Romulus, and DUDETM respectively. *ArchTM with online defragmentation is 14% lower that without it, demonstrating the necessity of using our online defragmentation.*

Contiguous memory Allocation. We evaluate the effectiveness of contiguous memory allocation (*CMAllocation*) in ArchTM. For comparison, we port ArchTM to use three state-of-the-art allocators, i.e., JEMalloc [23], PM allocator in PMDK [30], and Makalu [15]). Figure 8c reports the system throughput when ArchTM is equipped with the different allocators in TPC-C and TATP against PMEMKV.

The CMAllocation-based system achieves 9% and 6%



(a) Impact of online defragmenation.

(b) Memory fragmentation.

(c) Improvment from CMAlloction.

Figure 8: Evaluate the effectiveness of online defragmenation and contiguous memory allocation.

higher throughput than JEMalloc-based system on TPC-C and TATP, respectively. It also offers 20% and 18% higher throughout than PMDK- and Maruku-based systems. The customized locality-aware data path enables CMAllocation to encourage sequential writes on PM for better performance. In the PMDK and Maruku allocators, the poor scalability and frequent metadata updates become the bottleneck.

Checkpoint and Recovery Time. The checkpoint frequency trades off system throughput with recovery time. We vary the frequency from one second to 60 seconds in TPC-C against PMEMKV. We compare the system throughput with and without checkpoints, and find that checkpoints impose 11% overhead at the highest checkpoint frequency (i.e., one second). At a moderate checkpoint frequency, e.g., 30 seconds, the throughput loss diminishes to less than 1%.

We trigger a random crash after the program runs two minutes and then time the recovery. As expected, the recovery time increases linearly as the checkpoint frequency decreases. For the 30 GB workload set of TPC-C, ArchTM recovers the system in eight seconds at a checkpoint interval of 30 seconds and the object lookup table consumes 5.6 GB DRAM. For the same experiment, the other four systems recover faster than ArchTM. The overhead in recovery in ArchTM comes from scanning the PM data area because ArchTM needs to identify updates since the last checkpoint before the crash to rebuild the object lookup table. ArchTM trades a slightly longer recovery time for better runtime performance based on the assumption that crashes in the production environment are infrequent [10].

Transaction abort rate. Transaction aborts occur when a transaction tries to get the write lock of the writer of a persistent object but fails. We measure the abort rate. With 24 threads running highly write-intensive workloads with 80% update rate using the hash table and red-black tree, the abort rate is 1% and 2% on average, respectively. With 24 threads running the TPC-C and TATP, the abort rate is 2% and 2% on average, respectively. In general, the abort rate is very low.

7 Related Work

Undo-logging based PM transactions. Intel's PMDK [30] (libpmemobj) and NV-Heap [19] use undo-logging to log persistent objects on PM for crash recovery. Atlas [16] also uses undo-logging. It provides compiler and runtime supports to instrument writes to PM. JUSTDO logging [33] implements an Atlas-like log management system designed for machines with persistent caches. It stores the program counter and resumes the execution of critical sections from the same point where a crash happens. iDO [47] optimizes JUSTDO logging by avoiding logging each persistent store. Specifically, iDO divides the critical section into several idempotent code regions and only logs live program states at the beginning of each idempotent region within the critical section.

Redo-logging based PM transactions. NVthreads [29] supports redo-logging for multi-threaded C/C++ programs. It logs dirty pages tracked by the OS page protection between critical sections. DUDETM [46] uses shadow DRAM to decouple transaction updates and redo-logging. It leverages a background thread to copy and persist the modifications in redo logs to hide the logging overhead. Romulus [21] and Pisces [27] use variants of redo-logging. They both keep two copies of the data and replicate updates from one copy to the other to ensure crash consistency. Romulus uses a volatile log to record memory locations modified during a transaction to improve the performance of data copy. Pisces targets read-most workloads and explores snapshot isolation to ensure lock-free read operations.

CoW-based PM transactions. CDDS [68], BPFS [20], and multi-version concurrency control based transactions (e.g., TimeStone [40]) create a new copy and apply updates to the new copy to avoid writing log records.

The above logging-based and CoW-based works optimize PM transactions by reducing data replication or persistence overhead. In contrast, ArchTM introduces architecture-awareness to adapt the transaction system to leverage the micro-architecture (i.e., internal buffer and data size block) on the PM hardware. With the architecture-awareness, ArchTM improves the efficiency of PM writes by avoiding small writes and encouraging coalescable writes.

8 Conclusions

Enabling high-performance transactions is critical for leveraging persistent memory for data-intensive applications. We reveal performance problems in common transaction implementations on real PM hardware and highlight the importance of considering PM architecture characteristics for transaction performance. In this paper, we present ArchTM, an architecture-aware PM transaction system. On average, ArchTM outperforms the state-of-the-art PM transaction systems (PMDK, Romulus, DudeTM, and the Oracle system) by 58x, 5x, 3x, and 7x respectively.

Acknowledgment. This work was partially supported by U.S. National Science Foundation (CNS-1617967, CCF-1553645 and CCF1718194). This research was supported by the Exascale Computing Project (17-SC-20-SC). LLNL-CONF-808913. We thank our shepherd, Natacha Crook and anonymous reviewers for their constructive comments and suggestions.

References

[1] Key/value datastore for persistent memory. https://github.com/pmem/pmemkv.

[2] mongoDB. https://www.mongodb.com/.

[3] MySQL. https://www.mysql.com/.

[4] Oracle. www.oracle.com.

[5] PostgreSQL. https://www.postgresql.org/.

[6] Rdtscp — read time-stamp counter and processor id. www.felixcloutier.com/x86/rdtscp.

[7] Redis. https://redis.io/.

[8] A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Technical report, USA, 1999.

[9] Aerospike. Building Real-Time Database at Petabyte Scale. https://www.aerospike.com/partners/intel-optane/.

[10] M. Alshboul, J. Tuck, and Y. Solihin. Lazy Persistency: A High-Performing and Write-Efficient Software Persistency Technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, June 2018.

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

[12] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[13] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, 1995.

[14] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, 2000.

[15] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. *SIGPLAN Not.*, 51(10):677–694, October 2016.

[16] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[17] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[18] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. Atmem: Adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, 2020.

[19] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[20] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[21] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 271–282, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Christian Craft. Persistent Memory Primer. https://blogs.oracle.com/database/persistent-memory-primer.

[23] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. Technical report, 2006. http://jemalloc.net/.

[24] H. Garcia-Molina and K. Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[25] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *Proc. VLDB Endow.*, 13(10):1304–1318, April 2020.

[26] Wolfram Gloger. Wolfram Gloger's malloc. http://www.malloc.de/en/.

[27] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[28] Eric Hanson. How to Use MemSQL with Intel's Optane Persistent Memory. https://www.memsql.com/blog/how_to_use_memsql_with_intels_optane_persistent_memory.

[29] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 468–482. ACM, 2017.

[30] Intel. Persistent Memory Development Kit. https://pmem.io/.

[31] Intel. ntel® Optane^{TM} Persistent Memory 200 Series Delivers on Average 25% More Bandwidth with up to 4.5 TB Total Memory per Socket. https://newsroom.intel.com/wp-content/uploads/sites/11/2020/06/Optane-Mem-200-Series-Product-Brief.pdf.

[32] Intel. Revolutionizing Memory and Storage. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[33] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016.

[34] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[35] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient Persist Barriers for Multicores. In *International Symposium on Microarchitecture*, 2015.

[36] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A scalable ordering primitive for multicore machines. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[37] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 2013.

[38] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated Persist Ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[39] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[40] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[41] Redis Labs. Redis Enterprise on Intel® Optane^{TM} DC Persistent Memory Offers Cost-Effective Scaling to Petabytes. https://redislabs.com/press/redis_enterprise_intel_optane_dc_persistent_memory_offers_cost_effective_scaling_petabytes_2.

[42] Scott T. Leutenegger and Daniel Dias. A Modeling Study of the TPC-C Benchmark. In *SIGMOD Record*, 1993.

[43] P. Li, D. R. Chakrabarti, C. Ding, and L. Yuan. Adaptive software caching for efficient nvram data persistence. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[44] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, 2017.

[45] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P. Stevenson. Si-tm: Reducing transactional memory abort rates through snapshot isolation. *SIGPLAN Not.*, 49(4):383–398, February 2014.

[46] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[47] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[48] S. Lu, A. Bernstein, and P. Lewis. Correct execution of transactions at different isolation levels. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1070–1081, 2004.

[49] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-Ordering Consistency for Persistent Memory. In *IEEE 32nd International Conference on Computer Design*, 2014.

[50] Virendra J. Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. Persistent memory transactions. *CoRR*, abs/1804.00701, 2018.

[51] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

[52] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. Mega: Overcoming traditional problems with os huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, 2019.

[53] Microsoft. Snapshot Isolation in SQL Server. https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server.

[54] Simo Neuvonen, Antoni Wolski, Markku manner, and Vilho Raatikka. Telecom Application Transaction Processing Benchmark . http://tatpbenchmark.sourceforge.net/.

[55] Lois Orosa and Rodolfo Azevedo. Logsi-htm: Log based snapshot isolation in hardware transactional memory. 07 2015.

[56] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[57] I. Peng, K. Wu, J. Ren, D. Li, and M. Gokhale. Demystifying the performance of hpc scientific applications on nvm-based memory systems. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 916–925, 2020.

[58] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the intel optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems*. ACM, 2019.

[59] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. Mesh: Compacting memory management for c/c++ applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, 2019.

[60] L. Qiaoyu, L. Jianwei, and X. Yubin. Performance analysis of data organization of the real-time memory database based on red-black tree. In *2010 International Conference on Computing, Control and Industrial Engineering*, 2010.

[61] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, 2019.

[62] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[63] Torvald Riegel, Christof Fetzer, , and Pascal Felber. Snapshot isolation for software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, TRANSACT'06, 2006.

[64] Paul Menage Sanjay Ghemawat. TCMalloc: Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/.

[65] SAP. Realize the Promise of In-Memory Computing. https://discover.sap.com/sap-hana-dc-persistent-memorynew/en-us/index.html.

[66] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory allocation for nvram. In *ADMS@VLDB*, 2015.

[67] H. Shu, H. Chen, H. Liu, Y. Lu, Q. Hu, and J. Shu. Empirical Study of Transactional Management for Persistent Memory. In *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium*, 2018.

[68] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, USA, 2011. USENIX Association.

[69] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.

[70] H. Wan, Y. Lu, Y. Xu, and J. Shu. Empirical Study of Redo and Undo Logging in Persistent Memory. In *5th Non-Volatile Memory Systems and Applications Symposium*, 2016.

[71] Kai Wu, Ivy Peng, Jie Ren, and Dong Li. Ribbon: High performance cache line flushing for persistent memory. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, 2020.

[72] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, 2020.

[73] P. Zardoshti, T. Zhou, Y. Liu, and M. Spear. Optimizing persistent memory transactions. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[74] Pengfei Zuo, Yu Hua, and Jie Wu. Level hashing: A high-performance and flexible-resizing persistent hashing index structure. *ACM Trans. Storage*, 2019.

SPHT: Scalable Persistent Hardware Transactions

Daniel Castro*, Alexandro Baldassin†, João Barreto*, Paolo Romano*
*INESC-ID & Instituto Superior Técnico †UNESP - Universidade Estadual Paulista

Abstract

With the emergence of byte-addressable Persistent Memory (PM), a number of works have recently addressed the problem of how to implement persistent transactional memory using off-the-shelf hardware transactional memory systems.

Using Intel Optane DC PM, we show, for the first time in the literature, experimental results highlighting several scalability bottlenecks of state of the art approaches, which so far have only been evaluated via PM emulation.

We tackle these limitations by proposing SPHT (Scalable Persistent Hardware Transactions), an innovative Persistent Transactional Memory that exploits a set of novel mechanisms aimed at enhancing scalability both during transaction processing and recovery. We show that SPHT enhances throughput by up to 2.6× on STAMP and achieves speedups of up to 2.8× in the log replay phase vs. state of the art solutions.

1 Introduction

The emerging byte-addressable Persistent Memory (PM) is poised to be the next revolution in computing architecture. In contrast to DRAM, PM has lower energy consumption, higher density and retains its contents even when powered off. Nearly one decade after the first research papers started investigating PM, typically resorting to inaccurate software-based emulations/simulations, the first DIMMs of PM are finally commercially available [23]. This constitutes a notable opportunity to validate, with real PM hardware, the efficiency of the PM-related methods that have been proposed so far.

Along this research avenue, this paper focuses on one problem that has received significant attention in the recent literature: how to implement Persistent Transactional Memory (PTM) in commodity systems equipped with PM and Hardware Transactional Memory (HTM).

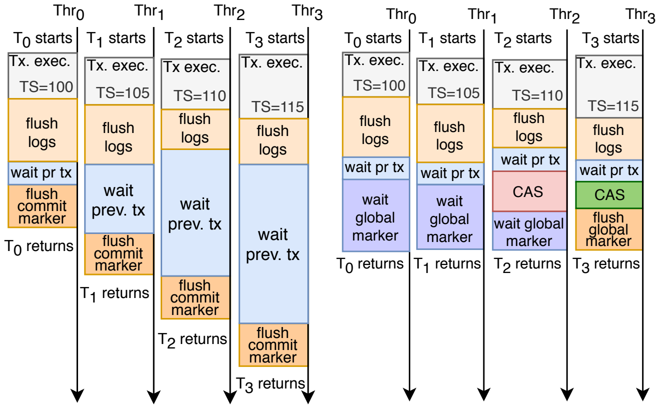
HTM implements in hardware [19, 21, 32] the abstraction of Transactional Memory (TM), an alternative to lock-based synchronization that can significantly simplify the development of concurrent applications [34]. Due to its hardware nature, HTM avoids the overhead imposed by software-based TM implementations. However, the reliance of commodity HTM implementations on CPU caches raises a crucial problem when applications access data stored in PM from within a HTM transaction. Since CPU caches are volatile in today's systems, HTM implementations do not guarantee that the effects of a hardware transaction are atomically transposed to PM when the transaction commits — although such effects are immediately visible to subsequent transactions.

To tackle this issue, recent proposals [4, 14, 15, 28] rely on a set of software-based extensions that, conceptually, are based on Write Ahead Logging (WAL) schemes [31]: first they log modifications and only then they modify the actual data. However, implementing a WAL scheme on commodity HTM raises several challenges. The fact that commercial HTMs deterministically abort transactions that try to persist the cached logs in PM is an impediment to reuse classical DBMS solutions [31]. Instead, logs need to be flushed outside of the transaction boundaries. This essentially decouples transaction *isolation* – as provided by the HTM's concurrency control – from transaction *durability* – as ensured by WAL.

This decoupling introduces a second challenge: PTM implementations need to ensure that the order by which the effects of a transactions become visible is consistent with the order by which it is persisted.

Existing solutions for commodity HTM cope with this challenge by introducing a sequential phase in the critical execution path of the commit logic. To circumvent this limitation, several solutions allow transactions that commit in HTM to externalize their results before their durability is ensured.

Unfortunately, this approach relaxes correctness, since it no longer guarantees *immediate durability* [26]. This is a fundamental limitation for applications that, after committing a transaction, can trigger externally visible actions. An external entity might observe actions that causally depend on a transaction whose writes to PM may not be recovered after a crash (under such relaxed PTMs). To cope with this, applications are extended with intricate compensation logic, which, we argue, is at odds with the original simplicity of transac-



(a) NV-HTM

(b) SPHT

Figure 1: (a) The main scalability limitation of NV-HTM [4]: the commit marker is updated in a decentralized but sequential way. (b) How SPHT avoids this limitation (see §3.1).

tional memory. We aim to avoid the pitfalls of relaxed PTM, by providing a PTM for commodity HTM/PM that ensures immediate durability while achieving high scalability.

As a **first contribution**, we experimentally evaluate the cost of ensuring immediate durability with today's state of the art PTMs in a real system equipped with HTM (Intel TSX) and PM (Intel Optane DC PM). We implement 6 PTM systems (disabling their relaxed durability optimizations) and experimentally evaluated them in STAMP [6] and TPC-C [37]. To the best of our knowledge, this represents the first study to evaluate PTMs on a real PM/HTM-equipped system.

As a **second contribution**, we address the question of whether immediate durability can scale on commodity HTM. We devise novel scalable techniques to address the limitations of existing PTMs, which we incorporated in SPHT (Scalable Persistent Hardware Transactions). In a nutshell, SPHT introduces a new commit logic that considerably mitigates the scalability bottlenecks of previous alternatives, providing up to 2.6×/2.2× speedups at 64 threads in, resp., STAMP/TPC-C. Moreover, SPHT introduces a novel approach to log replay that employs cross-transaction log linking and a NUMA-aware parallel background replayer. In large persistent heaps, the proposed approach achieves gains of 2.8×.

The remainder of the paper is organized as follows. §2 provides background on PTM, highlighting the scalability issues of previous solutions. §3 presents SPHT, which we evaluate in §4 against other 5 state of the art PTMs. Finally, §5 concludes the paper.

2 Background on PTM

Various works have investigated how to implement PTM systems. Existing solutions differ by the durability semantics they offer, the nature (hardware and/or software) of the mechanisms they adopt and in some key design dimensions.

Durability semantics. Some PTMs consider "classic" strong guarantees, i.e., if a transaction T returns successfully from its commit call, then T shall be recovered upon a crash and any transaction whose effects T observed shall also be recovered.

We use the term "immediate durability" [26] to refer to the above guarantees, although other papers call it "immediate persistence" [14] or "durable linearizability" [22].

Relaxed durability semantics, which other systems [14, 15, 22, 28] have considered, only ensure that the recovered state is equivalent to one produced by the sequential execution of a *subset* of the committed transactions. As such, these systems can fail to recover transactions that successfully returned from the commit call, e.g., because a crash occurs briefly after that. Intuitively, implementations that rely on relaxed durability semantics have higher throughput for two main reasons: (i) they require a less strict synchronization among concurrent transactions in their commit phase; and, (ii) they allow for removing the costs incurred to ensure durability out of the critical path of execution of the transaction commit logic.

However, when applications do require stricter semantics, programmers are faced with additional complexity: having to develop compensation logic or to manually specify for which sub-transactions immediate durability should be guaranteed [15]. The focus of this work is on immediate durability (despite some tested systems supporting relaxed durability). Next we discuss how systems ensure such semantics.

Software vs hardware implementations. The first PTM proposals relied on software mechanisms [8, 38]. These initial works paved the way for the following generations of software-based PTM implementations [9, 10, 20, 25–27, 29, 30, 41]. Essentially, these proposals extend different software transactional memory (STM) algorithms with logging and recovery mechanisms to ensure durability.

With the introduction of HTM support in mainstream CPUs [17, 33], a second wave of proposals has focused on how to enable the execution of hardware transactions (i.e., transactions executed using HTM) on PM. Due to its hardware nature, HTM avoids the notorious instrumentation costs of STM, which can impose significant overhead especially in applications with short-lived transactions [13]. In existing HTM systems, though, committed transactions are not guaranteed to be atomically persisted, as some of their writes may be lingering in the cache and not have been applied to PM. Further, commodity HTMs do not allow persisting the cached logs to PM within the hardware transaction context. This prevents the use of classical WAL schemes (conceptually at the basis of existing software-based PTMs), which assume that logs are always persisted before application data is.

Some works tackled these issues by proposing *ad hoc* hardware extensions [1, 3, 16, 24, 40]. As such, these solutions cannot be used with existing off-the-shelf systems. More recent works have overcome this shortcoming by proposing software-based approaches that operate on top of unmodified commodity HTM. The most notable examples are DudeTM [28], cc-HTM [15], NV-HTM [4] and Crafty [14]. All these solutions implement some form of WAL on top of HTM. For improved throughput, the log is typically implemented as a set of per-thread logs in PM[1].

Enforcing the WAL rule. Existing PTMs for commodity HTM rely on two main alternative strategies to enforce the WAL rule, i.e., ensure that the log of a transaction is persisted before any of its changes is applied to persistent data.

A first approach, adopted by DudeTM and NV-HTM, is to have hardware transactions access a *volatile* "shadow" copy of the persistent snapshot.

A second option is non-destructive undo logging, as proposed in Crafty [14]. In this approach, transactions execute directly in PM, with their writes tracked in a persistent undo log and a volatile redo log. To ensure the WAL rule, every write issued in an HTM transaction is undone before commit, which guarantees that the transaction does not alter the PTM's state. Next the undo log is persisted and only then the transaction's writes can be applied to PM.

In both strategies, after committing in HTM, the generated log(s) are flushed to PM in two steps. First, a commit marker is appended to the log. This marker defines the transaction as durable and includes a timestamp that is used to order durable transactions. Next, each logged write is *replayed*, in timestamp order, on the target memory location in PM.

The choice between shadow copy or non-destructive undo logging strongly impacts the available solution space. As we show next, the state of the art systems that implement the above approaches suffer from severe scalability limitations.

Ordering transactions in the logs. One key issue is how to establish the replay order of update transactions in the logs. Existing proposals opt for logical or physical timestamps.

In the first alternative a global logical clock is incremented before each transaction commit and is later appended to the redo log. This type of clocks are likely to become a contention point at high thread counts (§4), hindering scalability by generating frequent spurious aborts. DudeTM employs logical timestamps and, thus, suffers from the above limitation. An additional scalability issue of DudeTM is that its volatile per-thread redo-log has to be processed, copied and flushed by auxiliary thread(s) to a centralized redo-log in PM, incurring relevant synchronization costs.

In contrast, physical timestamps can be acquired at low latency and with no synchronization via, e.g., the x86 RDTSCP instruction. cc-HTM, NV-HTM and Crafty exploit this mechanism. To ensure that the state recovered after a crash is consistent, though, these systems require ensuring an additional property: before a transaction T with timestamp TS can append its commit marker to the log, any other committed transaction T' with timestamp $TS' < TS$ must be already marked as committed in the log. In fact, if this property were violated, upon recovery, T' may not be replayed, whereas T will - this would yield an inconsistent state in case T had observed some write of T' (since T logically depends on T').

Fig 1a illustrates the scheme employed by NV-HTM. An inherently sequential phase in the commit logic ultimately bounds the maximum system throughput to the rate at which commit markers can be persisted in the log. Considering that flushes incur a higher latency in PM than in DRAM [23], this scheme can severely hinder scalability.

Besides the above issue, Crafty adopts a non-destructive undo logging scheme, which incurs additional problems. After flushing the undo log of an HTM transaction T (recall that this is done outside the scope of T, after its commit), Crafty starts a new HTM transaction that atomically: (1) checks if a global clock has changed since T's first execution; and (2) in the negative case, replays T's redo log in PM and updates the global clock. If the global clock is found to have increased (i.e., *any* concurrent transaction did commit), the whole transaction logic of T is re-executed: if T produces the same writes as in its first execution, T is marked as durable; else, T's undo log is discarded and the whole process is restarted.

This approach has two main limitations: (1) the update of the global clock is likely to generate contention at high thread counts, causing frequent transaction re-executions; (2) executing twice a transaction not only introduces overhead, but also increases the likelihood of conflicts by extending the period of time during which transactions execute concurrently.

Log replay. Another key design choice is how to replay the writes in the redo log on PM, while respecting the timestamp order. With the exception of Crafty, log replay occurs only after the transaction(s) being replayed is already durable (as ensured by the persistent redo log). Therefore, the application threads do not need to wait for this phase in order to continue, which can be performed in background. However, there are two relevant exceptions where the progress of the application is affected by the log replay. The first one is upon recovery, where a stable snapshot is rebuilt from the persistent logs. The second one is during transaction processing: once the available log space is exhausted, the application threads have to wait for the log replay to reclaim log space. The efficiency of the log replay process is, thus, of paramount importance.

All the analysed solutions (but Crafty) adopt a non-scalable log replay mechanism: they sequentially replay the logs via a single background thread. Moreover, the efficiency of the replay phase in existing systems decreases as the number of threads processing transactions grows. The larger the thread count, in fact, the larger the number of per-thread logs that need to be examined in the replay phase to determine which transaction (from some per-thread log) should be replayed next, according to the timestamp order.

Summary. Table 1 summarizes the main scalability limitations of state of the art solutions. As we show in the remainder of the paper, SPHT avoids all of them. Regarding the

[1] With the exception of DudeTM, which maintains per-thread logs in volatile memory, whose entries are later flushed to log(s) in PM.

	DudeTM	cc-HTM	NV-HTM	Crafty	SPHT
Global clock updated by txs	Y	N	N	Y	N
Extended tx vulnerability window	N	N	N	Y	N
Sequential mechanism to ensure durability	N	Y	Y	Y	N
Sequential Log Replay	Y	Y	Y	N	N

Table 1: Summary of the factors limiting the scalability of proposed PTM implementations for commodity HTM (assuming their operation with immediate durability semantic).

scalability challenges associated with orderly redo logging, SPHT addresses them by introducing a novel, highly scalable commit protocol (§3.1) that amortizes the cost of ensuring immediate durability across multiple concurrent transactions. SPHT's design avoids spurious aborts due to the access to shared metadata from within hardware transactions [28] or to the need to execute a transaction twice [14]. Concerning log replay, SPHT overcomes the scalability limitations of existing solutions by introducing a mechanism for NUMA-aware parallel log replaying (§3.3) as well as a "log linking" technique (§3.2) that spares replayers from the cost of scanning every thread's log to determine the transaction replay order.

3 SPHT

SPHT assumes a system in which PM is exposed to applications by means of *persistent heaps*. A persistent heap is created by using the operating system (OS) support to memory-map the persistent data, stored in a PM-aware file system, into the application address space [36]. SPHT exposes a classic transaction demarcation API and transparently exploits the underlying HTM support along with a novel software-based scheme to ensure immediate durability.

Fig 2 illustrate SPHT's architecture, which includes two main processes: the **Transaction Executor** (TE), which runs the TM-based parallel application, and the **Log Replayer** (LR). Transactions are executed by multiple worker threads spawned by the TE process. The TE process also *mmaps* a persistent heap into its address space using the OS Copy-on-Write (CoW) option. This option creates a shadow copy of the persistent heap shared by all worker threads, which serves as a working snapshot (WS) that transactions access directly. Updates to the WS are not immediately propagated to the persistent heap. Thus, the updates generated by committed HTM transactions are still volatile.

Like most systems analysed in §2, each worker thread has a private durable redo log that it uses to track the updates performed by each transaction. Once a transaction commits, its updates may still reside in the cache. Thus, the redo log needs to be explicitly forced to persistent memory *after* the HTM commit. At that point, a timestamped commit marker declares the transaction as durable. We discuss how SPHT implements this mechanism in a highly scalable way in §3.1.

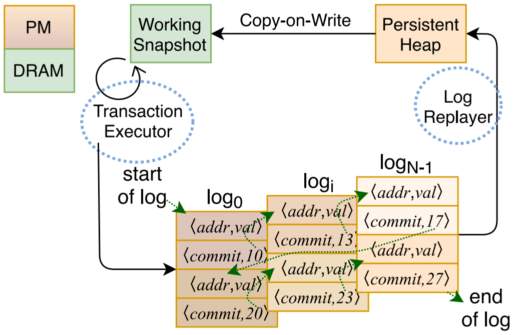


Figure 2: High level view of SPHT architecture.

Since transactions work on a shadow volatile working snapshot, the persistent heap is kept up to date by eventually replaying the redo logs. To handle this, the LR process *mmaps* the persistent heap into its address space, but in a shared (instead of private) state. As such, the LR can directly write to the persistent heap. As mentioned in §2, the LR process takes advantage of two novel ideas to ensure high scalability of log replay. First, it relies on a novel *log linking* mechanism that spares the replayer threads from the cost of having to determine which transaction should be replayed next. We present this mechanism in §3.2. Second, the log replay is parallelized in a NUMA-aware fashion, as detailed in §3.3.

3.1 Transaction processing and durability

The key idea that SPHT exploits to overcome the scalability limitations of state of the art solutions is to mitigate the cost of ensuring immediate durability by amortizing it across multiple transaction commits. SPHT's design is based on the observation that, at high thread count, a large number of transactions is likely to be concurrently trying to commit. SPHT exploits this observation by ensuring the immediate durability of *all* of them via a *single* update of a *persistent global marker*, noted $^{p}marker$, which stores the timestamp of the most recent durable transaction. This approach is similar in spirit to the group commit mechanism used in DBMSs [12], which in SPHT we customize to make use of HTM and PM.

Fig 1b illustrates the idea at the basis of the proposed mechanism, by considering the same execution used to illustrate the scalability limitations of NV-HTM (in Fig 1a). Similarly to NV-HTM, SPHT relies on physicals clocks to establish the order by which transactions are replayed. After an HTM commit, SPHT allows the threads to flush their logs out of order (i.e., without any inter-thread synchronization). As discussed in §2, there is a key issue with prior proposals based on physical timestamps (e.g., NV-HTM). The issue is that the log of a transaction T, after being flushed to PM, cannot be marked as durable just yet. In fact, there may exist some other transaction T' with a smaller timestamp and still not marked as durable, whose effects T may have observed.

SPHT copes with this issue as follows: each thread externalizes a timestamp ^{v}ts in volatile memory that contains the

Algorithm 1 SPHT- Base algorithm

```
    Shared Volatile Variables
1:  $^v ts$[N], $^v marked$[N], $^v isUpd$[N]
    Persistent Variables
2:  $^p writeLog$[N], $^p marker$
    Thread Local Volatile Variables
3:  $^v ts'$, $^v skipCAS$
4:  function BEGINTX
5:      $^v isUpd[myTid] \leftarrow$ FALSE
6:      $^v skipCAS \leftarrow$ FALSE
7:      UNSETPERSBIT($^v ts[myTid]$)                 ▷ Logs are not persistent
8:      $^v ts[myTid] \leftarrow$ RDTSCP             ▷ lower bound of final ts
9:      HTM_BEGIN                                    ▷ begin hw tx
10: function WRITE(addr, val)
11:     logWrite(addr, val)                          ▷ log to PM, no flush
12:     *addr← val                                   ▷ execute write
13: function COMMITTX
14:     $^v ts' \leftarrow$ RDTSCP                   ▷ store physical clock to local var.
15:     HTM_COMMIT                                   ▷ commit hw transaction
16:     $^v ts[myTid] \leftarrow ts'$                ▷ Externalize the final timestamp
17:     if isReadOnly then                           ▷ Read-only txs...
18:         SETPERSBIT($^v ts[myTid]$)               ▷ ...unblock the others
19:         return                                   ▷ ...and return immediately
20:     $^v isUpd[myTid] \leftarrow$ TRUE            ▷ Mark as update tx
21:     logCommit($^p writeLog[myTid]$, $ts'$)       ▷ Flush tx log.
22:     SETPERSBIT($^v ts[myTid]$)                   ▷ Signal logs are durable
23:     WAITPRECEDINGTXS
24:     UPDATEMARKER
25: function WAITPRECEDINGTXS
26:     for $t \in [0..N-1]$ do
27:                                   ▷ Wait until prec. txs have flushed their logs
28:         while $^v ts[t] < {}^v ts[myTid] \wedge \neg$ISPERSBIT($^v ts[t]$) wait
29:                                   ▷ If any update tx with large ts exists...
30:         if $^v ts[t] > {}^v ts[myTid] \wedge {}^v isUpd[t]$ then
31:             $^v skipCAS \leftarrow$ TRUE         ▷ this tx can skip the CAS
32: function UPDATEMARKER
33:                       ▷ Is it needed to and am I responsible for updating $^p marker$?
34:     if $^p marker < {}^v ts[myTid] \wedge \neg {}^v skipCAS$ then
35:         val← $^p marker$
36:         while val < $^v ts[myTid]$ do
37:             val ← CAS($^p marker$, val, $^v ts[myTid]$)
38:         if (CAS was successful) then
39:             flush($^p marker$)
40:             $^v marked[myTid] \leftarrow {}^v ts[myTid]$   ▷ Signals $^p marker$ is flushed.
41:             return
42:     while TRUE do                                ▷ Wait till flush of $^p marker$
43:         for $t \in [0..N-1]$ do                  ▷ ...is complete
44:             if $^v marked[t] \geq {}^v ts[myTid]$ then return
```

following information: (i) the timestamp of the last transaction; and, (ii) whether the log of the last transaction is persistent (*isPers* bit). After flushing its logs, T advertises to all other threads its completion by setting the *isPers* bit in its $^v ts$. T then enters a wait phase during which T scans the timestamps of the other worker threads with a two-fold purpose: (i) ensuring that any transaction with a smaller timestamp has finalized persisting its own logs; (ii) determining which is the transaction with the largest timestamp, among the ones currently in the commit phase.

The former condition guarantees that T can be safely marked as durable, by updating (and flushing) the global marker ($^p marker$). The latter condition enhances efficiency by exploiting, opportunistically, the presence of other concurrent transactions to reduce the number of updates (and flushes) of $^p marker$. Specifically, if T detects a transaction T' with a larger timestamp, T avoids updating $^p marker$, as T' will do so. When T' updates $^p marker$, the durability of T is also implicitly ensured, since T' will store in $^p marker$ its own timestamp, which is larger than the one of T and, as such, ensures also the durability of T.

This mechanism is not exempt from critical races. In fact, two transactions may assume to have the largest timestamp and attempt to update concurrently $^p marker$. We tackle this issue by manipulating $^p marker$ via a Compare-and-Swap (CAS) instruction. Fig 1b illustrates an example execution. Transactions T_0 and T_1 detect the presence of T_2 and/or T_3 and delegate to them the update of the global marker. T_2 and T_3 compete via a CAS to update $^p marker$. Assuming that T_3 succeeds, T_3 flushes $^p marker$, thus ensuring the durability of the 4 transactions. As shown in Fig 1, not only SPHT reduces the number of synchronous updates of the commit marker with respect to solutions like NV-HTM (reducing the pressure on the bandwidth-constrained PM [23]), but it also allows multiple transactions to be marked as durable in parallel.

Note that, if no concurrent transaction is detected after flushing the logs, the proposed solution has a cost similar to NV-HTM, as both require synchronously updating a commit marker. In the case of SPHT, though, a single global marker is updated, whereas in NV-HTM, each thread appends a commit marker to its own log. Because of this, SPHT uses a more expensive operation (i.e., a CAS) to update the global marker. As we will show in §4, though, this cost is largely outweighed by the scalability benefits that the SPHT's scheme enables. It is also worth pointing out that this design tends to minimize the chance that multiple transactions contend to CAS the global marker. In fact, $^p marker$ is only updated by a transaction that detects to have the largest timestamp. Thus, most of the CAS operations are uncontended and, therefore, introduce relatively low overhead in modern processors [35].

Pseudo-code. The above scheme is formalized in Alg 1. For simplicity, memory and persist barriers are omitted in the pseudo-code and are discussed below. First, all loads/stores to shared variables abide by C/C++ acquire/release semantics. Second, we use synchronous flushes (CLWB followed by SFENCE) in *logCommit* (l.21) and *flush* (l.39).

DATA STRUCTURES. We mark volatile data structures with a superscript v (^{v}) and persistent variables with a superscript p (^{p}) for clarity. SPHT maintains two persistent data structures: (i) per-thread redo-logs ($^p writeLog[N]$); and, (ii) a global marker ($^p marker$), which stores the timestamp (physical clock) of the most recently durably commmitted transaction, i.e., guaranteed to be replayed in case of a crash.

Each thread t (of the N available in the system) also uses the following global *volatile* data structures: (i) the timestamp of the last (or current) transaction T executed by t ($^v ts[N]$); (ii) the *isPers* flag, implemented by reserving a bit in $^v ts[t]$, which serves to notify whether t has (synchronously) flushed the logs of T to PM; (iii) the last timestamp t wrote to and flushed in $^p marker$ ($^v marked[N]$); and, (iv) a flag that advertises whether T is an update transaction ($^v isUpd[N]$).

The logs are per-thread circular buffers containing an ordered sequence of transactions. Each logged transaction is a sequence of (i) $\langle addr, val \rangle$ pairs (i.e., the transaction's write set) followed by (ii) a timestamp that serves also as an end delimiter. The timestamp is distinguishable from an address by setting its first bit to 1. For simplicity, we omit the metadata used to track the log's start and end.

BEGIN TRANSACTION. Before a thread t starts a hardware transaction T (via HTM_BEGIN, line 9), t stores the current value of the physical clock (obtained via RDTSC) in its ^{v}ts variable and sets its *isPers* bit to 0. It also sets its $^{v}isUpd$ variable to false, which informs other threads that T is not guaranteed to be an update transaction, yet.

It should be noted that, at this stage, the timestamp advertised in ^{v}ts represents a lower bound estimate on the final timestamp (i.e., the one establishing the durability order) that T will obtain right before committing (via HTM_COMMIT, l.15). This mechanism ensures the visibility of T throughout its execution to other concurrent threads.

WRITE INSTRUMENTATION. SPHT logs the writes (Alg 1, l.10) in PM via the logWrite primitive. Logging a write consists in appending a pair $\langle addr, val \rangle$ at the tail of the log.

COMMIT PHASE. Before committing the hardware transaction via HTM_COMMIT (Alg 1, l.15), the final timestamp is obtained by reading the physical clock and storing it in a local variable ($^{v}ts'$). This timestamp is only advertised in the shared variable ^{v}ts after HTM commits. The latter, in fact, is accessed non-transactionally by concurrent threads (in the WAITPRECEDINGTXS function) and updating it from within the hardware transaction would induce (spurious) aborts.

Read-only transactions, which produce no log, can return immediately. Before, though, they set *isPers* to 1, which, as we will see, unblocks concurrent threads that may be waiting.

Update transactions, instead, append their timestamp to the log and flush it (via the logCommit primitive). Next, they advertise that they are update transactions and that they are durable by setting their $^{v}isUpd$ and *isPers* flags, respectively.

Next, in WAITPRECEDINGTXS, T examines the timestamps of every concurrent transaction and waits until the ones with a smaller timestamp have finished flushing their logs (l.28). At this point it is safe to update the global marker with the timestamp of T. However, to enhance efficiency, in l.30, T determines whether there is an update transaction with a larger timestamp, say T'. In this case, when T' updates the global marker with its own timestamp, it also ensures the durability of T. Hence, T omits the updating of the global marker, sets the $^{v}skipCAS$ flag and just waits until a timestamp larger than its own has been persisted in the global marker.

Finally, the transaction executes UPDATEMARKER. Here, it verifies if the global marker does not yet ensure its own durability ($^{p}marker <^{v} ts[myTid]$) and if it cannot count on other transactions with larger timestamp to update $^{p}marker$ ($^{v}skipCAS$ is false): in such a case, the transaction attempts to CAS $^{p}marker$ (l.37) to the value of its ^{v}ts, until a timestamp larger than or equal to its own is present.

If T successfully executes its CAS, T ensures that the write it performed is persisted by flushing $^{p}marker$ (l.39). T advertises that $^{p}marker$ is flushed by writing its ^{v}ts in its $^{v}marked$ variable. After that, T returns.

If T fails the CAS, T needs to wait until it observes a value in the $^{v}marked$ array that is larger than its timestamp: this guarantees that some thread must have CASed and flushed a value in $^{p}marker$ that also ensures T's durability.

SINGLE GLOBAL LOCK (SGL). HTM is a best effort synchronization mechanism that, to ensure progress, normally relies on pessimistic fall-back path (e.g., activated if the transaction fails repeatedly to commit in hardware) based on a Single Global Lock (SGL). When this mode is activated, any concurrent hardware transaction is immediately aborted. However, in SPHT, if a thread activates the SGL path, there may still be transactions that have already completed executing in HTM, but are still in their commit phase (e.g., flushing their logs). To guarantee correct synchronization with these transactions, the SGL path ensures the durability of its updates by using the same logic of HTM transactions.

Correctness arguments. We prove the correctness of SPHT by showing that it satisfies two properties (which were already used to define NV-HTM's correctness criteria [4]): (C_1) the timestamps obtained during transaction execution reflect the HTM commit order[2]; (C_2) if a transaction T returns from a commit call to the application, the effects of T and of every committed transaction that precedes T in the HTM serialization order, are guaranteed to be durable.

SPHT and NV-HTM share the same timestamping scheme, which was already proved to ensure property C_1 [4]. Thus, in the following, we focus on proving that SPHT ensures C_2.

If a transaction T, executing at thread t, returns successfully from its commit call to the application then: (i) the log of t necessarily includes T, including its final commit marker (Alg 1 l.21); (ii) $^{p}marker$ persists a value larger or equal than the timestamp of T ($^{v}ts[t_T]$), since either T set $^{p}marker$ to $^{v}ts[t_T]$ (l.38-40) or some other concurrent transaction T' s.t. $^{v}ts[t'_{T'}] > {}^{v}ts[t_T]$ updated and flushed $^{p}marker$ (l.42-44).

These conditions ensure that upon recovery T will be replayed. It is only left to prove that, if T returns from its commit call, any committed transaction T', s.t. $^{v}ts[t'_{T'}] < {}^{v}ts[t_T]$, will also be replayed. This is guaranteed since, before returning from its commit call, T ensures that any thread that may be executing a transaction T' with a smaller timestamp has set *isPers* (l.22). Hence, the log of the thread that executed T' necessarily includes T', with its final commit marker, which, together with the condition $^{p}marker \geq {}^{v}ts[t_T] > {}^{v}ts[t'_{T'}]$, ensures that T' will also be replayed.

[2] More formally, if a transaction T with a timestamp ts conflicts with T' with ts', and $ts < ts'$, then T is serialized by HTM before T'.

3.2 Linking transactions in the log

The algorithm presented in the previous section (similarly to other solutions [4, 14, 15, 28]) requires replayers to scan the whole set of per-thread logs to determine the transaction that should be replayed next. This can have a significant impact on the log replay performance (up to 3.5× slowdown, §4), especially in systems where a large number of threads can process transactions (since each thread maintains its own log).

SPHT tackles this issue by extending the transactions' log with an additional entry that is used to store a pointer to the beginning of the next transaction in the replay order. Transactions update this pointer during their commit stage.

Let us denote with T_i the i-th transaction in replay order and assume that transactions are replayed from the oldest to the most recent one. In a nutshell, once transaction T_i has committed in hardware and established its final (physical clock based) timestamp, it needs to determine the identity of transaction $T_{i\text{-}1}$, and update the link slot in the log of $T_{i\text{-}1}$ with a pointer to (the start of) T_i's log.

Unfortunately, extending the algorithm presented in §3.1 to allow T_i to determine the identity of $T_{i\text{-}1}$ is not trivial. The key problem is that, when transaction T_i reaches its commit phase, the thread that committed $T_{i\text{-}1}$, denoted $t_{T_{i\text{-}1}}$, may have already started a new transaction and overwritten its timestamp $^{v}ts[t_{T_{i\text{-}1}}]$. This makes it impossible for T_i to determine the identity of $T_{i\text{-}1}$ by inspecting the ^{v}ts array.

To address this issue, SPHT tracks also the metadata of the previous transaction processed by each thread. This is sufficient since we ensure that if T_i has not determined the identity of $T_{i\text{-}1}$ yet, then $t_{T_{i\text{-}1}}$ will be able to start at most one new transaction. To ensure this property, T_i scans the metadata of the other threads and establishes its predecessor *before* setting its *isPers*. Recall that this scheme allows T_i to prevent transactions with larger timestamps from completing their commit phase. Thus, it prevents $t_{T_{i\text{-}1}}$ from committing any transaction that $t_{T_{i\text{-}1}}$ started after committing $T_{i\text{-}1}$.

During this scanning phase, T_i discriminates between (concurrent) transactions with smaller timestamps that have their *isPers* set to 1 or 0. Transactions with *isPers* set to 1 already established their final ^{v}ts, so their timestamp can immediately be analyzed to determine if any of them may be T_i's predecessor. Further, T_i does not need to wait for these transactions before moving on with UPDATEMARKER.

Transactions with smaller timestamps that have their *isPers* set to 0, though, prevent T_i from executing UPDATEMARKER. T_i tracks these transactions in $precTXs$, a set that will be consulted during T_i's wait phase. Before starting to wait, T_i sets *isPers* to 1 to unblock transactions with larger timestamps.

Next, the algorithm proceeds similarly to the base version. Namely, T_i waits for all transactions in $precTXs$ (i.e., that may precede T_i) and then executes the update marker logic. The key difference is that, in the wait phase, once T_i can determine the final timestamp for a transaction T_j, it also verifies whether T_j might be its preceding transaction. This is achieved by checking whether T_j has the largest timestamp among the transactions that precede T_i (i.e., $T_j = T_{i\text{-}1}$). Finally, before returning from the commit call, T_i updates the link slot of $T_{i\text{-}1}$ to point to the start of T_i's log.

Pseudo-code. The pseudo-code formalizing the proposed mechanism is reported in Alg 2. The lines of code that are unchanged with respect to Alg 1 are coloured in brown. For space constraints we have to omit the correctness proof for Alg 2, which can be found in our technical report [5].

Algorithm 2 SPHT- Forward linking.

```
Additional Shared Volatile Variables:
$^{v}logPos[N]$, $^{v}prevLogPos[N]$, $^{v}prevTs[N]$
Additional Thread Local Volatile Variables
$^{v}pTs$, $^{v}pLogPos$, $^{v}pThread$, $^{v}precTXs$
function BEGINTX
    $^{v}isUpd[myTid] \leftarrow$ FALSE ; $^{v}skipCAS \leftarrow$ FALSE ;
    atomic do                                          ▷ vectorial instr stores multiple fields
        $^{v}logPos[myTid] \leftarrow$ myLinkSlot             ▷ flags link pos for the next tx
        UNSETPERSBIT($^{v}ts[myTid]$)
        $^{v}ts[myTid] \leftarrow$ RDTSCP
    HTM_BEGIN
function COMMITTX
    $ts' \leftarrow$ RDTSCP ; HTM_COMMIT
    $* ^{v}logPos[myTid] \leftarrow ts'$                    ▷ flags stable ts in own log
    $^{v}ts[myTid] \leftarrow ts'$
    if isReadOnly then
        SETPERSBIT($^{v}ts[myTid]$)
        return
    $^{v}isUpd[myTid] \leftarrow$ TRUE ;
    logCommit($^{p}writeLog[myTid]$, $ts'$)
    SCANOTHERS                                         ▷ estimate prev & unstable txs
    SETPERSBIT($^{v}ts[myTid]$)                          ▷ next tx can write in link
    WAITUNSTABLETXS                                    ▷ discover prev tx
    UPDATEMARKER
    $* ^{v}pLogPos \leftarrow$ myLinkSlot                  ▷ link prev tx to my log
    atomic do                                          ▷ keep track of this tx
        $^{v}prevLogPos[myTid] \leftarrow {}^{v}logPos[myTid]$
        $^{v}prevTs[myTid] \leftarrow {}^{v}ts[myTid]$
function SCANOTHERS                                    ▷ estimate preceding TXs
    $^{v}pThread \leftarrow myTid$                          ▷ init search with own prev. tx
    $^{v}pTs \leftarrow {}^{v}prevTs[myTid]$ ; $^{v}pLogPos \leftarrow {}^{v}prevLogPos[myTid]$ ;
    for $t \in [0..N-1]$ do
        atomic do                                      ▷ for each $t$ take a snapshot using a...
            $tmpLogPos \leftarrow {}^{v}logPos[t]$                ▷ ... vectorial load
            $tmpPrevLogPos \leftarrow {}^{v}prevLogPos[t]$
            $tmpTs \leftarrow {}^{v}ts[t]$
            $tmpPrevTs \leftarrow {}^{v}prevTs[t]$
        if $tmpTs < {}^{v}ts[myTid]$ then                  ▷ search preceding txs
            if ISPERSBIT($tmpTs$) then                    ▷ search stable txs
                if $tmpTs > {}^{v}pTs$ then
                    $^{v}pTs \leftarrow tmpTs$ ; $^{v}pThread \leftarrow t$ ;
                    $^{v}pLogPos \leftarrow tmpLogPos$
                    continue
            else                                       ▷ prec. tx in $t$ that is still running
                append($^{v}precTXs$, $\langle t, tmpLogPos\rangle$)
        if $tmpPrevTs < {}^{v}ts[myTid] \land tmpPrevTs > {}^{v}pTs$ then
            $^{v}pThread \leftarrow t$                        ▷ search preceding txs
            $^{v}pTs \leftarrow tmpPrevTs$
            $^{v}pLogPos \leftarrow tmpPrevLogPos$
function WAITPRECEDINGTXS
    for $\langle t, {}^{v}logPos[t]\rangle \in precTXs$ do
        while $^{v}ts[t] < {}^{v}ts[myTid] \land \neg$ISPERSBIT($^{v}ts[t]$) wait
        if ISTS($*^{v}logPos[t]$)$\land *^{v}logPos[t] < {}^{v}ts[myTid] \land *^{v}logPos[t] > {}^{v}pTs$ then
            $\langle ^{v}pTs, {}^{v}pThread\rangle \leftarrow \langle * {}^{v}logPos[t], t\rangle$
            $^{v}pLogPos \leftarrow {}^{v}logPos[t]$
        if $^{v}ts[t] > {}^{v}ts[myTid] \land {}^{v}isUpd[t]$ then
            $^{v}skipCAS \leftarrow$ TRUE
```

ATOMIC ACCESS TO METADATA. In the scanning phase (SCANOTHERS , l.27), T_i needs to obtain a consistent snapshot of the metadata of every other thread. This was not an issue in Alg 1, since the per-thread metadata that T_i had to observe was just the timestamp and *isPers*, which are stored within then same single memory word. Alg 2, though, requires T_i to atomically observe a larger set of metadata, i.e., the timestamp (included *isPers*) and the position in the log of the last two transactions processed by each thread, which amounts to 32 bytes. To cope with this issue, we store these metadata contiguously in (volatile) memory and read/write them using vectorial instructions (i.e., x86 AVX), which in recent CPUs guarantee atomic multi-word manipulations [39].

TRACKING THE PRECEDING TRANSACTION. As discussed above, in SCANOTHERS, by setting its *isPers* to 0 (l.7), T_i prevents thread $t_{T_{i-1}}$ from completing the commit of its next transaction. As soon as T_i sets its *isPers* to 1, though, $t_{T_{i-1}}$ can commit a possibly unbounded number of transactions and, by the time T_i accesses $t_{T_{i-1}}$'s metadata, in WAITPRECEDINGTXS, the information regarding T_{i-1}'s timestamp and log pointer may have been already overwritten. We address this issue as follows: (i) once a thread establishes the final timestamp for a transaction T_i, it stores T's timestamp also in the link slot of T_i's log, so that this information remains accessible to the thread that executes T_{i+1} even after *isPers* of T_{i+1} is set (which, as mentioned, allows t_{T_i} to commit an arbitrary number of transactions); (ii) in SCANOTHERS, when T_i detects a transaction T_j with a smaller timestamp and *isPers* set to 0, T_i stores in *precTXs* both the identifier of the thread t_{T_j} and the position of T_j in t_{T_j}'s log; (iii) in WAITPRECEDINGTXS, T_i can then access the timestamp of T_j in t_{T_j}'s log via the pointer stored in *precTXs*.

Note that, when a transaction T_i executes WAITPRECEDINGTXS, it can include in *precTXs* transactions that have a smaller timestamp but are not T_i's immediate predecessor. Denote such a transaction as T_j. By the time T_i inspects the link slot in their log via the $^v logPos$ pointer (l.51), the link slot may have been already updated by T_j's immediate successor (i.e., T_{j+1}). In this case, T_i must safely detect that T_j cannot be its own predecessor. This is achieved by exploiting the fact that whenever a transaction timestamp is stored in the link slot (l.12), its first bit (which, recall, we use to encode *isPers*) is always set to 0. So, in order to tell whether the link slot is storing a timestamp or a pointer (ISTS() primitive, l.51), we always set to 1 the first bit of any pointer that we store in the log (and reset it 0 during the reply). This is safe since in typical architectures the first bits of an address in user-space is always guaranteed to be zero, but alternative approaches should be used if SPHT were to be used within the kernel.

Backward linking. The proposed technique can be adapted straightforwardly to link transactions in "backward" order, i.e., from the most recent to the oldest one. This enables techniques for filtering duplicate writes [4] by replaying the logs backwards and applying only the the most recent write to each memory position. Backward linking can be achieved by adapting the above logic so to have T_i store into its own log a reference to the start of $T_{i\text{-}1}$'s log.

3.3 NUMA-Aware Parallel Log Replay

The LR makes use of a snapshot in PM and the per-thread logs to produce a fresher persistent snapshot. The last transaction to be considered for replay, say T_i, is the one, among the transactions in the log, to have the largest timestamp that is also smaller than or equal to the persistent global marker ($^p marker$). Any transaction more recent than the $^p marker$ is guaranteed to not have returned from the commit call. Thus, it can be safely discarded. Using a single threaded replayer, it suffices to apply the modifications by following the links stored in the logs (see §3.2). Before pruning the log, to ensure the durability of the replayed writes, SPHT calls the x86 WBINVD instruction, which efficiently drains the caches.

The key problem to enable parallel replay in the LR is how to ensure that the order by which writes are replayed by multiple concurrent replayers respects the sequential order established in the logs. SPHT circumvents the usage of additional synchronization among different replayer threads by ensuring that their writes target disjoint memory regions. This sharding makes the replay completely parallel and spares threads from enforcing a specific write order.

Fig 3 illustrates the concept, by showing two replayer threads that navigate through the (decentralized) log following the linking information. For illustration purposes, we consider a simplistic sharding policy, which assigns responsibility of even/odd addresses to replayer threads 0 and 1, respectively.

In reality, SPHT uses a more sophisticated policy, which aims at pursuing four goals: (i) minimize overhead for the replayer; (ii) balance load among different threads; (iii) promote cache locality; and, (iv) take advantage of NUMA systems.

Specifically, SPHT shards the transactional heap in contiguous chunks of configurable size, which are strided across a fixed number of parallel replayers. This allows for mapping a given memory address to the corresponding replayer thread via an efficient hash function that simply inspects the most significant part of the address.

Arguably, using small chunks can benefit load balancing: by interleaving in a fine-grained way the regions that each replayer thread is responsible for, it is less likely that a single frequently accessed memory region is assigned exclusively to a single thread (which may generate load imbalances and hamper the global efficiency of the parallel replay process). We observed that chunks with a granularity close to the cache line size generate excessive cache traffic, leading to poor replay performance. This led us to opt for a granularity of 4KB (typical size of pages mapped in DRAM).

As mentioned, one of the design goals of the SPHT's replay logic is to take advantage of the asymmetry of modern

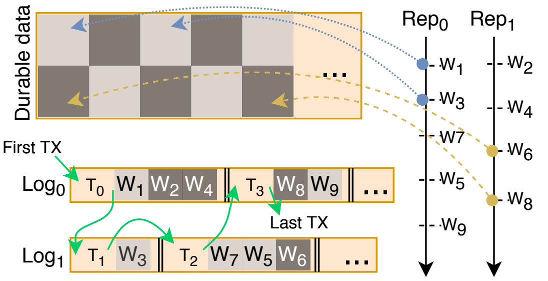


Figure 3: Parallel log replay. The notation W_i denotes a write to address i. Rep$_0$ and Rep$_1$ are responsible for odd/even addresses respectively.

NUMA systems, where accesses to local memory regions experience lower latency and higher throughput than accesses to memory regions hosted by a remote node. We exploit this feature by scattering (i.e., pinning) in round robin the set of replayers across the available NUMA nodes and making each replayer responsible of applying only the writes that target an address in their local NUMA node. To this end, we developed a simple NUMA-aware memory allocator that organizes the transactional heap into N different arenas, one for each of the N available NUMA nodes. During transaction processing, the allocation of memory regions across NUMA nodes uses a simple round robin policy to balance memory usage (but clearly alternative policies could be used [11, 18, 42]). This custom memory allocator ensures that the arenas associated with each NUMA node are placed at known address ranges. This allows the replayer threads to detect in a precise and efficient way whether any memory address in the log is mapped to their local NUMA node.

Note that, although this approach requires all replayers to scan the whole log, it is effective for two reasons: (i) since replayers execute at roughly the same speed and issue repeated read requests for the same log regions close in time, these reads are likely to be served from the CPU caches (as we will experimentally confirm in Section 4.2); (ii) PM's performance is asymmetric (read bandwidth is ~3× larger than write bandwidth [23]) hence the main bottleneck of the replay process is the apply phase, rather than log scanning.

Finally, this sharding scheme can be used in conjunction with duplicate filtering schemes [4], e.g., which scan the logs from the most recent to the oldest entry to avoid replaying duplicate writes to the same memory position. In this case, the key issue to address is how to ensure that the tracking of duplicate writes remains correct despite the existence of multiple concurrent replayers. In order to avoid costly synchronization among replayers, SPHT avoids using shared data structures to filter duplicates (e.g., thread-safe set implementations). Conversely, each replayer thread r maintains a volatile bitmap that only tracks writes to memory regions that r is responsible for (each bit of the bitmap tracking writes to a 8 bytes in PM, i.e., the granularity of each write in the log).

4 Experimental Evaluation

Our experiments seek answers to the following main questions: (i) how severe are the scalability limitations of state of the art solutions mentioned in §2 when evaluated on a real PM system (§4.1 and §4.2)? (ii) what are the performance benefits of SPHT's commit logic (§4.1)? (iii) what are the gains of the linking technique during log replay (§4.2) and what are the costs it introduces during transaction processing (§4.1)? (iv) how scalable is the parallel replay technique (§4.2)?

Experimental settings. We conducted all experiments in a dual-socket Intel Xeon Gold 5218 CPU (16 Cores / 32 Threads) equipped with 128GB of DRAM and 512GB of Intel Optane DC PM (4× 128GB). The PM is configured in "App mode" [23] using 2 namespaces and interleaved access. The presented results are the average of 10 runs.

We consider 8 different PTMs, whose implementation we make publicly available[3]: SPHT-NL (no linking), SPHT-FL (forward linking), SPHT-BL (backward linking), NV-HTM [4], DudeTM [28], Crafty [14], cc-HTM [15] and PSTM [38]. PSTM is a software TM that extends TinySTM with durable transactions using Mnemosyne's [38] algorithm. For fairness, we implemented all systems in §2 in a common framework and all of them provide immediate durability. Checkpointing is disabled during transaction processing for all solutions that accumulate logs[4]. The HTM solutions fall back to SGL after 10 retries.

4.1 Transaction processing

We evaluate the performance of SPHT using the STAMP [6] benchmark suite and TPC-C [37]. STAMP was already used to evaluate several prior related solutions [4, 14, 15], since it encompasses transactional applications that, although not originally proposed for PM, would transparently benefit from PM to attain crash-tolerance and/or have access to larger heaps. TPC-C is widely used to benchmark database systems.

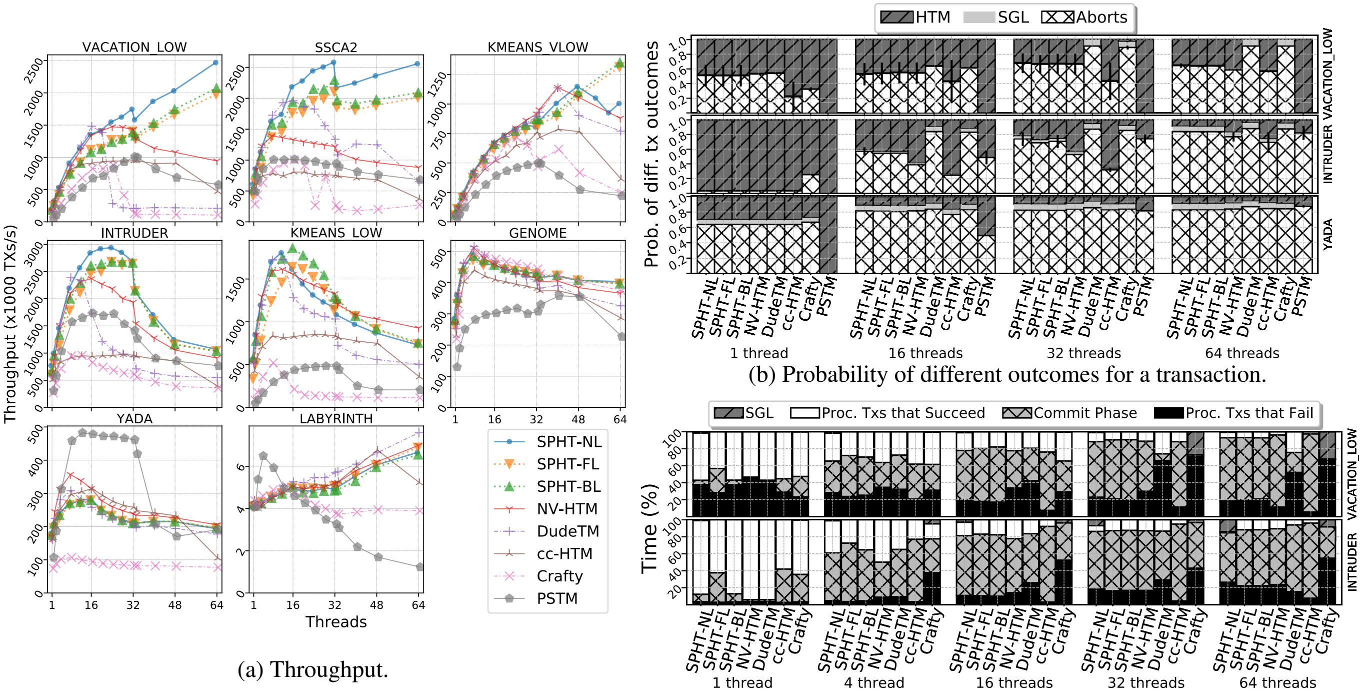
4.1.1 STAMP

STAMP includes 8 benchmarks, but we do not consider Bayes, as it is known to generate unstable performance results [7]. We consider the standard low contention workloads for Vacation and Kmeans. We also configured Kmeans to generate an additional workload with lower contention (KMEANS_VLOW), thus enabling the PTMs to achieve higher scalability levels.

Fig 4a reports throughput as a function of the number of worker threads. The top row contains low contention workloads (VACATION_LOW, SSCA2 and KMEANS_VLOW). The second row contains contention-prone workloads (INTRUDER, KMEANS_LOW and GENOME). And the bottom row have

[3] bitbucket.org/daniel_castro1993/spht

[4] cc-HTM has to activate checkpointing upon completing each transaction in order to comply with immediate durability (see transaction barrier in [15]).



(a) Throughput.

(b) Probability of different outcomes for a transaction.

(c) Transaction time breakdown.

Figure 4: STAMP [6] using standard (++) parameters: (a) throughput; (b) probability for a transaction to commit or abort in HTM, or enter the SGL; (c) breakdown of time spent: in SGL, commits in HTM, aborts in HTM, and in the commit phase.

workloads that are notoriously unsuited for HTM (YADA and LABYRINTH), since their long transactions have a memory footprint that often exceeds the CPU cache capacity.

Low contention benchmarks. SPHT achieves the largest gains with respect to the considered baselines in VACATION_LOW, where it scales up to 64 threads (with a small drop at 33 threads, when we start activating threads on the second socket [2]). At the maximum thread count, SPHT is 2.6× faster than NV-HTM (the second best solution). This can be explained by analyzing the data in Fig 4c, which reports a breakdown of the percentage of time spent by each solution in different activities: at 64 threads, NV-HTM spends significantly more time in the commit phase than SPHT, 76% vs 62%. As a consequence, the ratio between the time spent processing transactions and the time spent committing is ~2× higher for SPHT (~60% vs ~30%). Analogous considerations apply to cc-HTM, which spends almost 95% of time in the commit phase starting at 32 threads, when the single background applier thread becomes the system's bottleneck.

Analyzing the data in Fig 4b, which reports the probability for a transaction to abort, commit in HTM or by using the SGL, we notice that the HTM-based solutions suffer from a non-negligible abort probability even when using a single thread. We verified that this is the case also for non-durable HTM. The reason is that the memory footprint of some transactions exceed the HTM capacity. As the thread count grows, though, Crafty and DudeTM experience a much higher abort rate than SPHT. In Crafty's case, rolling back the transaction and replaying it afterwards (and using a conservative mechanism to detect conflicts in between these phases [14]) leads to higher conflict rates than with the SPHT's variants. In DudeTM's case, the global serialization clock imposes spurious conflicts, which are amplified at high thread counts.

SPHT-FL and SPHT-BL remain the most competitive solutions at high thread count, although they impose an overhead of up to around 25% in VACATION_LOW as well as in SSCA2 w.r.t. the no linking version. It should be noted, however, that the overhead incurred by the linking mechanism is at most 5% in all the other benchmarks. In KMEAN_VLOW, though, the linking variants actually outperform SPHT-NL. The explanation for this behaviour is that the additional operations performed by the linking variants in the commit phase serve as a back-off mechanism, reducing the overall contention. Although not shown in Fig 4b for space constraints, the abort rate with KMEANS_VLOW at 64 threads is 79%, 45% and 43% for SPHT-NL, SPHT-FL and SPHT-BL, respectively.

Finally, at high thread count, the gains of the SPHT variants w.r.t. existing solutions tend to reduce in KMEANS_VLOW, as this benchmarks generates higher contention than VACATION_LOW and SSCA2. Still, at 64 thread the SPHT variants achieve ~30% higher throughput than the best baseline (NV-HTM) and ~5× speed-ups w.r.t. the remaining ones.

Contention-prone benchmarks. These benchmarks scalability is inherently limited by their contention prone nature: above a given number of threads the likelihood of conflicts between transactions grows close to 1 and throughput is severely hampered in all solutions. Yet, it is worth noting that, in INTRUDER and KMEANS_LOW, all the SPHT variants do scale to a large number of threads and achieve significant speed-ups w.r.t. all other solutions: e.g., SPHT achieves a peak throughput that is ~30% higher than the most competitive baseline, i.e., NV-HTM, scaling up to 24 threads.

HTM-unfriendly workloads. Finally, in LABYRINTH and YADA, as expected, PSTM outperforms all the HTM-based solutions, including SPHT. That is not surprising given that these benchmarks generate large and contention prone transactions, which do not lend themselves to be effectively parallelized using HTM. It is also unsurprising that most of the HTM-based solutions achieve similar performance in these HTM-unfriendly workloads, where a significant fraction of the transactions has to be committed using the SGL (in which case all the tested solutions tend to follow a very similar behavior). The only exception being Crafty, which incurs a much larger overhead than the other HTM-based solutions, due to the large abort costs that it incurs in these workloads.

4.1.2 TPC-C

We implemented three transactions of the TPC-C benchmark, namely Payment, New-Order and Delivery, and report the results in Fig 5. All solutions suffer a throughput drop when they enter hyper-threading after 16 threads, which we do not observe in STAMP. After that drop, SPHT and its linking variants are the only solutions capable of scaling up to 48 threads. As in KMEANS, the backoff introduced by the linking mechanism allows SPHT-FL and SPHT-BL to reduce abort rates (bottom plot of Fig 5). NV-HTM stops scaling above 8 threads, although achieving abort rates that are comparable to or lower than SPHT's. This suggests that NV-HTM is being bottle-necked by its sequential commit mechanism. DudeTM exhibits the same issues as in VACATION_LOW: after 12 threads the global clock creates spurious aborts that hinder throughput (as shown, e.g., at 16 threads). cc-HTM's background thread limits its scalability beyond 8 threads. Crafty's non-destructive undo logging scheme also imposes higher abort rates than NV-HTM and SPHT.

4.2 Log replay

We evaluate the two main novel techniques at the basis of the proposed log replay scheme: (i) linking transactions in the log and (ii) using multiple parallel replayers. For space constraints, we cannot explicitly evaluate the gains deriving from our NUMA-aware design, which, however, we use in all the experiments discussed next. Overall, in the tested system, our NUMA-aware design doubles the bandwidth available to the replayer threads, which is key to increase scalability.

The efficiency of these mechanisms is affected by a number of variables including: (i) the heap size; (ii) the average number of writes per transaction; (iii) the use of filtering technique and the level of duplicates in the log.

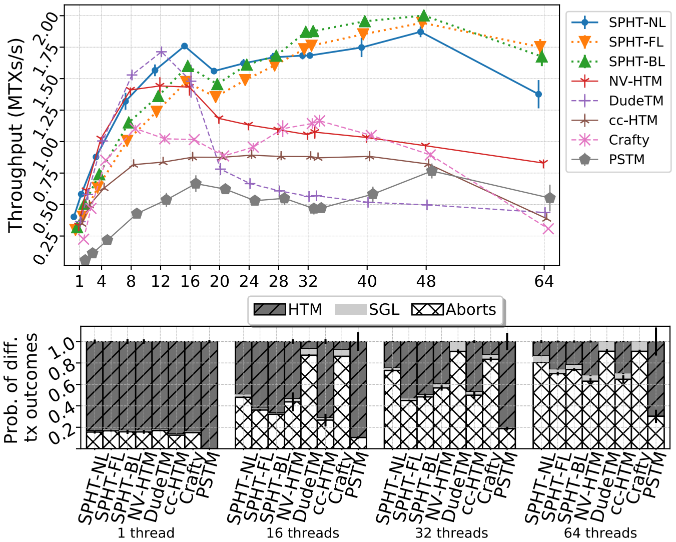


Figure 5: TPC-C using 32 warehouses, 95% Payment, 2% New Order, 3% Delivery transactions.

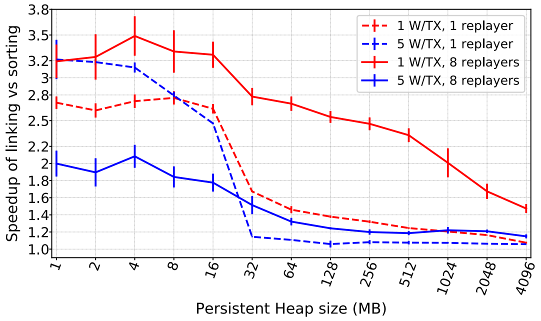


Figure 6: Performance benefits of linking.

We explore those parameters with a synthetic benchmark in which transactions access the persistent heap uniformly at random, generating a configurable number of writes in each transaction. Once the benchmark completes, the LR fully replays the produced logs and we evaluate its throughput in terms of number of logged writes replayed per second. In the following, we set the number of worker threads to 64, each producing one log (i.e., total of 64 logs to replay).

Linking. Fig 6 shows the relative gain in log processing throughput stemming from linking with respect to a classical solution [4, 28], called *sorting*, where the replay order is established by analyzing all the per-thread logs. In this experiment, the logs contain a total of 10M transactions. We vary on the x-axis the heap size and consider 4 scenarios in which: (i) transactions issue either 1 or 5 writes; (ii) replayer uses either 1 or 8 threads. Linking provides the largest benefits for small heaps (up to 3.5× speed-ups below 4MB). For large heaps, the gains of linking tend to reduce, but remain still solid (~50%) with 8 parallel replayers and 1 W/TX.

These result can be explained by considering that the heap size affects the locality of the writes issued in the replay phase

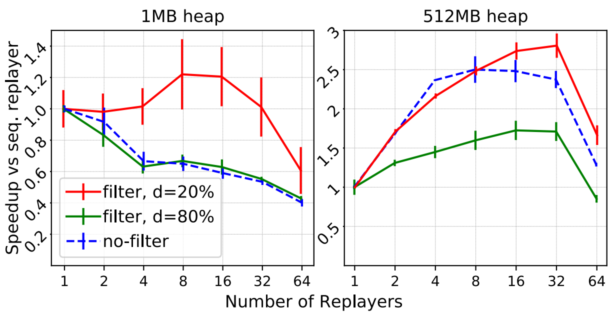


Figure 7: Speedup of parallel replay for 1MB and 512MB heap. The no-filter approach is compared with filtering for two levels of duplicates in the log: 20% and 80%.

and to what extent this write traffic can be served within the CPU cache (22MB in our case): if the writes can be replayed in cache, their relative cost decreases, amplifying the gains stemming from using an efficient mechanism to determine which transaction to replay next. Analogously, the number of writes per transaction affects the relative frequency of use linking and sorting. In fact, we see that generally the fewer the writes per transaction, the larger the gains of linking[5].

Parallel replay. Next, in Fig 7 we study how varying the degree of parallelism affects the speed-ups achievable w.r.t. sequential replay. We consider in this study also a version of the log replay that exploits the backward filtering technique [4], and use our synthetic benchmark to generate logs with 20% and 80% of duplicate writes.

The right plot, which considers a 512MB heap, shows peak gains of up to 2.8×. The use of filtering favours the scalability of the parallel replay technique and the maximum speed-ups are obtained for 20% of duplicates. This can be explained by considering that filtering reduces the write traffic towards PM, which represents the bottleneck in the no-filter scenario. For the case of 80% duplicates, though, filtering also reduces substantially the amount of writes that are effectively generated during the replay process. Accordingly, this reduces also the opportunities from benefiting from the proposed parallel log replay, which explains why the absolute speedups decrease as the duplicates' level grows from 20% to 80%.

With small heaps of 1MB (left plot), the efficiency of the parallel log replay degrades significantly. Only for the case of filtering with 20% of duplicates we observe speedups of ~20% (at 8 threads). In the other considered scenarios, parallelism ends up hindering performance. This can be explained by considering that writes to such a small heap are served entirely in the processor's cache and that the existence of a (possibly large) number of replayers intensively updating such a small working set is likely to generate strong contention and interference in the cache subsystem. Although this result pinpoints a limitation of the proposed technique, we argue that most applications that make use of large scale multicore machines and PM will likely adopt much larger heaps.

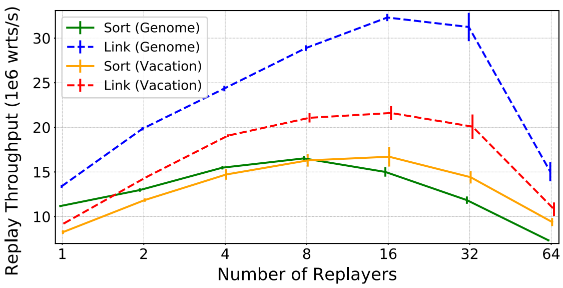


Figure 8: Log replay in VACATION_LOW and GENOME.

replayers	1	2	4	8	16
VACATION_LOW	8.64% (±0.03)	6.32% (±0.01)	5.81% (±0.03)	5.20% (±0.02)	4.75% (±0.01)
GENOME	8.85% (±0.04)	5.76% (±0.05)	4.99% (±0.06)	4.26% (±0.03)	3.82% (±0.05)

Table 2: L1 cache misses in the replay phase using linking.

Next we evaluate the joint use of parallel replay and linking, this time using realistic benchmarks, namely, VACATION_LOW and GENOME (shown in Fig 8). The proposed parallel log replay scheme has better throughput when compared to a conventional sorting approach, yielding ~1.3× and ~2.1× peak speedup, resp., for VACATION_LOW and GENOME at 16 threads. The joint use of linking further amplifies the speedups of parallel replay by an additional 35%, demonstrating how these two techniques can be effectively employed in synergy to accelerate the log replay process.

Finally in table Table 2 we report the L1 cache misses when varying the number of replayers from 1 to 16. We can observe that the cache misses decrease as the parallelism increases. This is expected, since all the replaying threads scan the whole log (i.e., generate the same stream of read accesses), confirming that this cost is amortized by an increase in the cache hits as the thread count increases.

5 Conclusions

This paper pinpointed several scalability limitations that affect existing PTM systems for off-the-shelf HTM. We tackled these limitations by proposing SPHT, a novel PTM system that integrates a number of innovative techniques targeting both the transaction processing and the log replay phases.

We evaluated SPHT in a system equipped with Intel Optane DC PM and compared it against other 5 state of the art PTM systems that had been so far only evaluated via emulation. SPHT achieves of up to 2.6× throughput gains during transaction processing, when compared to the most competitive baseline, accelerating log replay by up to 2.8×.

Acknowledgments

This work was partially supported by FCT (UIDB /50021/2020), FAPESP (2018/15519-5, 2019/10471-7) and EU's H2020 R&I programme (EPEEC project, GA 801051).

[5] Except for the case of 1 thread and heaps smaller than 8MB, arguably due to caching effects.

References

[1] Hillel Avni and Trevor Brown. Persistent hybrid transactional memory for databases. *Proceedings of the VLDB Endowment*, 10:409–420, Nov 2016.

[2] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. Investigating the performance of hardware transactions on a multi-socket machine. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'16, page 121–132, New York, NY, USA, 2016. Association for Computing Machinery.

[3] M. Cai, C. C. Coats, and J. Huang. Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 584–596, 2020.

[4] Daniel Castro, Paolo Romano, and João Barreto. Hardware transactional memory meets memory persistency. *Journal of Parallel and Distributed Computing*, 130:63–79, 2019.

[5] Daniel Castro, Paolo Romano, João Barreto, and Alexandro Baldassin. Scalable persistent hardware transactions. Technical Report 1, INESC-ID, January 2021.

[6] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46, Seattle, WA, USA, 2008. IEEE.

[7] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European Conference on Computer Systems*, pages 27–40, Paris,France, 2010. ACM.

[8] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems - ASPLOS'11*, 47(4):105–118, jun 2011.

[9] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures - SPAA'18*, pages 271–282, Vienna, Austria, 2018. ACM Press.

[10] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. *SIGARCH Comput. Archit. News*, 41(1):381–394, March 2013.

[12] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, June 1984.

[13] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 3–14, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, htm-compatible persistent transactions. In *41st ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2020*, London, UK, 2020. ACM.

[15] Ellis Giles, Kshitij Doshi, and Peter Varman. Continuous checkpointing of htm transactions in nvm. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management - ISMM'17*, pages 70–81, Barcelona, Spain, 2017. ACM Press.

[16] Ellis Giles, Kshitij Doshi, and Peter Varman. Hardware Transactional Persistent Memory. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS'18, pages 190–205, Alexandria, Virginia, USA, October 2018. ACM.

[17] Bhavishya Goel, Ruben Titos-Gil, Anurag Negi, Sally A. McKee, and Per Stenstrom. Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell. In IEEE, editor, *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 615–624, Phoenix, AZ, may 2014. IEEE.

[18] D. Gureya, J. Neto, R. Karimi, J. Barreto, P. Bhatotia, V. Quema, R. Rodrigues, P. Romano, and V. Vlassov. Bandwidth-aware page placement in numa. In *34th*

IEEE International Parallel & Distributed Processing Symposium (IPDPS 2020), IPDPS'20, New Orleans, Louisiana USA, 2020. IEEE.

[19] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[20] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware logging in transaction systems. *Proceedings of the VLDB Endowment*, 8(4):389–400, 2014.

[21] Intel Corporation. Desktop 4th Generation Intel Core Processor Family (Revision 028). Technical report, Intel Corporation, 2015.

[22] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*, pages 313–327, Paris, France, 2016. Springer.

[23] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[24] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Dhtm: Durable hardware transactional memory. In *45th Annual International Symposium on Computer Architecture - ISCA'18*, pages 452–465, Los Angeles, CA, USA, June 2018. ACM.

[25] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. In *Proceedings of the twenty first international conference on Architectural support for programming languages and operating systems - ASPLOS'16*, volume 51, pages 399–411, Atlanta, Georgia, USA, 2016. ACM.

[26] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with TimeStone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 335–349, New York, NY, USA, 2020. Association for Computing Machinery.

[27] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 335–349, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Mengxing Liu, Mingxing Zhang, Kang Chen, and Xuehai Qian. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS'17*, pages 329–343, Xi'an, China, 2017. ACM Press.

[29] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence in transactional persistent memory. *IEEE Symposium on Mass Storage Systems and Technologies*, 2015-August(1), 2015.

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

[31] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[32] Peter Bergner, Alon Shalev Houfater, Madhusudnanan Kandeasamy, David Wendt, Suresh Warrier, Julian Wang, Bernhard King Smith, Will Schmidt, Bill Schmidt, Steve Munroe, Tulio Magno, Alex Mericas, Mauricio Oliveira, and Brian Hall. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8.* IBM Redbooks, 2015.

[33] Peter Bergner, Alon Shalev Houfater, Madhusudnanan Kandeasamy, David Wendt, Suresh Warrier, Julian Wang, Bernhard King Smith, Will Schmidt, Bill Schmidt, Steve Munroe, Tulio Magno, Alex Mericas, Mauricio Oliveira, and Brian Hall. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8.* IBM Redbooks, IBM, 2015.

[34] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? *SIGPLAN Not.*, 45(5):47–56, January 2010.

[35] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. In *Proceedings of the 2015 International*

Conference on Parallel Architecture and Compilation (PACT), PACT '15, page 445–456, USA, 2015. IEEE Computer Society.

[36] Storage Networking Industry Association (SNIA) Technical Position. NVM Programming Model Version 1.2, jun 2017.

[37] Transaction Processing Performance Council. TPC-C Benchmark Revision 5.11.0.

[38] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems - ASPLOS'11*, pages 91–104, Newport Beach, California, USA, 2011. ACM Press.

[39] Darius Šidlauskas, Simonas Šaltenis, and Christian S. Jensen. Processing of extreme moving-object update and query workloads in main memory. *The VLDB Journal*, 23(5):817–841, October 2014.

[40] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen. Persistent transactional memory. *IEEE Computer Architecture Letters*, 14(1):58–61, Jan 2015.

[41] Zhenwei Wu, Kai Lu, Andy Nisbet, Wenzhe Zhang, and Mikel Luján. Pmthreads: Persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20)*, US, June 2020. ACM.

[42] Seongdae Yu, Seongbeom Park, and Woongki Baek. Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA, 2017. Association for Computing Machinery.

The Dilemma between Deduplication and Locality: Can Both be Achieved?

Xiangyu Zou[†], *Jingsong Yuan*[†], *Philip Shilane*[*], *Wen Xia*[†‡], *Haijun Zhang*[†], *and Xuan Wang*[†]
† Harbin Institute of Technology, Shenzhen ∗ Dell Technologies
‡ Wuhan National Laboratory for Optoelectronics
Corresponding author: xiawen@hit.edu.cn

Abstract

Data deduplication is widely used to reduce the size of backup workloads, but it has the known disadvantage of causing poor data locality, also referred to as the fragmentation problem, which leads to poor restore and garbage collection (GC) performance. Current research has considered writing duplicates to maintain locality (e.g. rewriting) or caching data in memory or SSD, but fragmentation continues to hurt restore and GC performance.

Investigating the locality issue, we observed that most duplicate chunks in a backup are directly from its previous backup. We therefore propose a novel management-friendly deduplication framework, called MFDedup, that maintains the locality of backup workloads by using a data classification approach to generate an optimal data layout. Specifically, we use two key techniques: Neighbor-Duplicate-Focus indexing (NDF) and Across-Version-Aware Reorganization scheme (AVAR), to perform duplicate detection against a previous backup and then rearrange chunks with an offline and iterative algorithm into a compact, sequential layout that nearly eliminates random I/O during restoration.

Evaluation results with four backup datasets demonstrates that, compared with state-of-the-art techniques, MFDedup achieves deduplication ratios that are 1.12× to 2.19× higher and restore throughputs that are 2.63× to 11.64× faster due to the optimal data layout we achieve. While the rearranging stage introduces overheads, it is more than offset by a nearly-zero overhead GC process. Moreover, the NDF index only requires indexes for two backup versions, while the traditional index grows with the number of versions retained.

1 Introduction

Deduplication is an important data reduction technique in modern commercial backup systems because it usually achieves a high deduplication ratio (the logical size divided by the post deduplication size), which was found to often be in the range of $10 \sim 30\times$ [6], thus greatly reducing storage costs. The basic technique of deduplication is to replace redundant chunks of data with references to identical chunks that have already been stored [39]. While deduplication has been applied to numerous storage and networking topics [6, 12, 21, 29, 45], our research focuses on hard-drive based deduplication for backup storage because it remains one of the most significant use cases.

For deduplication on hard drive systems, fragmentation is a serious problem: chunks from a backup that are logically consecutive may refer to previously written chunks scattered across the disks (poor locality). As a result, this fragmentation problem causes: ① poor restore performance since many random disk reads are required; ② Garbage Collection (GC) of deduplicated systems is also challenging because as previous backup versions are deleted, referenced and unreferenced chunks may be located together, and referenced chunks must be preserved to avoid data loss.

Generally, the root cause of this fragmentation (or poor locality) problem is the sharing of chunks between backup versions due to deduplication. Deduplication systems usually group the deduplicated chunks into a large unit called a **container** (often 4MB in size) for compression and maximizing write performance to arrays of disks and use a chunk-reference list (e.g. a recipe) to record referenced chunks for each backup version. As an example, consider backup *version* 1 that has few or no duplicates, so its chunks are stored sequentially in containers. Then, *version* 2 may be highly redundant with the first with small modifications throughout the backup, so its recipe has references to many chunks of the first version intermixed with references to newly written chunks. Later, *version N* tends to have even worse locality as it refers to chunks written by many previous backup versions, so restoring a backup version involves random seeks back and forth across the disks, and read amplification is high since an accessed container may have needed and unneeded chunks.

To alleviate the fragmentation problem for better restore performance, many techniques have been proposed that write some duplicates (called rewriting approaches) according to their 'fragmentation degree' to maintain a level of data locality [14, 22, 23, 30, 31]. Alternatively, there have been proposals to use memory or SSDs to cache the fragmented chunks or frequently referenced chunks [2,25], which also helps achieve a higher restore speed, though with increased hardware costs. However, fragmentation inevitably becomes worse with high generation backup versions. According to our experimental observations on four backup datasets (see Figures 7 and 9 in Section 5), even using the state-of-the-art rewriting techniques of Capping [23] and HAR [15], the restore speed drops to about 1/8∼1/3 of the sequential read speed of storage devices while the actual deduplication ratio drops about 20% ∼ 40% due to 'rewriting'.

The performance of GC is also impacted by data locality in traditional **container-based** data layouts. As older backup versions are deleted, some chunks become unreferenced by any version and can be removed from containers to reclaim space. Generally, GC includes two stages: selecting which containers have unreferenced chunks and migrating referenced chunks into new containers so selected containers can be freed. Several approaches [17,37] have explored ways to quickly select containers for the first stage. When locality is poor, containers will have a mix of referenced and unreferenced chunks, so the migration stage is time-consuming because many chunks must be read and rewritten.

Overall, existing solutions for improving restore and GC performance struggle with the dilemma between deduplication and the locality of backup workloads. Meanwhile, previous work on fragmentation in non-deduplicated storage usually reorganized data to improve the layout [18]. However, due to chunks being shared between backup versions (a complex chunk reference relationship), it seems infeasible to design an optimal layout for all backup versions while maintaining the space savings of deduplication. Moreover, reorganizing chunks can be expensive since some chunks may be referenced by all or most versions: in this case, reorganizing one chunk means almost all versions are involved [23].

In our observations of deduplicating backups, we find that almost all the duplicate chunks in a backup version B_{i+1} are derived from its previous version B_i (studied in Section 3.3), which suggests it is feasible to design *an optimal data layout of deduplicated chunks with nearly no fragmentation*, as explained with three points: ① This **optimal data layout** classifies chunks into categories (like containers) according to their reference relationship. For example, the chunks (a set M), referenced and only referenced by backup versions B_i and B_j, are classified into one category, where a category is similar to a variable-sized container. Classification ensures that if a chunk is required when restoring a backup version, other chunks in the same category are also required, which means loading an entire category does not cause read amplification. ② However, the number of categories (containers) can grow dramatically to about 2^n categories for n backup versions, according to our observation and theoretical analysis. ③ With the above observation that some rare chunk-reference relationships can be ignored, so we only consider chunks that are referenced by one version or by consecutive versions. In this way, the chunk-reference relationship is simplified and the number of categories is reduced to $n(n+1)/2$ for n backup versions, which makes the OPT data layout feasible.

Note that this classification-based OPT data layout is significantly different from a traditional deduplication framework. Traditional deduplication mainly focuses on the write path of deduplication and rarely manages the location and placement of chunks, which we call **write-friendly**. In contrast, our approach tries to redesign the data layout of deduplicated chunks to eliminate the fragmentation problem and thus achieve dramatically faster restore and GC performance, which we call **management-friendly**. In our implementation of the OPT data layout using an offline method of iteratively arranging chunks for each incoming backup version, we find the costs are acceptable, especially compared with the huge overheads of restore and GC in previous write-friendly approaches.

To this end, we propose a novel Management-Friendly Deduplication framework, called MFDedup, that introduces two new techniques: Neighbor-Duplicate-Focus indexing (NDF) and Across-Version-Aware Reorganization scheme (AVAR). Together, they generate and maintain the OPT data layout, which eliminates the fragmentation problem. Specifically, the contributions of this paper are three folds:

- We propose NDF to only detect duplicates of a backup version (B_{i+1}) with its previous version (B_i), which utilizes our observation, and provides an opportunity to build the OPT data layout. NDF significantly reduces the memory footprint for the fingerprint index while achieving a near-exact deduplication ratio.
- After deduplicating the new version B_{i+1} using NDF, AVAR arranges the unique chunks of B_{i+1} into the OPT data layout by classifying and grouping according to the simplified reference relationship between chunks and versions. By iteratively updateing the OPT layout, GC becomes a simple operation of immediately deleting the oldest categories as the oldest versions are deleted.
- Evaluation results with four backup datasets suggest that, compared with previous approaches, MFDedup achieves a significantly higher deduplication ratio (1.66× to 3.28× higher) and restore throughput (2.63× to 11.64× higher). Restore throughput fully utilizes the storage devices. Meanwhile, the NDF index is a fixed and limited overhead compared with traditional global index, and GC in MFDedup has nearly zero-overhead.

2 Background and Related Work

2.1 Background of Data Deduplication

Data deduplication is a widely used data reduction approach for storage systems [8,12,27,33,34,38,44,45]. In general, a typical data deduplication system splits the input data stream (e.g., backup files, database snapshots, virtual machine images, etc.) into multiple data "chunks" (e.g., 8KB size) that are each uniquely identified with a cryptographically secure hash signature (e.g., SHA-1), also called a fingerprint [29,34].

Deduplication systems then deduplicate data chunks according to their fingerprints and store only one physical copy to achieve the goal of saving storage space.

Backup storage and locality. *Backup storage* often leverages data deduplication due to the highly redundant nature of the data. In backup storage systems, workloads usually are a series of backups versions (i.e., successive snapshots of the primary data), and the size of backups can be greatly reduced to about 1/10-1/30 of their original size, reducing hardware costs. *Locality* in backup workloads means that the chunks of a backup stream will appear in approximately the same order in each full backup with a high probability, which is widely exploited for improving deduplication performance, such as for fingerprint indexing, restoring, etc., by utilizing the high sequential I/O speed of HDDs.

Container-based I/O. Many deduplication-based storage systems usually combine with compression techniques, and all chunks are stored in containers as the basic unit for compression. Thus, storage I/O are usually based on containers. Usually, containers are immutable and have a fixed size (e.g., 4MB). Containers offers several benefits: ① Writing in large units achieves the maximum sequential throughput of hard drives and is compatible with striping across multiple drives in a RAID configuration. Hard drives remain significantly cheaper than SSDs and other media, and cost is an important consideration for backup storage. ② The locality of data in containers is frequently leveraged to improve the efficiency of identifying duplicates as well as for restoring backups to clients [45].

Fragmentation Problem. Fragmentation in deduplication systems is related to container-based I/O and the seek latency of HDDs. This is because different backups will share chunks, and these shared chunks are randomly distributed across containers. In other words, *spatial locality* of each backup will be destroyed after deduplication. Due to container-based I/O, when we restore a backup, even if only a few shared chunks in a container are required, we have to read the whole container from HDDs, which is sometimes called *read amplification* (defined as $\frac{Total\ Size\ of\ Loaded\ Containers}{Size\ of\ Actually\ Restored\ Data}$ during restores). Even if a system supports compression regions within a container, a full compression region must be read and decompressed to supply a needed chunk. In addition, since the required containers for each backup are randomly distributed across the HDDs, *seeking* to these required containers on HDDs is also time-consuming. Moreover, the read amplification and seek issues become worse as the number of backups increases.

2.2 Deduplication Techniques

A typical deduplication system usually consists of several techniques, including chunking, fingerprint indexing, restore optimizations, garbage collection, etc.

Chunking Techniques. Content-Defined Chunking (CDC) [13, 29, 32, 41, 42] is a widely used chunking approach to split the backup stream into variable-sized chunks according to the content, which can handle the 'boundary-shift' problem existing in Fix-Sized Chunking [34].

Fingerprint Index Techniques. Checking the fingerprint index (i.e., detecting duplicates) is a critical step in the workflow of deduplication. Fingerprint indices grow as a fraction of backup storage system capacity, so keeping them in memory is expensive and impractical while putting them in HDDs will cause a deduplication system bottleneck for indexing. Several approaches [5, 7, 17, 24, 26, 28, 40, 45] have been proposed, and most leverage spatial or temporal locality by using the fingerprint index to load many fingerprints from disk that were written at the same time or consecutively in a file.

Restore Optimization Techniques. As introduced in Section 2.1, fragmentation introduces read amplification and many disk seeks when restoring a backup after deduplication. Among the restore optimization approaches, there are two main approaches to review that can be used separately or together: '*rewriting*' and '*cache*'. Rewriting trades-off deduplication space savings to improve locality by selectively writing duplicates [9, 10, 14, 22, 23, 30, 31]. Rewriting lowers the deduplication ratio, and results show read amplification remains $2\times \sim 4\times$ after rewriting. The caching approach uses SSDs or memory to cache chunks that are frequently referenced or believed to be needed in the near future [2, 25], but the cache hit ratio still depends on locality, and read amplification is not addressed.

Among rewriting approaches, Capping [23] follows a simple policy. When deduplicating against a previously-written container, record how many chunks in the container are referenced for the current backup. For containers with low reuse, it rewrites chunks to improve the locality of the current backup version. HAR [15], utilizing the similarity of backup streams, identifies sparse containers according to historical information, and rewrites chunks which refer to those containers. In contrast, our approach writes minimal duplicate chunks and creates a data layout without any fragmentation or read amplification.

Garbage Collection Techniques. Customers usually configure a retention policy for backup files using their backup software, which often involves retaining weeks or months of backups and deleting backups older than the retention policy. GC then removes unreferenced chunks from the system [11, 14, 17, 37]. There are generally two kinds of GC in deduplicated systems. The first one is traditional Mark-Sweep [11,17,37]: it walks the backups and marks the chunks referenced from those backups, and then the unreferenced chunks are swept away. In practice, this often requires copying live chunks from a partially-unreferenced container and forming new containers. Although numerous optimizations have been proposed [11, 14, 17, 37], copying live chunks and writing new containers is I/O intensive [15].

The second approach is the Container-Marker Algorithm (CMA) [15]. CMA maintains a container manifest to record referenced backups for each container and then deletes the whole containers that are unreferenced, which is a coarse-grained GC approach that maintains containers if any chunks are still referenced and thus causes wasted space. In contrast, we focus on fine-grained GC, but have an approach with dramatically lower overheads than previous techniques.

3 Observation and Motivation

3.1 Analysis for Fragmentation and Read Amplification after Deduplication

As discussed in Section 2.1, fragmentation causes serious read amplification in deduplication systems using container-based I/O. In this subsection, we analyze the cause of read amplification with a detailed example.

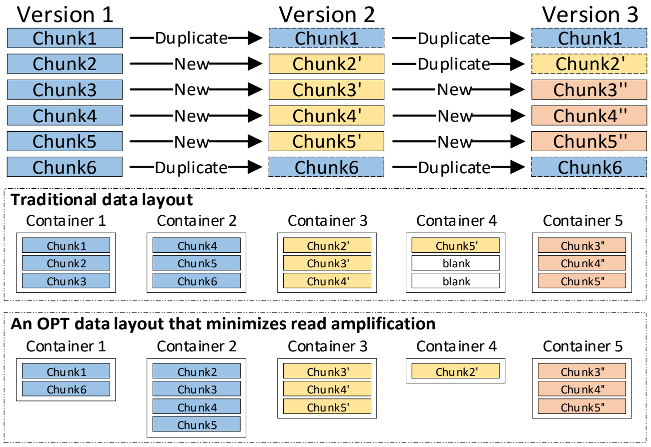


Figure 1: Examples of running exact deduplication on three backup versions with a traditional data layout versus an optimal data layout based on classification.

In traditional deduplication systems, after deduplication, all remaining chunks are stored in containers in the order they appear in a backup. Figure 1 shows an example of the traditional data layout after deduplication of three backup versions. Deduplicated chunks from three backup versions are stored in five containers using the traditional data layout, and *Chunk* 6 is referenced by all the three versions. Because of the container-based I/O, no matter which version we want to restore, we always need to read *Container* 2 from HDDs.

Read Amplification. For *Versions* 2 and 3, only *Chunk* 6 is needed from *Container* 2, which includes other chunks unreferenced by *Versions* 2 and 3. This is an example of poor spatial locality. Therefore, loading *Container* 2 causes read amplification (i.e., we read two unneeded chunks) when restoring these two versions. But for *Version* 1, all chunks in *Container* 2 are required, which means a strong spatial locality, and there is no fragmentation and read amplification. Note that under the traditional data layout, *Chunk* 6 is a fragmented chunk for *Versions* 2 and 3, but is not a fragmented chunk for *Version* 1, which means fragmentation is dependent on the backup version and associated with chunks' reference relationship.

3.2 An Optimal Data Layout

In this subsection, we present and discuss a classification-based optimal (OPT) data layout according to the chunks' reference relationships as mentioned in the last subsection.

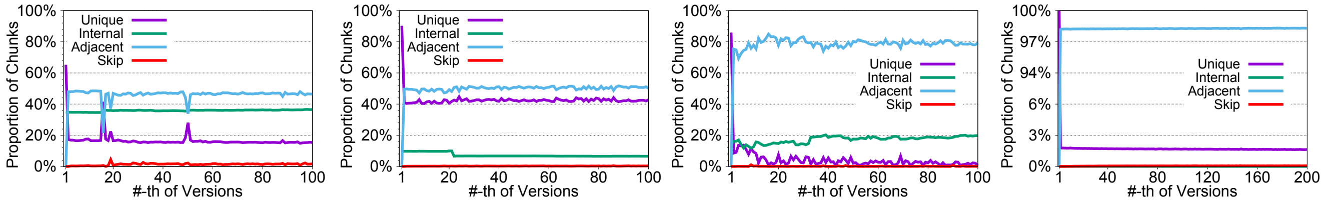
An Example of Classification: For the three chunks {4, 5, 6} in *Container* 2 (in the traditional data layout in Figure 1), according to their reference relationship, we can classify them into two categories (like containers). The first category includes *Chunks* 4 and 5, which are only referenced by *Version* 1, and the second category is *Chunk* 6, which is referenced by all three versions. If we store two categories separately in different containers, we could load both categories when restoring *Version* 1 and load only the second category for *Versions* 2 and 3. In this way, the fragmentation problem of these three chunks are resolved, and there will be no read amplification when restoring any of the versions.

Classification-based Data Layout. Here we continue the previous example and classify all chunks into five categories (like containers) according to their reference relationship and then *store each category into a variable-sized container*, as shown in the 'OPT data layout' in Figure 1:

- *Container* 1 is referenced by Versions {1, 2, 3}.
- *Container* 2 is referenced by Version 1.
- *Container* 3 is referenced by Version 2.
- *Container* 4 is referenced by Versions {2, 3}.
- *Container* 5 is referenced by Version 3.

This layout keeps strong spatial locality for each backup version with a read amplification of 1, so we refer to it as the OPT data layout that minimizes read amplification. For example, if we want to restore *Version* 3, we need to load *Containers* $1, 4$ *and* 5, which does not load any unrequired chunks. Meanwhile, there is also no read amplification when restoring *Versions* 1 and 2. Therefore, Figure 1 provides a possible solution to eliminate fragmentation for that example of three backup versions. In the worst case of three backup versions, there will be seven categories (i.e., total number of $\binom{3}{1}+\binom{3}{2}+\binom{3}{3}=2^3-1$ reference relationship). Here $\binom{n}{k}$ means choosing k from n elements.

Challenges for OPT Data Layout. Actual backup workloads are much more complicated than the example shown in Figure 1. Specifically, if we follow the idea of classification on n backup versions, there will be 2^n-1 (i.e., $\binom{n}{1}+\binom{n}{2}+...+\binom{n}{n}=2^n-1$) categories that are stored as 2^n-1 containers. Assuming there are 30 backup versions with about 1 million unique 8KB chunks (totaling 8 GB after deduplication), there will be more than 1,000,000,000 containers after classification and thus most of the time each container has only one or very few chunks. In other words, this OPT data layout solves the read amplification problem but requires more seek operations for these very small containers, which also causes poor data locality.



(a) WEB Dataset (b) CHM Dataset (c) VMS Dataset (d) SYN Dataset

Figure 2: Distribution of four kinds of chunks on four backup datasets. *Skip* duplicate chunks are the least common.

3.3 Derivation Relationship of Backups

In this subsection, we will present our key observation about the deduplication relationship of backups through an analysis of four large backup datasets. These relationships can be exploited to greatly decrease the number of categories (i.e., containers) needed for the OPT data layout.

In backup storage systems, workloads usually consist of a series of backup images, which are all generated from the original data on a primary storage system (i.e. laptop, server, database, etc.). Therefore, the duplicate chunks of each backup are not randomly distributed but are derived from the chunks of the last backup as we will show with experiments, which is consistent with the typical consecutive pattern of duplicates that is leveraged by many systems [24, 40, 45] for high deduplication performance.

To better illustrate our observation, we denote four kinds of chunks in a backup version B_i as follows:

- *Internal duplicate chunks*, whose referenced chunks are also in B_i.
- *Adjacent duplicate chunks*, whose referenced chunks are not in B_i but in the last version B_{i-1}.
- *Skip duplicate chunks*, whose referenced chunks are neither in B_i nor in B_{i-1}.
- *Unique chunks*: the non-duplicate chunks.

Key Observation. Figure 2 studies the distribution of the four kinds of chunks on four backup datasets running with exact deduplication. From Figure 2, we can observe that most duplicate chunks for a backup version are from the previous version (*Adjacent*) and within the current backup itself (*Internal*). Adjacent and Internal account for more than 99.5% in most datasets. The *Skip* duplicate chunks only consist of a small fraction (less than 0.5% in most datasets) of all duplicate chunks. This observation supports an approach of avoiding deduplicating Skip chunks to preserve locality since they would only have a small impact on the deduplication ratio.

Motivated by the above observation of the duplicate chunks' pattern, we avoid deduplicating Skip chunks and treat them as Unique chunks for the current version. This greatly simplifies chunk-reference relationships: each physical chunk must be referenced by one version or by consecutive versions (e.g., B_i, ... B_{i+k}) in the former OPT data layout. With this condition, we can greatly reduce the number of classified categories (containers). Taking three backups for example, because each chunk must be referenced by successive versions $\{B_i, ..., B_{i+k}\}$, where $i \geq 1$ and $i+k \leq 3$. Thus, when k=0, there are $\binom{3}{1}$ categories; when $k \neq 0$, there are $\binom{3}{2}$ categories (choosing the start and the end one for successive versions). Hence, the number of categories in this example is reduced from seven to six.

In general, if we have n backup versions and improve the OPT data layout by exploiting the duplicate chunks' pattern, the upper limit of the number of categories (containers) will be $\binom{n}{1} + \binom{n}{2} = n(n+1)/2$, which is much less than $2^n - 1$. Continuing an earlier example with 30 backup versions, there will be 465 containers with about 2150 chunks on average, so the average size of container is about 17.6MB. This condition makes the classification feasible for the OPT data layout while the size of containers is large enough to maintain the spatial locality of backup workloads.

4 Design and Implementation

4.1 MFDedup Overview

Based on our key observations about the relationships between backups and the OPT data layout mentioned in Section 3, implementing the optimal data layout in a deduplicated backup storage systems is feasible by the following two key design principles: ① *All chunks are classified into categories (like containers) according to their reference relationship.* ② *Skip duplicate chunks are treated as unique chunks to simplify the chunks' reference relationship.*

In this paper, we propose our approach MFDedup, a management-friendly deduplication framework using the aforementioned optimal (OPT) data layout. Our approach is to maintain the locality of backup workloads, which eliminates fragmentation that slows restore and GC. MFDedup follows the above two design principles and reorganizes chunks in an offline algorithm to achieve the OPT data layout, by using two key techniques:

- **Neighbor-Duplicate-Focus indexing (NDF)**. MFDedup only removes duplicates between neighboring backup versions. Hence, we only need to build and access a local fingerprint index consisting of the neighboring backup versions' fingerprints, whose resource requirements are lower compared with traditional global fingerprint index, as detailed in Section 4.2.
- **Across-Version-Aware Reorganization (AVAR).** After detecting duplicate chunks of each new backup ver-

sion using NDF, MFDedup offline arranges the deduplicated chunks of the last backup version. Specifically, these chunks are classified and grouped to iteratively update the OPT data layout according to our simplified chunk-reference relationship, as detailed in Section 4.3.

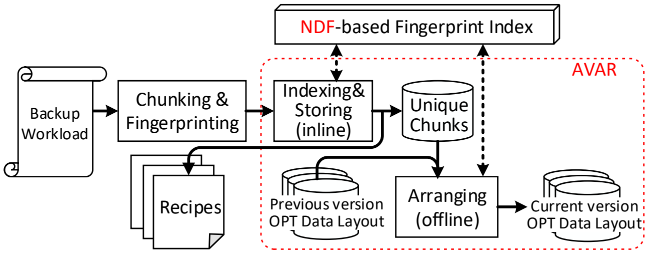

NDF-based Fingerprint Index
AVAR
Backup Workload
Chunking & Fingerprinting
Indexing& Storing (inline)
Unique Chunks
Recipes
Previous version OPT Data Layout
Arranging (offline)
Current version OPT Data Layout

Figure 3: An overview of MFDedup framework.

The overall workflow of the MFDedup framework is shown in Figure 3, which includes three key stages: *Chunking & Fingerprinting, Indexing & Storing, and Arranging*. Chunking & Fingerprinting refers to splitting the backup stream into chunks using Content-Defined Chunking [29, 39] and then calculating a fingerprint (i.e. SHA1 digest) for each chunk. Indexing & Storing detects duplicate and unique chunks from the previous backup version by using NDF-based fingerprint index and then stores unique chunks and a **Recipe** for each backup. Arranging is an offline process, which iteratively updates the OPT data layout version by version, with the support of NDF-based fingerprint index. Note that the Recipe records the chunk-fingerprint sequence of a backup version, which is used to recover the backup version after deduplication.

In general, MFDedup applies online deduplication using NDF, which removes duplicates only between neighboring backup versions, and then an offline Arranging using AVAR, which keeps the OPT data layout to maintain locality of backup workloads and thus eliminate fragmentation.

4.2 Neighbor-Duplicate-Focus Indexing

In this section, we will introduce the Neighbor-Duplicate-Focus indexing (NDF) technique in MFDedup, which is based on our observation (Section 3.3) that most duplicate chunks exist between neighboring versions in a backup storage system (i.e., duplicate chunks of backup version B_i are nearly all from its previous version B_{i-1}). Therefore, MFDedup chooses to treat *Skip* duplicate chunks as unique chunks instead of deduplicating them. In other words, MFDedup only identifies duplicate chunks in backup version B_i that are identical to chunks either in B_i (within the same version) or B_{i-1} (the previous version). We refer to this as a **NDF-based fingerprint index**.

In the NDF implementation, we maintain an independent fingerprint index table for each backup version. Besides using NDF in the Indexing & Storing stage of MFDedup for duplicate detection, NDF is also used in the *Arranging* stage for classification as detailed in next subsection. After a fingerprint index table is used in the above two stages for the latest two backup versions, it can be released. Therefore, we only need to maintain two fingerprint indices, which could be kept in memory if they are small (they are typically much smaller than the traditional index that stores all versions) or loaded using previous locality-based approaches [24, 45]. Assuming a fingerprint index entry takes 20 bytes (i.e., the size of SHA1 digest), the size of a backup version is 10GB, and the expected chunk size is 8KB, the total memory cost of NDF-based fingerprint index will be $2 \times 10GB/8KB \times 20B = 50MB$ (about 0.4882% size of a backup version).

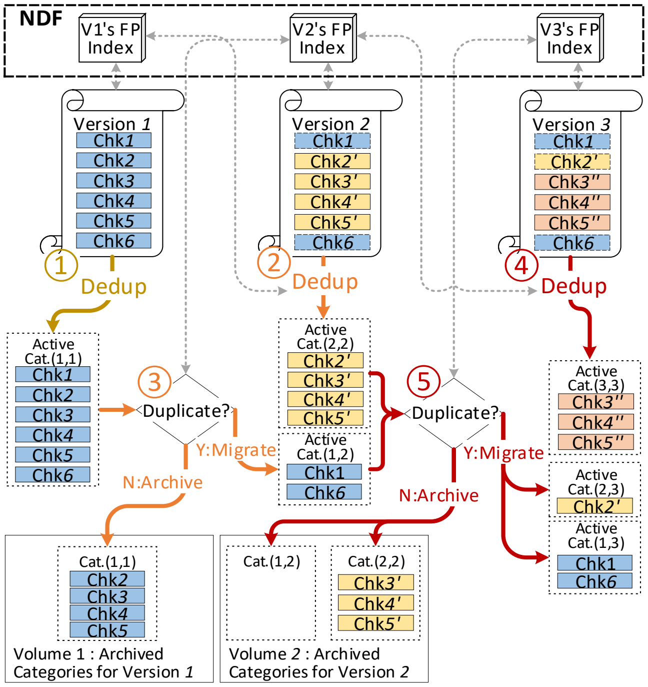


Figure 4: An example of the AVAR workflow on three backup versions, which is presented by a solid line in five steps: ① *Deduplicating* Version 1 → ② *Deduplicating* Version 2 → ③ *Arranging* Version 1 → ④ *Deduplicating* Version 3 → ⑤ *Arranging* Version 2. Gray dashed lines refer to fingerprint indexing operations.

Indexing overhead of NDF is related to the data size of two most recent backup versions, which is considerably smaller than traditional deduplication systems that have a global fingerprint index. Meanwhile, NDF is able to achieve a near-exact deduplication ratio while supporting the OPT data layout in MFDedup (detailed in Section 4.3).

4.3 Across-Version-Aware Reorganization

In this subsection, we will introduce Across-Version-Aware Reorganization (AVAR) in MFDedup, which is designed to eliminate fragmentation and generate the OPT data layout combining with the NDF technique.

There are two stages in AVAR, which are the *Deduplicating* stage (i.e., *Indexing & Storing* in Section 4.1) and the *Arranging* stage, and both utilize the NDF-based fingerprint index.

Figure 4 gives an example of AVAR on three backup versions, running with the two stages alternating (except the first backup version): The Deduplicating stage identifies unique chunks of a backup version B_i. Then the *Arranging* stage updates the OPT data layout for backup version sets $\{B_1..B_i\}$, by reorganizing the previous OPT data layout of $\{B_1..B_{i-1}\}$ with the unique chunks of B_i. Note that there is naturally an OPT data layout when the first backup version is stored, so the Arranging stage is not required after the Deduplicating stage. More details about the two stages of AVAR are elaborated below.

Deduplicating* Stage.** In this stage, we detect duplicate chunks using the NDF-based fingerprint index and then store unique chunks as well as Recipes. In the remainder of this section, we ignore Recipes, and focus on how data chunks are managed for the OPT data layout. Therefore, as shown in Steps ①, ② and ④ of Figure 4, the Deduplicating Stage is responsible for storing unique chunks of the latest new backup version B_i into a new active ***Category, which is currently only referenced by B_i. Note that chunks in the active Category may be referenced by future backup versions and thus will be processed by the Arranging stage later.

Note that each Category is named with a pair of numbers in MFDedup, which reflects which backup versions refer to chunks in this category. For example, if chunks in a category are referenced from consecutive Versions 2, 3, and 4, we denote this category as **Cat.(2, 4)**.

***Arranging* Stage.** According to the 1^{st} principle of MFDedup, classification methods used for generating the OPT data layout are based on the reference relationship between chunks and backup versions, which means the old OPT data layout expires when a new version arrives and is processed by the *Deduplicating* stage. This is because the reference relationship between chunks and backups has changed. Therefore, the Arranging stage is responsible for iteratively updating the existing OPT data layout with new unique chunks of the incoming version (after the Deduplicating stage), following the two design principle of MFDedup.

To better present the iterative process of our Arranging stage, Figure 5 shows a general evolution example of OPT data layout with three backup versions. According to the design principle ② in Section 4.1 (i.e., *Skip* duplicate chunks are ignored and treated as unique chunks), in backup version sets $\{B_1..B_{n-1}\}$. Then Cat.(1,n-4) is not referenced by the last backup version B_{n-1} and cannot be referenced by B_n and later backup versions. As a result, these kinds of categories are always carried forward as part of the OPT data layout and are referred to as **archived**. On the other hand, Cat.(1,n-1) that is referenced by the last backup version B_{n-1}, will be split into two categories when backing up a new version B_n. We call those categories **active**.

Therefore, in our implementation of AVAR, classified categories (containers) have two states: **Active** and **Archived** when updating the OPT data layout. Archived means the categories are immutable, while Active means the categories will be further 'arranged' by MFDedup after future backups. More specifically, in the Arranging stage of AVAR, we focus on active categories, which derive new active categories and archived categories. Like the example of *Step* ⑤ shown in Figure 4, Cat.(1,2) and Cat.(2,2) are the only two existing (old) active categories after backing up Version 3, and we check each chunk of them with the fingerprint index of backup version 3. Duplicate chunks, existing in Version 3, are migrated to new active Cat.(1,3) and Cat.(2,3), and other chunks are arranged in archived Cat.(1,2) and Cat.(2,2). After migrating and archiving, old active Cat.(1,2) and Cat.(2,2) are no longer required and thus deleted.

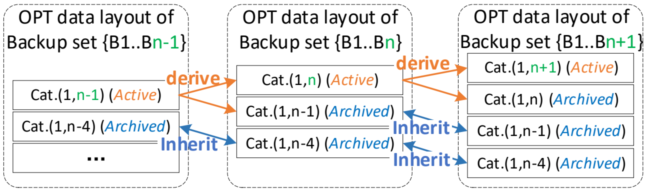


Figure 5: An example of the OPT data layout's evolution on three backup versions. Some categories are inherited from the previous version, and some derive new categories.

Grouping. After Arranging existing Active categories, the new Archived categories are grouped into a **Volume** by the order of their name (e.g. in the order of Cat.(1,3), Cat.(2,3), Cat.(3,3)), for easier storage management. The benefits of grouping categories will be introduced in the next section.

4.4 Restore and Garbage Collection

Restore and Garbage Collection both benefit from our OPT data layout in MFDedup, and their workflows are greatly simplified as elaborated in this subsection.

Restore. When restoring a backup version in MFDedup, we only need to read the required categories on the OPT data layout, which is referenced by the to-be-restored version. Meanwhile, tracing required chunks (categories) for restore is totally metadata-free in MFDedup with the support of the OPT data layout (i.e., can be calculated by our layout).

For the situation that there are n backup versions stored in MFDedup, and we want to restore a backup version B_k, all categories referenced by B_k are required. For example, Cat.(3,k+2) is required, because it is referenced from B_3 to B_{k+2}, which includes B_k. Thus, all required categories for B_k could be represented as:

$$\begin{aligned} Required\ Cat. &= \{Cat.(i,j)\}, where\ 1 \le i \le k \le j \le n \\ &= \cup_{j=k}^{n} \cup_{i=1}^{j} Cat.(i,j). \end{aligned} \tag{1}$$

Like the example of Figure 6, there are four stored backups. According to Equation 1, restoring Version 3 requires the blue-colored categories. Note that according to our grouping approach, $\cup_{i=1}^{j} Cat.(i,j)$ are always sequentially grouped in the same Volume. Thus, loading $\cup_{j=k}^{n} \cup_{i=1}^{j} Cat.(i,j)$ requires n sequential reads at most.

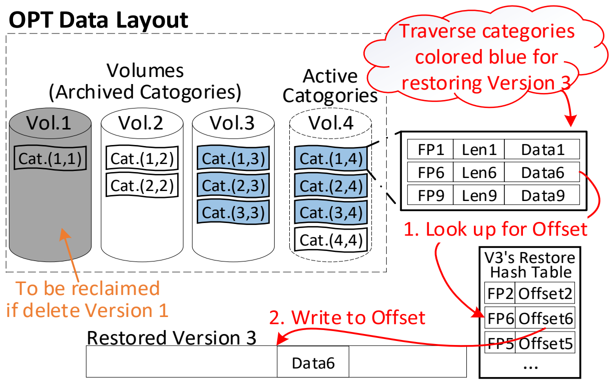


Figure 6: An example of restore and deletion on the OPT data layout with four backup versions.

A recipe is required to restore a backup version, which is used to build a ‘restore’ hash table, whose format is shown in Figure 6. The entry of the hash table is a pair like <fingerprint, offset>, which records a chunks’ fingerprint and its offset in the to-be-restored version.

The workflow of restore is shown in Figure 6, after getting the required categories, the chunks are restored one by one according to the Recipe for Version 3. Therefore, MFDedup only needs to seek to the required volumes and then sequentially read the required (consecutive) categories in those volumes, which achieves a superior restore performance (i.e., few seeks and large sequential I/Os).

Deletion and Garbage Collection. As a result of our OPT data layout, deletion and garbage collection are naturally simple, and the space can be immediately reclaimed in MFDedup. In deduplication systems, deleting a backup version means reclaiming its unique chunks (those not referenced by other backups). FIFO-based deletion in MFDedup simply deletes and reclaims the earliest volumes, because they consist of unique chunks of the earliest backup versions. For example in Figure 6, we can reclaim space of *Version* 1 by directly deleting *Volume* 1.

MFDedup also supports deleting other backup versions besides the earliest ones. From the description of the Deduplicating stage, we see that the unique chunks of each backup version are always stored in the last category of each volume (see Figure 6). Thus, we can also delete any backup version by resizing the corresponding volume using ‘*truncate()*’ (i.e., deleting the last category in this file). For example, if we want to delete *Version* 2 in Figure 6, we can remove the archived Category 3 by just truncating *Volume* 2. A previous work [11] mentioned that the CMA approach [15] only supports FIFO deletion, while we support any deletion pattern.

In this way, we no longer apply traditional GC techniques in MFDedup, such as mark-sweep or reference-count management, since the chunk-reference relationship is naturally designed into our classification-based OPT data layout. This is a dramatic reduction in system resources (CPU cycles, RAM, I/O) and coding complexity.

4.5 Discussion and Limitations

In this subsection, we discuss overheads, limitations, and corresponding possible optimizations of MFDedup in a deployed system to support various backup workloads.

Self-Organization of OPT data layout. This OPT data layout is self-organized and simple, and the cost of metadata is greatly reduced. The exact physical position and the reference counts of each unique chunk, which are usually used for restore and GC in traditional deduplication systems, are not needed for MFDedup. For example, in restore, the required categories for each version are calculable in the OPT data layout.

Backups Size. While backup sizes can vary over a wide range, many VM backups are $\sim$100GB, and the index is 400MB for the most recent virtual machines. Wallace [38] and Amvrosiadis [4] also suggested the majority of backups were 50-500GB in Data Domain and Symantec production systems. Hence MFDedup can be directly applied in these scenarios with a reasonable memory overhead.

Fingerprint Prefetching for Larger Backups. Although the current design of MFDedup has the index in RAM, previous techniques for prefetching and caching sequences of fingerprints (designed for a large fingerprint index) [5,7,17,24,26,28,40,45] could also be used in MFDedup. We expect the sequential locality to also exist in MFDedup for two reasons. On the one hand, sequential locality exists inside categories, although it is destroyed across categories by Arranging. On the other hand, recipes also keep the sequential locality of each backup.

Restoring for Larger Backups. When restoring a single large backup, since chunks are organized with categories in MFDedup, we could also organize a ‘restore’ hash table (recording pairs <fingerprint, offset>) for each category, and then load the hash tables to memory separately. Besides, a single backup could be divided by MFDedup into several smaller sub-units (e.g., each <100GB) to relieve the memory burden for both backing up and restoring. But, when the size of a single backup is huge (such as over $10TB$), and there is only one category for this backup, the hash table would also be very large (over $10TB/8KB \times (20B+8B) = 35GB$, here we assume ‘fingerprint’ and ‘offset’ take 20B and 8B, respectively), which is difficult to maintain in memory, so MFDedup can not handle these use cases yet.

Incremental Backups vs. Full Backups As we introduced in Section 4, MFDedup is designed for full backups. For incremental backups, we could add an API to distinguish between incremental and full backups. Also, as synthetic full backups have already become broadly used, the incremental changes are typically relative to the last “full” backup synthesized, so MFDedup can also be directly applied.

Reserved Space for Arranging. Arranging is an offline process, in which chunks are migrated or archived, and it requires additional reserved space. As shown in Figure 4, Arranging runs on active categories, thus, the reserved space is equal to the maximum size of active categories, which is much smaller than a full backup and is studied in Section 5.6.

What if Arranging Falls Behind. If there are a lot of workloads to back up and not enough time to finish the *Arranging* stage in MFDedup, we can skip it temporarily, and apply it in future idle time. Before Arranging catches up, the OPT data layout is not updated with new incoming backup versions. The more Arranging falls behind, the more seriously OPT data layout is damaged, with an increase in read amplification and decrease in restore throughput. However, this a rare case since users usually create full backups daily or less frequently [3, 23], which provides enough time for our offline Arranging. In addition, a higher deduplication ratio leads to a smaller read amplification and also a smaller reduction in restore throughput when Arranging falls behind.

Time Overhead of Offline Arranging. In MFDedup, we have transferred background work from GC to Arranging while achieving many benefits: high restore speed, immediate space reclamation, etc. Arranging is an offline process that traverses active categories of a backup version, migrates duplicate chunks, and archives the remaining chunks. The time cost of Arranging is close to or better than perfect garbage collection with the benefits of better restore and GC performance of MFDedup as evaluated in Section 5.5. In the paper, we always run Arranging after each backup to keep the data layout healthy (i.e., optimal), but Arranging could act like GC: just running once after several backups. In this case, Arranging falls behind and will cause slight read amplification, as discussed in “What if Arranging Falls Behind”. Besides, several Arranging tasks, in which duplicate chunks will be migrated several times, could be merged in this situation, which could reduce the total overhead for Arranging, though this has not been evaluated.

Out-of-Order Restore. Unlike the implementation of the traditional deduplication framework, restore in MFDedup is out-of-order, which means the writing order of restored chunks does not absolutely follow their logical order in workloads. While the chunks in volumes are generally in order for a backup, there are logical gaps that are filled by other volumes, which causes random writes to the restored version. Although the sequential locality still exists inside categories, as discussed in “Fingerprint Prefetching”, restore will have better performance if the destination media has good random write performance (e.g., on SSDs). Besides, some previous techniques, like a reassembly buffer [20], could be applied to improve the performance when streaming a restore to HDD devices.

5 Performance Evaluation

5.1 Experimental Setup

Evaluation Platform and Configurations. We perform our experiments on a workstation running Ubuntu 18.04 with an Intel Core i7-8700 @ 3.2GHz CPU, 64GB memory, Intel D3-S4610 SSDs, and 7200rpm HDDs.

Table 1: Four backup datasets used in evaluation.

Name	Total Size Before Dedup	Versions	Workload Descriptions
WEB	269 GB	100	Backup snapshots of website: news.sina.com, captured from June to September in 2016.
CHM	279 GB	100	Source codes of Chromium project from v82.0.4066 to v85.0.4165
VMS	1.55 TB	100	Backups of an Ubuntu 12.04 Virtual Machine
SYN	1.38 TB	200	Synthetic backups by simulating file create/delete/modify operations [36]

In our evaluation, we built a MFDedup prototype system and also built Destor [16] for comparison with several state-of-the-art techniques for restore and GC, including the History-Aware Rewriting algorithm (HAR) [15], Capping [23], and Container-Marker Algorithm (CMA) [14]. MFDedup and Destor use the same configuration in the Chunking & Fingerprinting stage: chunking uses FastCDC [41] with the minimum, average, and maximum chunk sizes set to 2KB, 8KB, and 64KB; Fingerprinting uses a SHA1 digest generated by Intel Intelligent Storage Acceleration Library Crypto Version (i.e., ISA-L_crypto [1]).

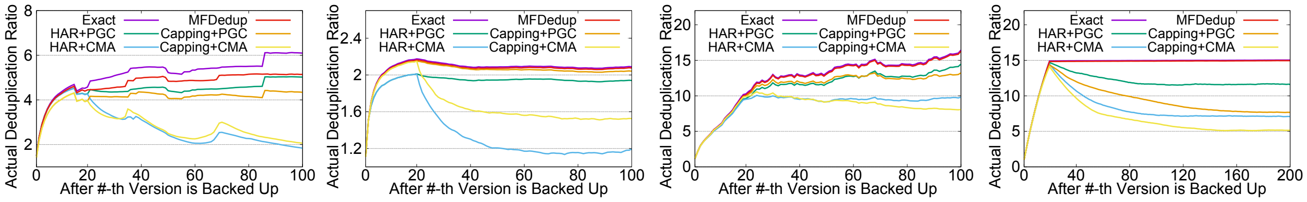
Experimental methods. To simulate real backup/restore scenarios, we separate the storage space of our workstation into two parts: a backup space using a 7200rpm HDD and a user space using an Intel D3-S4610 SSD. Both spaces (drives) have an XFS file system. It is typical in backup environments to use HDDs for cost reasons while primary systems often use SSD for higher performance.

To evaluate backup/restore performance, tested datasets are backed up from the user space to the backup space version by version while the restore runs in the reverse direction. Note that before each backup/restore, we always flush the file system cache using the command: “echo 3 > /proc/sys/vm/drop_caches”.

To simulate users’ retention (deletion) requirements in backup systems, we retain the most recent 20 versions. Thus Version $n-20$ is deleted after Version n is backed up, which is the same as the previous work HAR [15] and CMA [14]. For throughput (time cost) of backup, restore, and GC/Arranging in our evaluation, we present the average results of five runs.

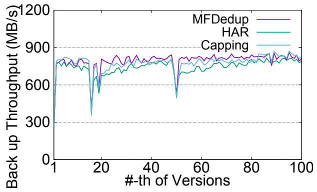
Container-based I/O is considered, because many deduplication-based storage systems usually combine with compression techniques, and all chunks are stored in containers as the basic unit for compression. Because here we focus on deduplication and compression techiniques are orthogonal to deduplication, we do not introduce compression in evaluations.

Evaluation Dataset. Four backup datasets are used for evaluation as shown in Table 1. These datasets represent various typical backup workloads, including website snapshots, an open source code project, virtual machine images, and a synthetic dataset, with deduplication ratios varying from 2.19 to 44.65. WEB, SYN, and VMS datasets have been used in several studies of data deduplication [15, 41, 43].



(a) WEB (b) CHM (c) VMS (d) SYN

Figure 7: Actual Deduplication Ratio of MFDedup and five approaches running on four datasets (retaining 20 backups).



(a) WEB (b) VMS

Figure 8: Backup throughput of MFDedup and two other rewriting approaches running on two selected datasets (due to space limit), without considering (offline) Arranging.

5.2 Actual Deduplication Ratio

As mentioned in Section 2, rewriting and GC techniques (e.g. HAR, Capping, and CMA) consume more storage space in exchange for better restore and GC performance. Meanwhile, MFDedup ignores Skip duplicate chunks to implement OPT data layout, which also reduces the deduplication ratio. Hence, in this subsection, we evaluate MFDedup and other approaches with the *Actual Deduplication Ratio* (denoted by ADR) defined as $\frac{Total\ Size\ of\ the\ Dataset}{Size\ after\ Running\ an\ Approach}$, which reflects the corresponding reduced deduplication ratio due to these techniques (such as rewriting).

Figure 7 shows ADR of MFDedup, Exact Deduplication, and other approaches, including combinations of rewriting (HAR and Capping) and GC (Perfect GC and CMA) techniques. Here MFDedup includes its GC approach. Note that we only retain the latest 20 backup versions in our evaluation, and thus Perfect GC and CMA represent two typical GC techniques using Mark-Sweep with utilization thresholds set at 0% and 100%, respectively. Perfect GC reclaims all possible space, while CMA runs faster but leaves unreferenced chunks in containers that are partially referenced, so they show two kinds of extreme impacts of GC.

Generally, Figure 7 shows that MFDedup achieves ADR that is very close to Exact deduplication, which is much higher than other rewriting and GC approaches. This is because the space cost of ignoring Skip duplicate chunks in MFDedup is quite small, especially compared with the number of rewritten chunks in other approaches.

Figure 7 also shows rewriting techniques cause a decrease in ADR when GC starts after version 21. When the CMA technique (higher GC speed, fewer unreferenced chunks removed) is added, this loss worsens. This is consistent with our discussion in Section 2: rewriting reduces deduplication while GC also can lead to more rewritten chunks. Meanwhile, deletion and GC are naturally supported in our OPT data layout with NDF and AVAR techniques, which has no fragmentation issue and thus no space cost for MFDedup. Overall, MFDedup achieves a 1.12× to 2.19× higher ADR than other approaches due to the OPT data layout.

5.3 Backup Throughput

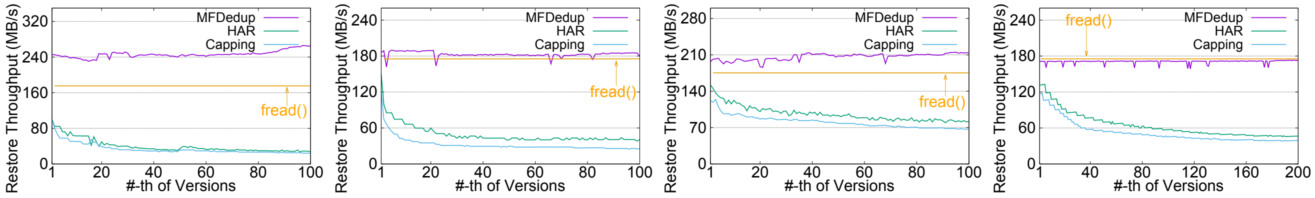
In this section, we study backup throughput of MFDedup compared with rewriting approaches. Here we do not consider the impact of GC since it is usually an offline process. Both HAR and Capping use a full-in-memory global fingerprint index while MFDedup applies NDF-based local fingerprint index. To minimize the performance impact of reading datasets, we back up the datasets from a ramdisk to measure the backup throughput.

Figure 8 shows backup throughput of the three approaches, which have similar results for a given dataset. This highlights that MFDedup does not sacrifice backup throughput to achieve the other benefits we discuss. The performance of the three techniques is similarly limited by the chunking and SHA1 digest calculation. In theory, since MFDedup no longer rewrites duplicate chunks (thus achieving higher Actual Deduplication Ratio in Section 5.2), its storage I/O when backing up will also be smaller than the traditional design.

Indexing Overhead. During backups, we measured the maximum memory cost for the NDF index, which varied from 6.27MB to 46.35MB (only indexing 2 backup versions). In contrast, traditional deduplication approaches maintain a global fingerprint index for all 20 backup versions and would require 26.81MB to 64.45MB space. Note that the traditional global index grows with the number of retained versions, while NDF only maintains 2 indices.

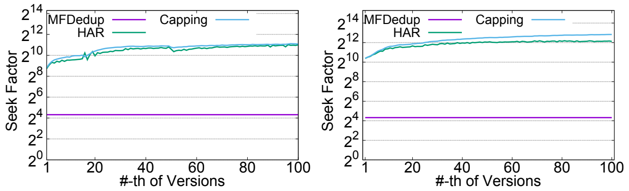
5.4 Restore Throughput

Previous approaches [15, 23] use **Speed Factor** to measure restore throughput. It is defined as the ratio of useful data restored per container read in deduplication-based backup systems, assuming using fix-sized containers as the read I/O unit [23]. Since MFDedup uses variable-sized containers to hold categories as the I/O unit, thus we define three metrics in this subsection, **Restore Throughput**, **Seek Number**, and **Read Amplification Factor**. Here Seek Number is defined as the number of seek operations required for reading containers/volumes on disk devices while Read Amplification Factor



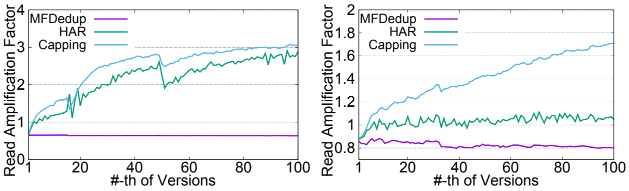
(a) WEB (b) CHM (c) VMS (d) SYN

Figure 9: Restore Throughput of MFDedup, HAR, and Capping on four backup datasets. *fread()* denotes sequential throughput of the backup device.



(a) WEB (b) VMS

Figure 10: Seek Number of MFDedup, HAR, and Capping on restoring two typical datasets (due to space limit).



(a) WEB (b) VMS

Figure 11: Read Amplification Factor of MFDedup, HAR, and Capping on restoring two datasets (due to space limit).

is defined in Section 2.2.

Figures 9, 10 and 11 present the restore results of MFDedup, HAR, and Capping on the three metrics, which demonstrate that Restore Throughput is generally consistent with the other two metrics. Figure 9 shows that HAR performs better than Capping in Restore Throughput, but MFDedup achieves up to 11.64× (WEB), 4.54× (CHM), 2.63× (VMS) and 3.73× (SYN) higher than HAR. This is because MFDedup has eliminated fragmentation by maintaining locality of backup workloads on the OPT data layout, while fragmentation (though alleviated) still exists in HAR and Capping based systems and becomes worse with higher versions.

Figure 10 shows the Seek Number on two datasets. Results for the other datasets were consistent and removed for space reasons. MFDedup reduces the Seek Number from thousands for HAR and Capping to 20, which is because it groups several archived categories into one big, sequentially written volume; Capping and HAR need more seek operations due to their scattered distribution of required chunks.

On the other hand, Figure 11 shows the Read Amplification Factor (results were consistent for all datasets). MFDedup has the smallest Read Amplification Factor, which is only 34.32% of Capping and 50.19% of HAR on average. This is because its OPT data layout has eliminated fragmentation. Meanwhile, HAR and Capping will encounter more unneeded chunks in loaded containers when restoring. Read Amplification Factor is less than 1 for MFDedup due to Internal deduplication within a backup version (Figure 2), so read chunks can be used multiple times for a restore. Therefore, restore throughput of MFDedup is even higher than the storage media: up to 1.5× of *fread()*, which means MFDedup can completely utilize the performance of storage devices.

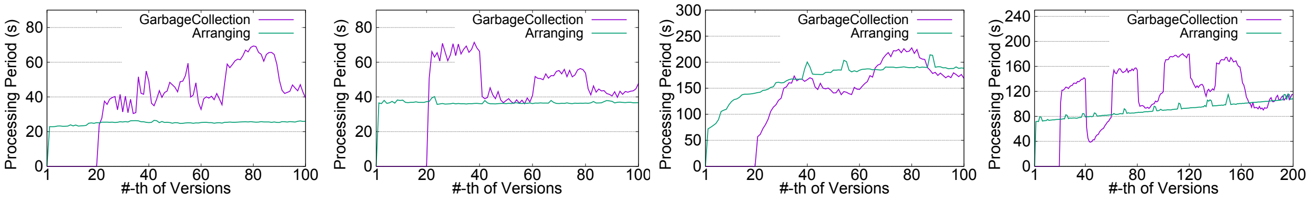
Note that these result are also evaluated while retaining 20 backup versions. If we retain more backup versions, the restore results of HAR and Capping will decrease, as is discussed in many previous works [15, 23]. Without fragmentation, MFDedup achieves a consistently high Restore Throughput, even when retaining more backup versions.

5.5 Arranging vs. Traditional GC

Compared with traditional deduplication approaches, MFDedup has the benefit of nearly zero-overhead Garbage Collection (GC), but adds the offline Arranging process. Therefore, in this subsection, we evaluate the time cost of Arranging in comparison with Perfect GC, which reflects the overhead for updating the OPT data layout in MFDedup.

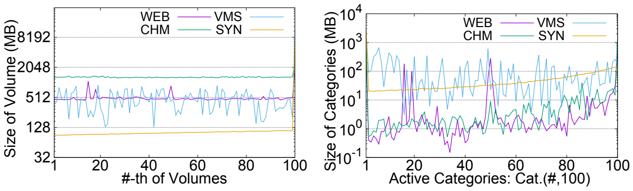
GC approaches mainly differ in the technique to select the containers and chunks to clean. Once selected though, all the GC techniques involve migrating referenced chunks into new, immutable containers. To simplify our evaluation, we conservatively focus on the cost of reading selected containers and migrating valid chunks into new containers since that is the common phase. This is a lower bound on the cost of GC since it neglects the selection phase, which involves enumerating the live files/chunks [11].

The results are shown in Figure 12 comparing Arranging and Perfect GC. Since we are retaining 20 versions, GC does not run for the first 20 versions, though Arranging does. Analyzing the steady-state performance after the 20th version, Arranging's total processing period is only 45% (WEB), 37% (CHM) and 25% (SYN) of GC's total processing time on average. But in VMS, Arranging takes 9% longer than GC because VMS's modification style (always change the same region in each backup) makes GC very easy. Generally, Figure 12 suggests that Arranging is usually faster than GC, which would take even more time if the selection phase were included in GC's total. When MFDedup runs its version of GC,



(a) WEB (b) CHM (c) VMS (d) SYN

Figure 12: Time cost comparison between Arranging of MFDedup and GC of traditional deduplication systems.



(a) Volume (b) Active Category

Figure 13: Size distribution of Volumes and Categories after deduplicating 100 backup versions with MFDedup.

the processing time is insignificant since large Volumes can be deleted at once without any copy-forward.

Arranging has a consistent processing time across versions, while GC's runtime is more variable, and consistent overheads are easier to plan for in a storage system. Arranging's processing time is consistent because it is a local process on a recent version, while GC is a global process. As Figure 6 shows, Arranging is always applied in Active categories generated in the same version, and it always achieves a better locality. On the other hand, GC in other techniques suffers from poor locality [17], because the selected containers and chunks are distributed randomly.

Note that other GC approaches will be faster than Perfect GC, but at the cost of greatly decreasing Actual Deduplication Ratio as discussed in Section 5.2. In contrast, MFDedup has almost no deduplication ratio loss for GC while supporting immediate deletion and GC, and also achieving nearly perfect restore performance with an acceptable Arranging cost, as shown in Figures 7, 9, and 12.

5.6 Size Distribution of Volumes/Categories

In this section, we demonstrate the data layout of MFDedup with the size of volumes and active categories. We back up 100 versions without retention for evaluation. After that, there will be 99 Volumes and 100 active categories, and these active categories compose a logical volume (they will be archived in a volume after the next Arranging).

Figure 13(a) shows the size of Volumes varying from 90MB to 1.3GB on our four datasets. The results can tell administrators how much space will be freed by MFDedup by deleting backup versions (volumes). The results also help administrators estimate how much more can be written to the deduplication system, since volumes represent the difference between neighboring versions. For previous deduplication systems, it is difficult to answer these two issues [35], though sketching approaches have been considered [19].

Figure 13(b) shows the size of active Categories vary in a large range. We learn that the maximum categories hold about 16.99% (WEB), 46.46% (CHM), 18.49% (VMS), 51.87% (SYN) of the size of the last backup version. This indicates the reserved-space requirement for offline Arranging in MFDedup is much smaller than a full backup as discussed in Section 4.5. Meanwhile, the reserved space can be further reduced by compressing categories.

6 Conclusion and Future Work

In this paper, we propose a management-friendly deduplication framework, MFDedup. Different from traditional 'Write Friendly' style deduplication architectures, MFDedup, is designed to be 'Management-Friendly' and solves the fragmentation problem in deduplication-based backup systems, by introducing a novel deduplication process (NDF) and a locality improvement process (AVAR) to generate OPT data layout and thus maintain locality of backup workloads.

With the benefits of eliminating the fragmentation problem, MFDedup improves actual deduplication ratios (1.12× to 2.19× higher) and restore throughput (2.63× to 11.64× higher) than previous approaches with accepted time cost on the offline 'Arranging' to update the OPT data layout, while GC in MFDedup is nearly zero-overhead.

As future work, we are considering adding delta compression in MFDedup for further space savings as well as handling more complex backup scenarios such as incremental backups.

Acknowledgments

We are grateful to our shepherd Danny Harnik and the anonymous reviewers for their insightful comments. This research was partly supported by National Key R&D Program of China under Grant no. 2018YFB1003800, 2018YFB1003805, the National Natural Science Foundation of China under Grant no. 61972441, no. 61972112, and no. 61832004, the Shenzhen Science and Technology Program under Grant no. JCYJ20190806143405318, no. JCYJ20200109113427092, and no. JCYJ20170413105929681, Innovation Fund of WNLO 2018WNLOKF008.

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

References

[1] Intel intelligent storage acceleration library crypto version. https://github.com/intel/isa-l_crypto. [Online].

[2] Yamini Allu, Fred Douglis, Mahesh Kamat, Ramya Prabhakar, Philip Shilane, and Rahul Ugale. Can't we all get along? redesigning protection storage for modern workloads. In *Proceedings of the 2018 USENIX Conference on USENIX Annual Technical Conference (ATC' 18)*, pages 705–718, Boston, MA, July 2018. USENIX Association.

[3] George Amvrosiadis and Medha Bhadkamkar. Identifying trends in enterprise data protection systems. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15)*, page 151–164, USA, September 2015. USENIX Association.

[4] George Amvrosiadis and Medha Bhadkamkar. Getting back up: Understanding how enterprise data backups fail. In *Proceedings of the 2016 USENIX Conference on USENIX Annual Technical Conference (ATC' 16)*, pages 479–492, Denver, CO, June 2016. USENIX Association.

[5] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–14, Haifa, Israel, October 2009. Association for Computing Machinery.

[6] Tony Asaro and Heidi Biggar. Data de-duplication and disk-to-disk backup systems: Technical and business considerations. *The Enterprise Strategy Group*, pages 2–15, 2007.

[7] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 1–9. IEEE, 2009.

[8] William J Bolosky, Scott Corbin, David Goebel, and John R Douceur. Single instance storage in windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24, Seattle, WA, August 2000. USENIX Association.

[9] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David H. C. Du. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST' 19)*, pages 129–142, Boston, MA, February 2019. USENIX Association.

[10] Zhichao Cao, Hao Wen, Fenggang Wu, and David H. C. Du. Alacc: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST' 18)*, pages 309–324, Oakland, CA, USA, February 2018. USENIX Association.

[11] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. The logic of physical garbage collection in deduplicating storage. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST' 17)*, Santa Clara, CA, USA, February 2017. USENIX Association.

[12] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data deduplication—large scale study and system design. In *Presented as part of the 2012 USENIX Annual Technical Conference (ATC'12)*, pages 285–296, Boston, MA, October 2012. USENIX Association.

[13] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30(2005), 2005.

[14] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):855–868, 2015.

[15] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*, page 181–192, Philadelphia, PA, October 2014. USENIX Association.

[16] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, page 331–344, Santa Clara, CA, February 2015. USENIX Association.

[17] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (ATC' 11)*, pages 1–25, Portland, OR, October 2011. USENIX Association.

[18] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (ATC' 17)*, pages 759–771, Santa Clara, CA, July 2017. USENIX Association.

[19] Danny Harnik, Moshik Hershcovitch, Yosef Shatsky, Amir Epstein, and Ronen Kat. Sketching volume capacities in deduplicated storage. *ACM Transactions on Storage*, 15(4), December 2019.

[20] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mos: A reusable networking stack for flow monitoring middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 113–129, Boston, MA, March 2017.

[21] Keren Jin and Ethan L Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, Haifa, Israel, oct 2009. Association for Computing Machinery.

[22] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, Haifa, Israel, October 2012. Association for Computing Machinery.

[23] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, page 183–198, San Jose, CA, February 2013. USENIX Association.

[24] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th Conference on File and Storage Technologies (FAST' 09)*, volume 9, pages 111–123, San Francisco, California, February 2009. USENIX Association.

[25] Bo Mao, Hong Jiang, Suzhen Wu, Yinjin Fu, and Lei Tian. Read-performance optimization for deduplication-based storage systems in the cloud. *ACM Trans. Storage*, 10(2), March 2014.

[26] Dirk Meister, Jürgen Kaiser, and André Brinkmann. Block locality caching for data deduplication. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*, Haifa, Israel, October 2013. Association for Computing Machinery.

[27] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (ToS)*, 7(4):1–20, 2012.

[28] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2011.

[29] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP' 01)*, pages 174–187, Banff, Alberta, Canada, December 2001. Association for Computing Machinery.

[30] Young Jin Nam, Dongchul Park, and David H.C. Du. Assuring demanded read performance of data deduplication storage with backup datasets. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '12)*, pages 201–208, USA, July 2012. IEEE Computer Society.

[31] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David HC Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications (HPCC' 11)*, pages 581–586, USA, July 2011. IEEE, IEEE Computer Society.

[32] Fan Ni and Song Jiang. Rapidcdc: Leveraging duplicate locality to accelerate chunking in cdc-based deduplication systems. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC' 19)*, pages 220–232, Santa Cruz, CA, USA, November 2019. Association for Computing Machinery.

[33] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC'04)*, pages 73–86, Boston, MA, October 2004. USENIX Association.

[34] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02*, volume 2, pages 89–101, Monterey, CA, February 2002. USENIX Association.

[35] Philip Shilane, Ravi Chitloor, and Uday Kiran Jonnala. 99 deduplication problems. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'16)*, page 86–90, Denver, CO, USA, June 2016. USENIX Association.

[36] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC'12)*, pages 261–272, Boston, MA, 2012. USENIX Association.

[37] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage (TOS)*, 5(4):1–28, 2009.

[38] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, volume 12, pages 4–4, San Jose, CA, February 2012. USENIX Association.

[39] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[40] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (ATC' 11)*, pages 26–30, Portland, OR, October 2011. USENIX Association.

[41] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Yucheng Zhang, and Qing Liu. Fastcdc: A fast and efficient content-defined chunking approach for data deduplication. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC' 16)*, page 101–114, Denver, CO, USA, October 2016. USENIX Association.

[42] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. Ae: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1337–1345. IEEE, August 2015.

[43] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *Proceedings of 17th USENIX Conference on File and Storage Technologies (FAST '19)*, pages 121–128, Santa Clara, CA, USA, February 2019. USENIX Association.

[44] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R. Butt. Duphunter: Flexible high-performance deduplication for docker registries. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 769–783. USENIX Association, July 2020.

[45] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST' 08)*, volume 8, pages 1–14, San Jose, California, February 2008. USENIX Association.

Remap-SSD: Safely and Efficiently Exploiting SSD Address Remapping to Eliminate Duplicate Writes

You Zhou[†], Qiulin Wu[†], Fei Wu[†][*], Hong Jiang[‡], Jian Zhou[†], and Changsheng Xie[†]
[†]*Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology*
[‡]*Department of Computer Science and Engineering, University of Texas at Arlington*

Abstract

Duplicate writes are prevalent in diverse storage systems, originating from data duplication, journaling, and data relocations, etc. As flash-based SSDs have been widely deployed, these writes can significantly degrade their performance and lifetime. To eliminate duplicate writes, prior studies have proposed innovative approaches that exploit the *address remapping* utility inside SSDs. However, remap operations lead to a mapping inconsistency problem, which may cause data loss and has *not* been properly addressed in existing studies.

In this paper, we propose a novel SSD design, called *Remap-SSD*, with two notable features. First, it provides a *remap* primitive, which allows the host software and SSD firmware to perform logical writes of duplicate data at almost zero cost. Second, a *hybrid* storage architecture is employed to maintain the mapping consistency. Small byte-addressable non-volatile RAM is used to persist remapping metadata in a log-structured manner and is managed synergistically with flash memory. We verify Remap-SSD on a software SSD emulator with three case studies: intra-SSD deduplication, SQLite journaling, and F2FS cleaning. Experimental results show that Remap-SSD can realize the full potential of address remapping to improve SSD performance and lifetime.

1 Introduction

Duplicate writes are pervasive in real-world storage systems. Not only data duplication is common [16, 51, 62, 64], but also a broad spectrum of system software and applications introduce duplicate writes. For example, many databases and file systems employ double-write journaling to guarantee write atomicity [24, 46, 55]; data relocations are required for space cleaning in log-structured/copy-on-write systems [35, 46] and for file defragmentation [23]; file copy and snapshotting operations are common behaviors [60, 66].

On the other hand, NAND flash-based *solid state drives (SSDs)* have been widely employed in various storage systems. Due to the idiosyncrasies of flash memory, the SSD-internal firmware, called *flash translation layer (FTL)*, performs out-of-place updates. Logical pages written from the host are always mapped to new free flash pages, while obsolete flash pages are invalidated. Thus, a *logical-to-physical (L2P)* mapping table is maintained to translate *logical page numbers (LPNs)* to *physical page numbers (PPNs)* [21, 42]. For fast lookups, this table is typically cached in SSD-internal DRAM. The FTL also conducts *garbage collection (GC)* periodically to reclaim invalid pages in the granularity of flash blocks, where valid pages are relocated and then the blocks are erased. Notice that writes are harmful to both the performance and lifetime of SSDs [14, 43]. This situation deteriorates, as flash technologies are scaling rapidly to increase the bit density but at the cost of degraded write speed and endurance [33].

To eliminate duplicate writes on flash memory, innovative approaches have been proposed to exploit the SSD *address remapping* functionality [16, 17, 22–24, 28, 34, 45, 46, 60]. By directly modifying the L2P mapping table, copies and moves of data pages as well as duplicate writes of repeating data pages can be completed quickly without conducting physical writes. Also, data transfers between the host and SSD can be avoided. Although enabling such remapping requires minor modifications to the host software and SSD interface, the benefits are quite worthwhile. The performance, lifetime, and space utilization of SSDs can be improved significantly.

However, remap operations lead to a critical *mapping inconsistency* problem, which may cause data corruption. Whenever a logical data page is written to a flash page, the FTL needs to store some *house-keeping metadata* including the relevant LPN either in the *out-of-band (OOB)* area of the same flash page [21, 41] or in another reserved flash page [8]. These persistent *physical-to-logical (P2L)* mappings are indispensable for completing data relocations during each GC operation and for recovering L2P mappings after sudden power failures (see Section 2). Remap operations change the L2P mappings, but the relevant P2L mappings on flash memory cannot be updated accordingly. Due to such mapping inconsistency, wrong L2P mappings would be modified after data relocations during GC or be restored during power-off recovery,

[*]Corresponding author. Email: wufei@hust.edu.cn.

compromising data consistency.

This mapping inconsistency problem, although crucial, has *not* been properly addressed in prior studies. The common solution in [16, 22, 23, 34, 45, 46] is to persist new P2L mappings generated by remap operations in a dedicated log on flash memory. Its main drawback is that the log size would increase continuously over time, incurring prohibitively high lookup overheads at last. Although limiting the log size could confine the lookup overheads, it would also restrict the usage of SSD address remapping. In addition, some other solutions have been proposed but only fit in very limited application scenarios of address remapping [24, 28]. These solutions and their drawbacks are discussed thoroughly in Section 3.3.

In this paper, we propose a novel SSD design, called *Remap-SSD*, to safely and efficiently exploit SSD address remapping for reducing duplicate writes. Its two notable features are: (1) providing a *remap* primitive, which allows the host software and SSD firmware to conduct logical writes of duplicate data at almost zero cost; and (2) employing a *hybrid* storage architecture, where small byte-addressable *non-volatile RAM (NVRAM)* is employed to store remapping metadata in a log-structured manner and is managed synergistically with flash storage. Remap-SSD not only ensures that persistent P2L mappings are always consistent with the latest L2P mappings, but also enables fast lookups of P2L mappings during GC. We verify Remap-SSD on FEMU (a software SSD emulator [38]) with three case studies: intra-SSD deduplication, SQLite journaling, and F2FS cleaning. Experimental results show that Remap-SSD can realize the full potential of address remapping for improving SSD performance and lifetime.

2 Background

Mappings in flash-based SSDs: Modern SSDs generally employ a page-level FTL, powered by embedded processors and DRAM, for high performance [20, 21]. Since a host logical page can be dynamically mapped to any flash page, an *L2P mapping* table is maintained for address translation. Assuming the page size is 4KB and each mapping entry takes 4B, the table size is about 0.1% of the SSD capacity. The table is persisted on flash memory and usually cached in DRAM for fast lookups, which locate on the critical path of I/O processing.

When a logical page is written to a flash page, the FTL transparently persists the reverse *P2L mapping* (i.e., the LPN) and *write timestamp* as house-keeping metadata on flash memory for two reasons. First, data pages are periodically migrated on flash memory for GC and wear leveling purposes. P2L mappings need to be retrieved to locate and modify the relevant L2P mappings after the migrations. Second, the mapping consistency needs to be guaranteed. The latest L2P mappings in DRAM may get lost after sudden power failures [42]. By scanning the persistent metadata, the FTL can obtain all the PPN-LPN entries and write order of PPNs, from which the latest L2P mappings can be restored.

Flash management: SSDs are architected with a number of channels connecting many flash dies, each of which is a parallel unit for accesses [30]. It has been a common practice, especially for high-performance SSDs, to organize flash storage in *superblocks* [8, 14, 20, 54, 58]. A superblock consists of flash blocks with the same offset across multiple dies. Both space allocations for data writes and GC operations are performed in the unit of a superblock. This has several advantages. First, the intra-SSD parallelism can be maximized. Second, flash management is simplified due to a large granularity. Third, it facilitates die-level RAID, as parity can be easily added in each superblock [14, 33, 67]. Finally, the FTL can accelerate the recovery speed of L2P mappings by storing house-keeping metadata of each superblock collectively in its tail flash pages [8]. Then, only a small amount of tail flash pages need to be scanned, rather than all the flash pages.

Non-volatile RAM: NVRAM technologies (e.g., PCRAM and MRAM) have received much attention and their developments are advancing [47]. Compared to flash technologies, they offer attractive benefits, such as lower latency and byte-addressability, but have lower bit density and higher cost. Therefore, NVRAM complements flash memory well and has opened up new opportunities to enhance flash-based SSDs for various purposes [26, 28, 40, 44]. Notably, SSDs with hybrid storage architectures have entered the market since 2019 (e.g., Intel Optane memory H10 with Optane memory and QLC flash [7]) and will gain increased popularity in the near future.

3 Motivation

SSDs have been deployed in diverse storage systems [18, 19], where duplicate writes are prevalent. We illustrate this with several examples in Section 3.1. Although duplicate writes degrade the performance, lifetime, and space utilization of SSDs, they can be eliminated by exploiting SSD address remapping. We detail where and how prior studies leverage SSD address remapping in Section 3.2 and their drawbacks in ensuring mapping consistency in Section 3.3.

3.1 Duplicate Writes

Data Duplication. One major source of duplicate writes is *data duplication*, which is commonplace [10, 39, 51, 62, 64]. For instance, in the disk images of some departmental working environments [16] and file system images collected from smartphones [64], the data duplication rate is 8%~86% and an average of 33%, respectively, while duplicate writes account for 6%~28% and 22%~48% of total writes; in the three production systems at FIU, the ratios of duplicate writes range from 33% to 92% [22].

Journaling. To guarantee write atomicity, journaling approaches have been widely used in databases (e.g., MySQL and SQLite) and file systems (e.g., ext4 and XFS) [24, 46]. Either before-images (e.g., *rollback journaling*) or after-images (e.g., *write-ahead logging*) of updated pages are

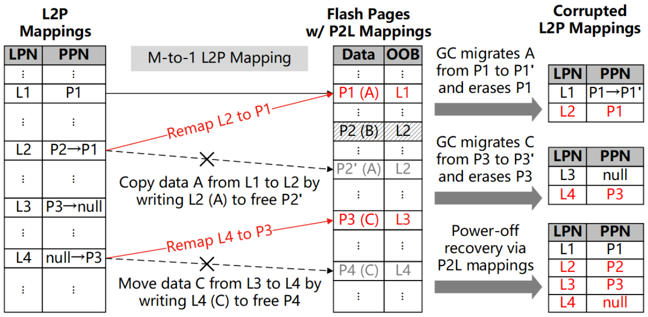

L2P Mappings
LPN
PPN
L1
P1
L2
P2→P1
L3
P3→null
L4
null→P3
M-to-1 L2P Mapping
Remap L2 to P1
Copy data A from L1 to L2 by writing L2 (A) to free P2'
Remap L4 to P3
Move data C from L3 to L4 by writing L4 (C) to free P4
Flash Pages w/ P2L Mappings
Data
OOB
P1 (A)
L1
P2 (B)
L2
P2' (A)
L2
P3 (C)
L3
P4 (C)
L4
GC migrates A from P1 to P1' and erases P1
GC migrates C from P3 to P3' and erases P3
Power-off recovery via P2L mappings
Corrupted L2P Mappings
LPN
PPN
L1
P1→P1'
L2
P1
L3
null
L4
P3
L1
P1
L2
P2
L3
P3
L4
null

Figure 1: **Examples of SSD remap operations.** Duplicate writes to LPNs *L2* and *L4* can be completed through address remapping without writing flash pages. However, L2P and P2L mappings become inconsistent, causing data corruption.

written in a dedicated log, after which updates are applied to home/original locations in place. Such journaling introduces double writes of data, for example, causing a worst-case slowdown of about 73% in ext4 compared to no journaling [55].

Data Relocation. Copy-on-write and log-structuring mechanisms are popular means to provide write atomicity and write sequentiality (e.g., in Couchbase and F2FS) [35, 46]. They conduct out-of-place updates, so periodical *cleaning* or *compaction* operations are required to reclaim storage space occupied by stale data. In addition, file fragmentation has been a long-standing problem that degrades the performance of file systems. Many file systems recommend periodical *defragmentation* [23]. Both cleaning/compaction and defragmentation cause data relocations and thus duplicate writes.

Data Copy and Snapshot. Data copy is a frequent behavior of users and applications. *Snapshotting*, which provides point-in-time states of data volumes, is an important feature and a common routine in storage systems [56]. These operations may introduce duplicate writes to create physical data copies.

3.2 Exploiting SSD Address Remapping

To eliminate duplicate writes, the SSD address remapping functionality can be utilized. Assume LPN *Ly* is written with a duplicate data page copied or moved from LPN *Lx*. The FTL can realize the write by remapping *Ly* to the flash page storing *Lx*, rather than by writing a new free flash page. Such remap operations, as shown in Figure 1, can be done quickly by updating the relevant L2P mappings in SSD-internal DRAM.

Many prior studies have proposed to exploit SSD address remapping in a spectrum of application scenarios, as summarized in Figure 2 and Table 1. Among the studies, a body of works integrate a data deduplication engine inside SSDs [16, 22, 34, 50, 63, 65].[1] The engine identifies duplicate data pages written from the host (through hashing fingerprints). Instead of writing them to flash memory, they can be remapped to existing flash pages that store the same contents. Address remapping is also attractive for reducing journaling overheads [17, 24, 45, 46, 60]. After data pages to be updated are written to the log, they can be applied by remapping LPNs of their original locations to the relevant flash pages storing the log. Using remapping for snapshotting files [60] is straightforward, like copying data *A* in Figure 1. Data relocations for cleaning [28], compaction [46], and defragmentation [23] can be accomplished similarly to moving data *C* in Figure 1.

However, address remapping causes a critical *mapping inconsistency problem*. Remap operations modify the L2P mappings, but the relevant P2L mappings on flash memory cannot be updated accordingly (because flash memory does *not* support in-place updates). Such inconsistency between L2P and P2L mappings would finally cause data corruption, since L2P mappings would be altered incorrectly during GC or be rebuilt falsely during power-off recovery. For example, in Figure 1, after remapping LPN *L2* (previously mapped to PPN *P2*) to PPN *P1* (already referenced by *L1*), the L2P and P2L mappings of *P1* become inconsistent ($\{L1, L2\} \rightarrow P1$ vs. $P1 \rightarrow L1$). Then, after a GC operation migrates the data page on PPN *P1* to $P1'$ and erases *P1*, *L2* would still be mapped to *P1* wrongly. Consider another scenario where L2P mappings need to be restored after a sudden power outage. An improper L2P mapping, i.e., $L2 \rightarrow P2$, would be recovered from the P2L mapping, i.e., $P2 \rightarrow L2$, persisted on flash memory.

Although several schemes have been proposed in existing studies to cope with the mapping inconsistency, they suffer from severe drawbacks. To facilitate in-depth analysis of the drawbacks in Section 3.3, we classify the applications of remapping in two dimensions. Note that remap operations change the L2P mapping regularity from conventional 1-to-1 to *M-to-1*. In the first dimension, a remapping scenario is considered as *P*-type, if the maximum *M*, namely *degree of L2P association*, is predefined. Otherwise, it is *U*-type. For example, data relocation and journaling are P-type (*M* equals to 1 and 2, respectively), while deduplication and file copy are U-type (*M* depends on content popularity and user behaviors, respectively). In the second dimension, a remapping scenario is *D*-type, if the LPNs and PPNs for future remapping are deterministic at the time of the PPNs being written. Otherwise, it is *N*-type. For instance, in write-ahead logging (*D*-type), when data pages being updated are written to the log, the LPNs of their original locations are already known.

Combining the two dimensions (P/U-type and D/N-type), applications of SSD address remapping are divided into three types (*PD*, *PN*, and *UN*), as shown in Figure 2. The *UD* type is not applicable because the *U* type and *D* type contradict with each other.

[1] Intra-SSD deduplication presents a drop-in solution that is highly desirable for two reasons. First, the detrimental effects of writes on SSDs can be substantially alleviated without modifying the host software and consuming host computing and memory resources. Second, data deduplication can be implemented efficiently by utilizing the FTL's functionalities (e.g., address remapping and GC) [16]. Also, a hardware hash unit can be employed [22].

Table 1: Prior studies exploiting SSD address remapping.

Name	Applications of remapping	Schemes for mapping consistency guarantee	Major drawbacks
JFTL [17]	Write-ahead logging (WAL)	None	N/A
ANViL [60]	Snapshots, data deduplication, WAL		
CAFTL [16], CA-SSD [22]	Intra-SSD data deduplication	Maintain a dedicated log on flash memory to record P2L mappings changed by address remapping	High lookup overheads of P2L mappings during GC, poor scalability
Janusd [23]	File system defragmentation		
Copyless copy [45]	WAL, intra-SSD data deduplication		
SHARE [46]	WAL, compaction, tree wandering in copy-on-write databases		
PebbleSSD [28]	Cleaning in log-structured file systems	Replace (fixed-size) flash OOB with byte-addressable NVRAM	Only apply in *P*-type remapping scenarios
WAL-SSD [24]	WAL	Write the predetermined LPN for future remapping to flash OOB	Only apply in *PD*-type remapping scenarios

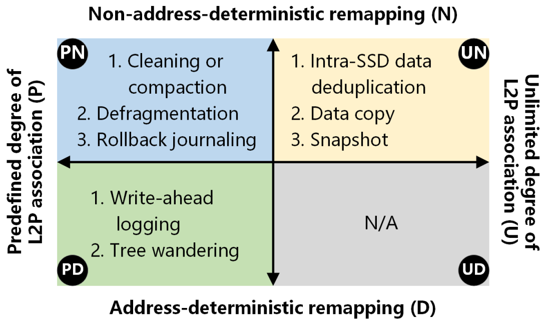

Non-address-deterministic remapping (N)
PN
1. Cleaning or compaction
2. Defragmentation
3. Rollback journaling
UN
1. Intra-SSD data deduplication
2. Data copy
3. Snapshot
Predefined degree of L2P association (P)
Unlimited degree of L2P association (U)
1. Write-ahead logging
2. Tree wandering
N/A
PD
UD
Address-deterministic remapping (D)

Figure 2: **Applications of SSD address remapping.** They can be classified according to characteristics of remapping.

3.3 Schemes for Mapping Consistency

To address the mapping inconsistency problem caused by remapping, several schemes have been proposed, as listed in Table 1. Taking all types of remapping scenarios into consideration, the common scheme adopted in [16,22,23,45,46] is to maintain a dedicated log on flash memory for persisting the P2L mappings changed by remapping. This scheme is referred to as *Remap-SSD-FLog* in Section 5. Its major drawback is that it requires scanning the entire log to retrieve certain P2L mappings during every GC operation and power-off recovery. Especially, the log size increases continuously and could grow very large as remap operations are used. Assume the SSD capacity is 4TB, page size is 4KB, and each log entry for a page remap operation takes at least 12B (e.g., 4B PPN + 4B LPN + 4B timestamp). When 5% or 20% of data pages have been remapped (these ratios are quite reasonable, considering the popularity of duplicate writes discussed in Section 3.1), the log size is as large as 600MB or 2.4GB, respectively. Hence, the lookup overheads of P2L mappings would increase over time and finally become exceedingly high. It would *not* be an effective solution to add high-speed NVRAM for storing the log (denoted as *Remap-SSD-NLog* in Section 5). This is because the scanning process would still be very time-consuming, e.g., from tens of milliseconds to seconds when the log size is hundreds of megabytes.

To confine the lookup overheads, Janusd [23] sets a limit on the log size and reclaims obsolete mapping entries periodically. However, remap operations have to be disabled when the number of valid entries reaches the limit. Additionally, high reclamation overheads are introduced, i.e., reading and re-writing the entire log on flash memory.

PebbleSSD [28] proposes an NVRAM-enhanced scheme, which replaces the fixed-size OOB area in flash pages with byte-addressable NVRAM. Therefore, P2L mappings of remapped data pages can be updated in place in the NVRAM OOB, retaining consistent with the L2P mappings. However, due to the limited OOB size, this scheme only fits in *P*-type remapping scenarios, where the maximum degree of L2P association is limited and small. For *UN*-type remapping, where the degree of L2P association may be high, large NVRAM OOB area would be required. This would greatly increase the cost. Moreover, NVRAM space utilization would be low, since not all flash pages have high degrees of L2P association.

By utilizing the property of *PD*-type remapping, WAL-SSD [24] writes the predetermined LPN for future remapping to the OOB area when the relevant flash page is written. Thus, the L2P and P2L mappings of the flash page are consistent after the predefined remap operation. This scheme is only applicable for *PD*-type remapping scenarios, because the LPNs for future remapping are totally uncertain in *N*-type scenarios.

In summary, existing SSD designs that exploit address remapping restrict the application scenarios and/or usage frequency of remapping severely, mainly due to the L2P and P2L mapping inconsistency problem. Furthermore, simply enhancing the SSD with extra NVRAM is inadequate to remove the restrictions. As a consequence, the full potential of SSD address remapping is largely underutilized.

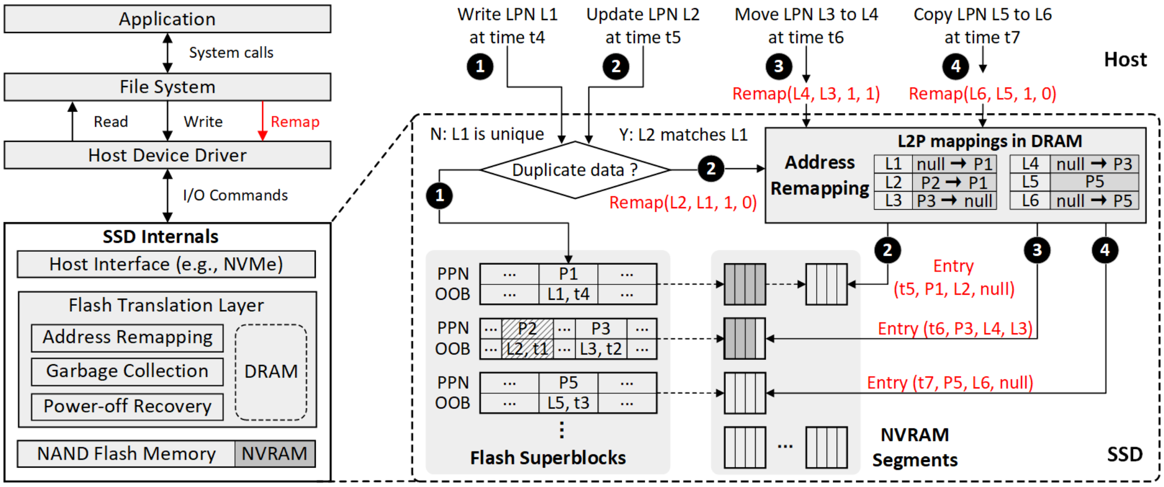


Figure 3: **Overview of Remap-SSD.** The SSD supports a `remap` primitive, which can be invoked by host software (❸❹) or the FTL internally (e.g., by an intra-SSD deduplication engine ❷). To guarantee the L2P and P2L mapping consistency, remapping metadata entries are persisted in NVRAM segments that are exclusively allocated to each flash superblock on demand.

4 Design

In this section, we present a novel SSD design, called Remap-SSD. The goal is to maximize the utilization of address remapping in diverse application scenarios and meanwhile maintain the L2P and P2L mapping consistency efficiently.

4.1 Overview of Remap-SSD

Remap-SSD provides a *remap* primitive at the firmware/FTL level, which embodies the address remapping utility, as shown in Figure 3. The primitive is exposed to the host software as a vendor specific command, which is supported inherently in current interface techniques (e.g., NVMe and SATA). Through the primitive, applications and file systems can copy or relocate data pages without performing flash writes. Furthermore, the primitive can be used internally by the FTL, e.g., to eliminate writes of duplicate data when an intra-SSD deduplication engine is employed.

The remap primitive is formatted as `remap(tgtLPN, srcLPN, length, remapFlag)` (*tgt*: target, *src*: source). It remaps a range of LPNs between `tgtLPN` and `tgtLPN + length - 1` to the flash pages currently mapped to the range of LPNs between `srcLPN` and `srcLPN + length - 1`. The `remapFlag` parameter is a 1-bit flag indicating whether the source LPNs should be deallocated/invalidated or not after remapping. For data relocations, the corresponding flash pages should no longer be mapped to source LPNs (`remapFlag = 1`). Regarding data copies, the L2P mappings of source LPNs are retained (`remapFlag = 0`) and the degrees of L2P association of relevant flash pages increase by one. Both remapping and invalidation of LPNs are realized by directly modifying the L2P mapping table in SSD-internal DRAM.

Another notable feature of Remap-SSD is a hybrid storage architecture consisting of flash memory and byte-addressable NVRAM. Flash memory is organized in superblocks for data storage. Each superblock consists of flash blocks with the same offset across all flash dies. Besides P2L mappings and write timestamps that are persisted on flash memory along with data pages, Remap-SSD stores additional house-keeping metadata on NVRAM for address remapping, called *remapping metadata (RMM)*. Whenever an LPN is remapped to a flash page, a RMM entry that includes the changed P2L mapping is written to NVRAM. A remap command is considered to be completed successfully only after the involved L2P mappings have been modified in DRAM and the relevant RMM entries have been persisted on NVRAM. Modifications of L2P mappings are *not* required to be persisted because they can be recovered from house-keeping metadata (see Section 4.5). Thus, remap operations can be carried out quickly.

We introduce how to manage RMM entries on NVRAM in Section 4.2, which is the key to solve the problems of high-overhead lookups and poor scalability in the exiting solution (Remap-SSD-FLog). Details of RMM, which guarantee the mapping consistency and remapping atomicity, are described in Section 4.3. Sections 4.4 and 4.5 present how Remap-SSD performs GC operations and power-off recovery, respectively.

4.2 Co-management of Flash and NVRAM

Naively logging RMM entries would result in an expensive scan of the log for every lookup of P2L mappings, as analyzed in Section 3.3. To address this challenge, Remap-SSD takes advantage of a key observation that a flash superblock is the basic unit of free space allocations (for data writes) and GC operations. This observation delivers a favorable conclusion that *retrievals of P2L mappings are always performed in the granularity of a flash superblock.*

P2L mappings are retrieved during GC and power-off recovery. In each GC operation, the FTL selects a victim flash superblock, where valid data pages are read out and written to a free flash superblock. Before the migrations, valid P2L mappings of the victim superblock need to be retrieved so that the involved L2P mappings can be updated to point to new physical locations. After the migrations, the victim superblock can be erased and become free. The main process of power-off recovery is rebuilding the latest L2P mapping table based on house-keeping metadata of data pages that have been persisted on flash memory. This process starts with scanning the house-keeping metadata in write time order. Since data pages and their house-keeping metadata are written to flash memory superblock by superblock, P2L mappings of data pages in a superblock are examined together.

Based on the conclusion, Remap-SSD manages flash memory and NVRAM synergistically. The NVRAM volume is divided into fixed-size *segments*, which are exclusively allocated to a flash superblock *on demand* to store its RMM entries. A *segment validity bitmap (SV-bitmap)* is maintained in DRAM or NVRAM to indicate whether each segment is used or free. Each segment is partitioned into slots, which are written with RMM entries in a *log-structured* manner. When any data page in a flash superblock is remapped, the relevant RMM entry is appended in the free NVRAM segment allocated to the superblock (e.g., ❸ in Figure 3). If the superblock has no segments yet (e.g., ❹ in Figure 3) or the segment in use is full (e.g., ❷ in Figure 3), a new free segment is assigned first. We refer to the NVRAM segments that belong to a flash superblock as a *segment group*. A group contains zero or an unfixed number of segments, which are linked together.

An NVRAM segment group is actually a small and size-varied local log of remapping metadata for a flash superblock.[2] Compared with scanning a single global log for retrieving P2L mappings during GC in prior studies (i.e., Remap-SSD-FLog), Remap-SSD achieves fast lookups by scanning only a segment group. Meanwhile, Remap-SSD is adaptive to workloads and has high NVRAM utilization.

4.3 Remapping Metadata

Contents of RMM entries should be carefully designed to serve three goals: mapping consistency, atomicity of remap operations, and space efficiency.

First, the changed P2L mapping and timestamp of an LPN remapping should be recorded for power-off recovery of L2P mappings. Recall that a remap operation is to remap a target LPN to the PPN that is currently mapped to a source LPN; if it is a relocation-based remapping (`remapFlag=1`), the source LPN needs to be deallocated. The P2L mapping contains four fields: a pair of *target LPN and PPN*, a *remapping flag*, and an *alterable field*, i.e., a *source LPN* if the flag is set or *null* value otherwise. Without the last two fields, deallocations of source LPNs could not be recognized and then L2P mappings of source LPNs may be revived undesirably after power-off recovery.[3] The *timestamp* can be virtual time. In the current implementation, we use the number of host write/remap operations that have been performed in the SSD, i.e., *write/remap sequence number* for short.

Second, atomicity of remap operations should be maintained, as their executions may be disrupted by sudden power outages. We distinguish two atomicity levels: *remapping atomicity* and *command atomicity*. The former refers to the atomicity of remapping a single LPN, or more precisely, write atomicity of a RMM entry on NVRAM. A partially updated or written RMM entry would result in improper power-off recovery of L2P and P2L mappings and thus data corruption. A remap command includes one or multiple RMM entries that may scatter in several NVRAM segments. Command atomicity implies atomic remap commands. If the write of any RMM entry in a remap command fails, all the mapping changes caused by the command should be discarded.

Partial updates of RMM entries have been avoided by the log structure of NVRAM segments. Remap-SSD must be able to further detect incomplete writes of RMM entries on NVRAM for remapping atomicity, and moreover, recognize whether all the RMM entries of a remap command have been persisted successfully for command atomicity. This can be achieved by adding extra fields in each RMM entry.

Modern processors generally support 8-byte atomic writes to NVRAM [68]. Remap-SSD configures RMM entry size to be a multiple of 8 bytes, say $K * 8$ bytes. As K is larger than one, Remap-SSD adopts a simple *tornbit* mechanism implemented by Mnemosyne [59] to guarantee atomic writes of RMM entries. In every 8 bytes, a single torn bit is preserved. NVRAM segments are initialized to zeros when allocated for use. Completely written entries will have all K torn bits set as ones, while incomplete entries, which have at least one zero torn bit, will be discarded during power-off recovery.

If command atomicity is desired, three more fields are required in a RMM entry: the *start LPN* and *length* of the remap command, a *command atomicity flag* indicating whether the remap command is required to be atomic. Each remap command can be identified by its write/remap sequence number. When RMM entries on NVRAM are scanned during power-off recovery, a remap command is successfully executed only if all the RMM entries in its LPN range are found to be intact. Otherwise, the remap command is partially performed and will be abandoned to guarantee command atomicity.

Current applications commonly require remapping atomicity. This resembles regular SSDs, where single-page write atomicity is guaranteed and maybe only some of data pages in a write command are persisted after a sudden power outage. Atomic remap commands are similar to the advanced

[2] For SSDs that do not employ a superblock-based FTL, our design still applies and the only change is that the granularity becomes a flash block.

[3] The interface protocols may require an SSD to return an error or some deterministic value (e.g., zeros) when a deallocated LPN is read [4].

Table 2: Remapping metadata entry.

First 8 bytes		Second 8 bytes	
0	Torn bit	64	Torn bit
1-21	Flash page offset in superblock	65-95	Target LPN
		96	Remapping flag
22-63	Write/Remap sequence number	97-127	Null or source LPN

atomic-write primitives proposed in [49,52] and NVMe specification [4]. Although these atomic commands are not widely used yet, they provide an option to reduce the complexity and overheads for atomicity assurance in the host software. In the current implementation, Remap-SSD ensures only remapping atomicity by default.

The third goal of elaborating a RMM entry is to improve the space efficiency, which can be realized by compacting its fields. The target PPN is replaced by its *physical page offset* in the resident flash superblock, as each NVRAM segment is dedicated to a specific superblock. Also, the unused bits in LPN fields can be utilized. Assuming the SSD capacity and page size are 4TB and 4KB, respectively, a 4B LPN field can spare two bits for holding the torn bit and/or remapping flag. Table 2 shows an example layout of a RMM entry, whose size is 16B. The entry size can be extended to 24B, if any fields demand more bits or command atomicity is required.

Besides RMM entries, each NVRAM segment contains a *segment metadata entry* in its head slot. This entry stores a *flash superblock ID* which the segment is associated with, the current *write/remap sequence number*, a *segment sequence number* among the segments allocated to the superblock, and a *next segment ID* that links the segments in a group. The former three fields are written immediately when the segment is allocated, while the next segment ID is written when the segment is full and a next free segment is allocated. The association relationships between flash superblocks and NVRAM segments can be restored from segment metadata entries.

4.4 Garbage Collection

Both the writes of data pages to flash superblocks and RMM entries to NVRAM segments are conducted in a log-structured fashion. Thus, GC is required to reclaim invalid flash pages and invalid RMM entries.

When free flash superblocks run out, a flash GC operation is triggered on a victim superblock (e.g., with the most invalid pages). Since address remapping is enabled, a flash page may be referenced by multiple LPNs. Only flash pages without any references are invalid and can be recycled. The FTL maintains a *reference counting table (RC-table)* to track the number of references to each flash page. Consider the number of writes on most duplicate data is small (e.g., smaller than ten [16,34]). Four-bit counters are used by default.

NVRAM GC is performed both passively and actively. Reclamation of a flash superblock causes a passive recycle on its NVRAM segment group. An active recycle is triggered when free NVRAM segments run out. The NVRAM segment group with the most invalid RMM entries will be selected as the victim. Invalid RMM entries are those whose P2L mappings are *not* consistent with the latest L2P mappings. The FTL tracks the number of invalid RMM entries in each segment group. Specifically, a bitmap is used to indicate whether the current L2P mapping of each LPN is established by a remap or write operation, called *LR-bitmap*. When a remapped LPN is remapped again or written to a new PPN, the number of invalid RMM entries in the segment group of the flash superblock where the stale PPN resides increases by one. To recycle a segment group, RMM entries in it are checked, where valid entries are migrated to a new group of free segments and invalid ones are discarded. Then, the stale segment group is zeroed to be free.

The usage of address remapping is limited by both the reference counting capability and the NVRAM capacity. If the counter of a flash page reaches its maximum, remapping to this page is prohibited. Also, if all the NVRAM segments are filled with valid RMM entries, remap operations are disabled. It is important to note that these two cases do *not* mean Remap-SSD would return a failure on the relevant remap command and require the host software to perform error handling. Instead, Remap-SSD internally transforms the prevented remap operations to regular physical writes of duplicate data pages, which is transparent to the host. Therefore, host software can maximize the utilization of remap commands without concerning the operational details inside the SSD. In addition, to restrain NVRAM GC overheads, remap operations are disabled when the ratio of valid RMM entries is larger than a high watermark (95% by default).

4.5 Power-off Recovery

Power-off recovery aims to recover the FTL to a consistent state with the latest mappings after sudden power outages. The key is to ensure P2L mappings of data pages that have been written on flash memory are persistent. Then, the most recent L2P mapping table can be rebuilt from P2L mappings.

Remap-SSD maintains *head and tail metadata* in each flash superblock for fast power-off recovery, similar to conventional SSDs [8]. When a flash superblock is allocated, head metadata are written first before any data writes, including at least the type, write timestamp, and erase count of the superblock. The type indicates whether the superblock stores host data pages or FTL metadata or other vendor-specific information. Write timestamps preserve the write order of superblocks. Note that flash pages in a block must be written sequentially and blocks in a superblock can be written in parallel. Remap-SSD chooses the first flash page of the *Xth* block in a superblock to keep head metadata, where *X* is the modulus of superblock ID and the number of blocks each superblock contains. This

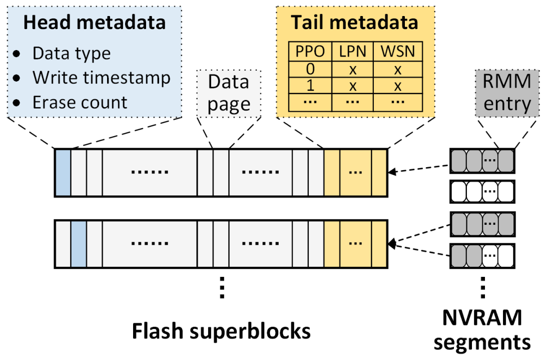


Figure 4: **Persistent metadata for power-off recovery.** *WSN*: write/remap sequence number, *PPO*: physical page offset in the superblock. The latest L2P mappings can be rebuilt from persistent metadata in flash superblocks and NVRAM.

enables concurrent reads to head metadata of different superblocks. Tail metadata are retained in the last several flash pages in each data superblock. They collectively hold the P2L mappings and write/remap sequence numbers of data pages that have been written in the superblock.

Power-off recovery of Remap-SSD relies on the head and tail metadata in flash superblocks and remapping metadata in NVRAM segments, as shown in Figure 4. The main recovery procedure includes three steps. First, head metadata of all flash superblocks are read to identify superblocks storing host data, which are organized in write time order. Second, tail metadata of superblocks are scanned in write time order, from which we can obtain the L2P mapping table established by data page writes and these writes' timestamps. The power-off recovery of traditional SSDs ends after this step. Third, Remap-SSD examines all NVRAM segments. Based on intact RMM entries whose timestamps are more recent than the write timestamps of relevant data pages, the changes to L2P mappings caused by the latest remap operations are applied. As the latest L2P mapping table has been recovered, the RC-table is also acquired. Moreover, segment metadata entries are used to restore the SV-bitmap and association relationships between flash superblocks and NVRAM segment groups.

4.6 Discussion

Hybrid Storage Architecture. One might wonder whether it is necessary for Remap-SSD to employ a hybrid storage architecture or whether NVRAM segments can be replaced by reserved flash pages. We argue that pure flash storage is *not* adequate to address the mapping inconsistency problem. This is mainly due to the size discrepancy between RMM entry and flash write unit. If NVRAM is not adopted, for each flash superblock containing remapped data, its RMM entries would have to be cached in DRAM and accumulate to a page size before being written to a flash page. Then, there would be a large amount of cached entries (from many superblocks) facing the risk of loss if sudden power outages occur. It is feasible to use supercapacitors for some level of power loss protection and periodically flush cached entries. However, this would lead to write amplification and underutilized storage space when cached entries of a superblock cannot fill a page. Also, supercapacitors increase the cost and raise new reliability concerns (e.g., aging effect [11]).

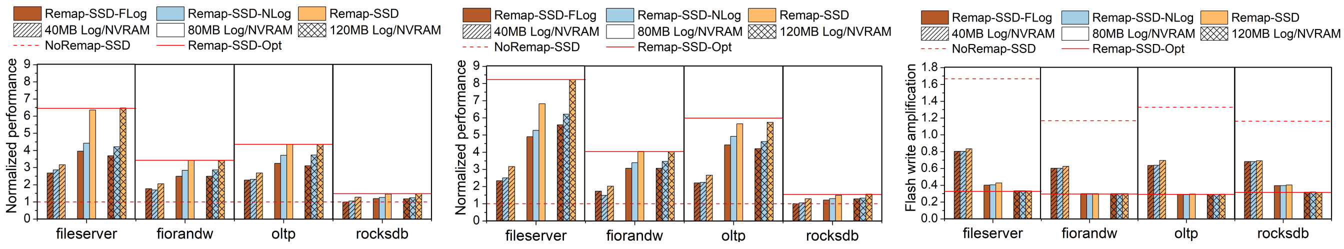
We should note that adding NVRAM in Remap-SSD has high cost-efficiency. The requirement for NVRAM capacity is small. Writes of every 1GB duplicate data through address remapping only produce 4MB RMM. In contrast, the utilization of remapping brings large savings on storage space and cost. Assume PCRAM, whose bit cost is roughly 5 times that of flash memory [47], is in use. The cost of storing RMM on PCRAM is only about 2% of the cost of storing duplicate data on flash memory. On the other hand, given 1GB NVRAM, which can accommodate a maximum of 256GB duplicate data, its cost can be compensated as long as 5GB flash storage space is saved. In addition, the NVRAM lifetime is not a concern, since NVRAM has more than 1,000 times better write endurance than flash memory.

Metadata Overheads. Compared with traditional SSDs whose address remapping ability is not exposed, Remap-SSD introduces extra metadata overheads. First, remapping metadata and segment metadata are stored in NVRAM. The NVRAM capacity limits the maximum number of valid RMM entries and thus unique LPNs that can be remapped. The segment metadata size is inversely proportional to the segment size, for example, 1.6% of NVRAM capacity when the segment size is 1KB. Second, the SV-bitmap (see Section 4.2), RC-table, and LR-bitmap (see Section 4.4) are maintained in DRAM or NVRAM (if DRAM is too small). The SV-bitmap size is negligible. The sizes of RC-table and LR-bitmap are proportional to the physical and logical capacities of the SSD, respectively. Assume the logical and physical capacities of the SSD are 4TB and 5TB, respectively, and the page size is 4KB. The RC-table (with 4-bit counters) size in Remap-SSD is 640MB, while that (with 1-bit counters) in conventional SSDs is 160MB. The LR-bitmap occupies 128MB space and can be embedded into the L2P mapping table if its PPN field has any unused bit.

5 Case Studies and Evaluation

5.1 Experimental Setups

To evaluate Remap-SSD, we perform three case studies with various applications: *intra-SSD deduplication*, *write-ahead logging in SQLite*, and *cleaning in F2FS*. Remap-SSD is compared with one scheme, called *NoRemap-SSD*, which does *not* exploit SSD address remapping, and three other schemes, which exploit SSD address remapping but differentiate in how to guarantee the mapping consistency. *Remap-SSD-FLog* maintains a dedicated log of RMM entries stored on parallel



(a) Normalized performance (10% duplicate data). (b) Normalized performance (30% duplicate data). (c) Flash write amplification (30% duplicate data).

Figure 5: **Intra-SSD deduplication with 10% and 30% data duplication ratios.** Performance (bandwidth or throughput) numbers are normalized to those of NoRemap-SSD, which does *not* perform deduplication. Remap-SSD-FLog, Remap-SSD-NLog, and Remap-SSD are evaluated in each workload with three log/NVRAM sizes, i.e., 40MB, 80MB, and 120MB. Flash write amplifications (lower is better) with 10% duplicate data are not shown as they present similar insights to Figure (c).

flash dies. This scheme corresponds to the commonly adopted solution in existing studies listed in Table 1. *Remap-SSD-NLog* enhances Remap-SSD-FLog by using NVRAM to store the log. *Remap-SSD-Opt* is an optimal case assuming RMM entries can always be retrieved in O(1) time. It also represents prior studies (i.e., PebbleSSD [28] and WAL-SSD [24]) that target only specific applications of remapping. The maximum usage of remapping in Remap-SSD-FLog, Remap-SSD-NLog, and Remap-SSD is restricted by the log/NVRAM size, while Remap-SSD-Opt has no limit. The NVRAM segment size is set as 1KB by default in Remap-SSD.

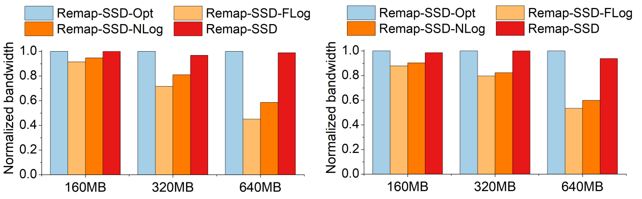
Most experiments are conducted on FEMU, a QEMU-based NVMe SSD emulator [38]. FEMU runs in a machine with 3.80GHz 16-core Intel i7-9800X CPU and 64GB DRAM. The emulated SSD is configured with 32GB logical capacity plus 4GB over-provisioning space (the total capacity is limited by DRAM size of the machine). Every flash block has 1024 pages whose size is 4KB. Each superblock contains 16 blocks, since the SSD consists of 16 parallel dies (each die has one plane). The flash read, write, and erase latencies are 50µs, 500µs, and 5ms, respectively. The NVRAM read and write latencies are 50ns and 500ns per 64B, respectively. In addition, we carry out some experiments of intra-SSD deduplication on SSDsim, a popular SSD simulator [25], to evaluate the schemes with a larger SSD and real-world traces. The simulated SSD has 256GB/288GB logical/physical capacity and 32 dies, while the flash block size remains unchanged. Write-dominant workloads are used for evaluation, since our work aims to reduce duplicate writes.

5.2 Intra-SSD Deduplication

Intra-SSD deduplication is a case worthwhile for studying for two reasons. First, data duplication incurs extensive duplicate writes, demanding the exploitation of address remapping. Second, deduplication generates complex *UN*-type remapping behaviors, similar to those in copying or snapshotting files. Such behaviors challenge the schemes for maintaining mapping consistency, so their efficiency differences can be clearly presented. In all the schemes excluding NoRemap-SSD, we implement a deduplication engine in the FTL, similar to CAFTL [16]. The FTL maintains a hash-based fingerprint store and computes the fingerprint of each logical data page written from the host. We assume a hardware hash unit is used and the computational overhead is 32µs [22]. If a fingerprint hits the store, the remap primitive is used to map the logical page to be written to the existing logical page that has the same content. Otherwise, the fingerprint is unique and added to the store and the logical page is written to flash memory.

We conduct two sets of experiments on FEMU-SSD running benchmark tools and on SSDsim running real-world traces. Benchmarks include the *fileserver* and *oltp* workloads in *filebench* [2], updating *RocksDB* with a zipfian request distribution in *YCSB* [6], and random-write workload (*fiorandw* for short) in *fio* [3]. These benchmarks do *not* include content locality in their data sets. Thus, we use their I/O patterns and simulate contents of logical data pages using a *zipf* distribution, which has been verified in characterizing the content popularity [22]. The distribution is expressed by $P(t_i) = C/t_i^a$, where, $C = 1/(\sum_{i=1}^{N} t_i^{-a})$, N is the number of unique contents in the data set, a is the *zipf* parameter representing the skewness in content popularity. We set a as 0.2 and the data duplication ratio as 10% or 30% (N equals to 90% or 70% of the total number of logical data pages, respectively). Real-workload traces include *homes* and *mail*, collected from production systems at FIU [22]. They contain real fingerprints of data pages, which can be used for deduplication.

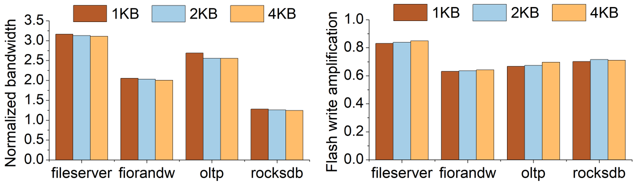
Figure 5 shows the performance and flash *write amplifications (WAs)* of the five schemes when data duplication ratio is 10% and 30%. The performance metric is bandwidth or throughput (operations per second), which is measured by benchmark tools. The WA results from valid data migrations during GC and is calculated as the ratio between total flash page writes and host page writes. Compared to NoRemap-SSD, the other schemes significantly improve the storage performance (e.g., by 1.5~8.2 times in Remap-SSD-Opt) and reduce the WA below one (e.g., by 40.5%~80.4% in Remap-SSD-Opt). Such benefits stem from intra-SSD data dedupli-



(a) SSD bandwidth in *homes*.

(b) SSD bandwidth in *mail*.

Figure 6: **Intra-SSD deduplication with real-world traces.** Bandwidth values are normalized to those of Remap-SSD-Opt. Different log/NVRAM sizes, 160MB, 320MB, and 640MB, are evaluated (SSD capacity is 256GB). Bandwidths of NoRemap-SSD are 6~40 times lower than those of the other schemes and are not shown in the figures.



(a) Normalized performance.

(b) Flash write amplification.

Figure 7: **Impacts of NVRAM segment size in Remap-SSD under intra-SSD deduplication (10% duplicate data).** The NVRAM size is 40MB. With a larger NVRAM, the impacts of segment size decrease.

cation, which completes host writes of duplicate data through quick remap operations without performing flash page writes. Moreover, deduplication reduces the GC overheads since it results in smaller storage space consumption and thus larger over-provisioning space.

For the three schemes that log remapping metadata (i.e., Remap-SSD-FLog, Remap-SSD-NLog, and Remap-SSD), the log/NVRAM size is a critical factor that affects their performance and WA. When the log/NVRAM size is enlarged, the performance increases because more RMM entries or remap operations can be afforded. With 30% data duplication ratio, 17%~34% of remap operations are demoted to regular flash writes (because the log/NVRAM is full) when the log/NVRAM size is 40MB. The percentages become up to 4.5% and 0%, respectively, when the log/NVRAM sizes are 80MB and 120MB. Compared to Remap-SSD-FLog and Remap-SSD-NLog, Remap-SSD improves the performance by an average of 20.2% and 17%, respectively, when the log/NVRAM size is 40MB. The improvements increase to 38.5% and 24.3% for an 80MB log/NVRAM, and further to 44.3% and 26.8% for a 120MB log/NVRAM. The main reason behind these performance improvements is that Remap-SSD-FLog and Remap-SSD-NLog suffer from high overheads of scanning the entire log, no matter on flash memory or faster NVRAM, in every GC operation. The larger the log size is, the higher the overheads are. In contrast, Remap-SSD always achieves fast lookups by maintaining a small local log for each GC unit on demand, rather than a global log.

On the other hand, Remap-SSD has slightly higher WAs than Remap-SSD-FLog and Remap-SSD-NLog when the log/NVRAM size is small, such as an average of 4.5% and 2.3% for log/NVRAM sizes of 40MB and 80MB, respectively. When the log/NVRAM size increases to 120MB, the three schemes obtain similar WAs. This is because Remap-SSD allocates NVRAM segments for separate local logs and may leave some segments underutilized, while Remap-SSD-FLog and Remap-SSD-NLog can fully utilize the flash/NVRAM log space and undertake more remap operations. When a larger log/NVRAM is used, the gaps on space utilization and remapping efficiency narrow.

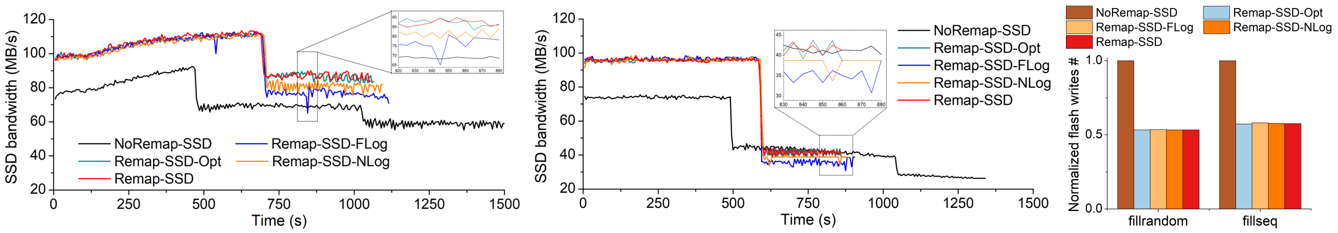
We also study the performance of Remap-SSD with a larger SSD and real-world traces, as shown in Figure 6. Before running each trace, we age the SSD by issuing random writes until flash GC is triggered and by filling NVRAM with 70% valid RMM entries with random LPNs. When the log/NVRAM size is 160MB, 320MB, and 640MB, Remap-SSD averagely improves the performance by 10.7%, 32.1%, and 97.3%, compared to Remap-SSD-FLog, and 7.2%, 22%, and 62.6% compared to Remap-SSD-NLog, respectively. Furthermore, Remap-SSD has close performance to Remap-SSD-Opt, e.g., an average of 2.1% and up to 6.2% lower performance. These results demonstrate rapidly increasing performance overheads of employing a global log when the log size grows and, on the other hand, the good scalability of Remap-SSD. Besides, the three schemes have similar WAs (not shown in figures), as segmenting large NVRAM in Remap-SSD negligibly degrades the space utilization.

Figure 7 shows sensitivity studies on the NVRAM segment size in Remap-SSD. A larger segment size results in trivial performance degradations and slight WA increases. This is because space utilization of NVRAM decreases as the allocation unit is enlarged. We set the segment size as 1KB by default, despite marginally higher segment metadata overheads.

From above results, we can make two conclusions. First, maintaining a global log for remapping metadata causes significant performance overheads, which are proportional to the log size. Second, Remap-SSD provides an efficient and scalable scheme that can maximize the utilization of SSD address remapping while ensuring the mapping consistency. When the NVRAM size increases, Remap-SSD's performance does *not* degrade and keeps comparable with that of Remap-SSD-Opt.

5.3 Write-ahead Logging in SQLite

Write-ahead logging (WAL) is a widely used approach for transactional atomicity in databases and file systems [24]. All modifications on the database file are written to a WAL file and then applied to original locations during checkpoint operations. With Remap-SSD, checkpointing writes can be



(a) SSD bandwidth under *fillrandom* workload.

(b) SSD bandwidth under *fillseq* workload.

(c) Flash page writes.

Figure 8: **Performance results of SQLite.** Numbers of flash page writes are normalized to those of NoRemap-SSD.

realized through the remap primitive, i.e., remapping LPNs of original locations to those in the WAL file. We use SQLite, a popular database [5], to verify Remap-SSD on reducing WAL overheads. One issue is that data pages in the SQLite WAL file are not page-aligned because they are interleaved with frame headers [37]. To make data pages aligned, we simply store frame headers collectively in reserved pages. The remap primitive is implemented as a new NVMe command and is invoked by SQLite through an extended *ioctl* system call.

We use the *db_bench* benchmark [1] to test SQLite (*synchronous*=NORMAL). Two tests are conducted: one writes 1.6 million values in random key order (*fillrandom*) and the other writes 1.5 million values in sequential key order (*fillseq*). The value size is 16KB. Figure 8 shows the SSD bandwidth over time and the numbers of total flash page writes of different schemes. Remap operations are counted in measuring the bandwidth. The log/NVRAM size is 80MB.

In each test, NoRemap-SSD sustains two sharp performance drops, e.g., at the time around 500s and 1000s in Figure 8(a). The first drop is because the SSD has undergone a full disk write and begins to conduct GC operations. At this time, the working set (i.e., the number of valid unique LPNs) size is moderate. As invalid flash pages has accumulated to a high level, GC overheads are small. Then, the working set grows and invalid flash pages are reclaimed over time, increasing the GC overheads significantly. This leads to the second performance drop. We can see the schemes that exploit SSD address remapping postpone the first performance drop and avoid the second drop, because remapping enables single-write WAL and largely reduces flash writes, e.g., by 44.5% on average (see Figure 8(c)). Also, the schemes with remapping finish the tests much faster than NoRemap-SSD. In addition, SSD bandwidth increases over time up to the first drop in Figure 8(a). The reason is that the ratio of reads, which originate from read-modify-write operations for small random updates, rises and the SSD processes reads faster than writes.

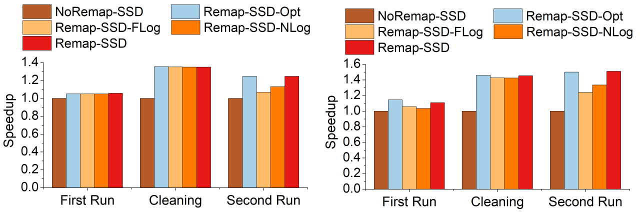
Remap-SSD always outperforms Remap-SSD-FLog and Remap-SSD-NLog, e.g., by an average of 15.1% and 7.8%, respectively, in the two workloads after GC has been triggered. Notably, Remap-SSD-FLog suffers from two bandwidth drops at time 540s and 845s in *fillrandom*. This owes to reclaiming invalid RMM entries in the log on flash memory, which is slower than that in Remap-SSD-NLog. The reclamation requires reading the entire log, writing back valid entries, and erasing flash blocks. In contrast, Remap-SSD looks up and reclaims RMM entries in a small unit, i.e., a segment group, whose largest size is found to be 117KB in the experiments of SQLite. These results exhibit the efficiency of RMM management in Remap-SSD.

We notice that there is a performance inversion between the schemes with remapping and NoRemap-SSD after the first performance drop at around 600s in Figure 8(b). This is attributed to higher GC overheads in the schemes with remapping. On the one hand, the schemes with remapping have a larger working set size than NoRemap-SSD at that time due to higher write bandwidth. On the other hand, despite eliminating WAL overheads, remapping reduces the number of invalid flash pages and thus GC efficiency. In NoRemap-SSD, the WAL file is overwritten repeatedly when it becomes full and its contents have been applied to the database file. Such overwrites lead to invalidation of flash pages that store obsolete WAL contents. By contrast, these flash pages remain valid in the schemes with remapping, because they are remapped to and referenced by relevant logical pages in the database file. As the working set size grows and invalid flash pages are reclaimed by GC over time in NoRemap-SSD, its GC overheads increase and the performance inversion between it and Remap-SSD ends.

5.4 Cleaning in F2FS

Considering the detrimental effects of random writes on SSDs, log-structured file systems naturally fit for SSDs and have drawn close attention [35]. They provide write sequentiality by organizing data in logs. However, cleaning is required to reclaim invalid data blocks. Similar to and independent from intra-SSD GC, the log cleaning process includes migrating valid data blocks and thus introduces duplicate writes. We modify F2FS, a state-of-the-art and popular log-structured file system designed for flash devices [35], to utilize the remap primitive for migrating valid data blocks at almost zero cost.

Two workloads are used for testing F2FS: the *fileserver* workload in *filebench*, updating *MongoDB* with a zipfian request distribution in *YCSB* [6]. Each test consists of three successive phases: (1) running the workload to generate in-



(a) *fileserver in filebench.*

(b) *YCSB on MongoDB.*

Figure 9: **Speedups in F2FS.** Performance is normalized to that of NoRemap-SSD. The log/NVRAM size is 80MB.

valid data blocks in F2FS; (2) manually triggering cleaning operations until all invalid data blocks in F2FS are reclaimed; (3) running the workload for the second time for performance evaluation. Figure 9 shows the speedups of the schemes with remapping over NoRemap-SSD on above three phases. The utilization of SSD address remapping accelerates the cleaning process (i.e., the second phase) by an average of 28.3% and improves F2FS performance at runtime by up to 50%. The cleaning process includes a large number of remap operations. Then, Remap-SSD-FLog and Remap-SSD-NLog contain much more RMM entries in the log in the third phase than in the first phase. As a result, average performance improvements of Remap-SSD over Remap-SSD-FLog and Remap-SSD-NLog are 2.8% and 4% in the first phase but increase to 19.1% and 11.6% in the third phase, respectively. These results verify the efficiency and scalability of Remap-SSD in exploiting SSD address remapping.

6 Related Work

Innovative SSD architectures have been an active field of study in both academia and industry. Below we discuss some representative designs in two areas related to Remap-SSD, i.e., *novel SSD interfaces* and *hybrid SSD architectures*.

Novel SSD interfaces. The conventional block interface impedes hardware-software co-designs that can maximally exploit the performance characteristics of flash storage. Hence, several new SSD interfaces have been devised. A number of designs employ remap or similar primitives to reduce duplicate writes by utilizing the SSD address remapping utility [16, 17, 23, 24, 28, 45, 46, 60]. Compared to these designs, Remap-SSD avoids their limitations on the usage of remapping (see Section 3) by solving the mapping inconsistency problem in an efficient manner.

Atomic-write interfaces have also been proposed by leveraging the copy-on-write nature of the FTL [31, 49, 52]. Through the interfaces, the burden of ensuring transactional atomicity can be removed from the host software. To eliminate redundant log layers across the storage stack and provide predicable performance, the open-channel and ZNS (zoned namespaces) interfaces allow the host to directly manipulate data layout on flash memory [13, 36, 48]. Recently, key-value (KV) interfaces [29, 32, 61] and dual block- and byte-addressable interfaces [9, 12] have been presented for SSDs. KV-SSDs consolidate KV management with the FTL to provide high-performance and scalable KV stores. Dual-interface SSDs open a fast and fine-grained path to access SSDs. Besides, Willow [53] proposed a user-programmable SSD that enables flexible interactions between the host and SSD. These schemes and Remap-SSD share the same design philosophy of breaking the block interface.

Hybrid SSD architectures. To address the idiosyncrasies of flash memory and take advantage of emerging NVRAM technologies, hybrid SSD architectures have been studied. NVRAM can be used in different ways for various purposes, e.g., to store the L2P mapping table for fast and energy-efficient address translation [26], to absorb small updates to data pages on flash memory [57], to replace flash OOB for supporting byte-addressable metadata [28], and to store intra-SSD RAID parity for reducing parity updating overheads [27, 67]. These efforts along with Remap-SSD demonstrate the large design space and great potentials of hybrid SSD architectures.

In addition, our design on the co-management of NVRAM and flash storage is partially inspired by the co-management of reserved space and value storage in HashKV [15]. As a KV store built on KV separation, HashKV divides value storage into fix-sized partitions and allows a partition to grow on demand by allocating segments in reserved space.

7 Conclusion

Reducing flash writes has been a long-standing goal in deploying SSDs. In this paper, we present Remap-SSD, which exports a remap interface and employs a flash and NVRAM hybrid storage architecture. It allows the host and FTL to maximally exploit the address remapping facility for eliminating duplicate writes. Meanwhile, Remap-SSD ensures the latest mappings can always be retrieved quickly and recovered from house-keeping metadata persisted on flash memory and NVRAM together with written or remapped data. Through three practical case studies, we demonstrate Remap-SSD delivers a safe, efficient, and scalable solution that exploits SSD address remapping for performance and lifetime improvements.

Acknowledgments

We would like to thank our shepherd, Patrick P. C. Lee, and the anonymous reviewers for their valuable feedback. This work was supported in part by the NSFC under Grant No. 61902137, No. U2001203, No. 61872413, No. 61821003, Key Area Research and Development Program of Guangdong Province under Grant No. 2019B010107001, National Key Research and Development Program of China under Grant No.2018YFB1003305, the 111 Project (No. B07038), and Key Laboratory of Information Storage System, Ministry of Education of China.

References

[1] Database Microbenchmarks. http://www.lmdb.tech/bench/microbench/.

[2] Filebench Benchmark. https://github.com/filebench/filebench/wiki.

[3] Fio Benchmark. https://github.com/axboe/fio.

[4] NVM express base specification. https://nvmexpress.org/resources/specifications/.

[5] SQLite Home Page. https://www.sqlite.org/index.html.

[6] YCSB Benchmark. https://github.com/brianfrankcooper/YCSB.

[7] Intel Optane Memory H10 with Solid State Storage. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-memory/optane-memory-h10.html, 2019.

[8] Flash translation layer in the storage performance development kit (SPDK). https://spdk.io/doc/ftl.html, 2020.

[9] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. FlatFlash: Exploiting the byte-accessibility of SSDs within a unified memory-storage hierarchy. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019.

[10] Mohammadamin Ajdari, Pyeongsu Park, Joonsung Kim, Dongup Kwon, and Jangwoo Kim. CIDR: A cost-effective in-line data reduction system for terabit-per-second scale SSD arrays. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*, pages 28–41, 2019.

[11] G. Alcicek, H. Gualous, P. Venet, R. Gallay, and A. Miraoui. Experimental study of temperature effect on ultracapacitor ageing. In *Proceedings of the European Conference on Power Electronics and Applications*, 2007.

[12] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2B-SSD: The case for dual, byte- and block-addressable solid-state drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*, 2018.

[13] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, 2017.

[14] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 105(9):1666–1704, 2017.

[15] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. HashKV: Enabling efficient updates in KV storage via hashing. In *Proceedings of the USENIX Annual Technical Conference (ATC'18)*, 2018.

[16] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*, pages 77–90, 2011.

[17] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. Jftl: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage*, 4(4), 2009.

[18] Kevin Conley. Flash: The Great Disruptor. Flash Memory Summit, 2015.

[19] Bob Fine. Mckesson mixes SSDs with HDDs for optimal performance and ROI. Flash Memory Summit, 2016.

[20] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. Amber*: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[21] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS'09)*, 2009.

[22] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*, 2011.

[23] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of the USENIX Annual Technical Conference (ATC'17)*, pages 759–771, 2017.

[24] Kyuhwa Han, Hyukjoong Kim, and Dongkun Shin. WAL-SSD: Address remapping-based write-ahead-logging solid-state disks. *IEEE Transactions on Computers*, 69(2):260–273, 2020.

[25] 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 ACM International Conference on Supercomputing (ICS'11)*, 2011.

[26] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Shuping Zhang, Jingning Liu, Wei Tong, Yi Qin, and Liuzheng Wang. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. In *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies (MSST'10)*, 2010.

[27] Soojun Im, Dongkun Shin, Dongkun Shin, Dongkun Shin, and Dongkun Shin. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Transactions on Computers*, 60(1):80–92, 2011.

[28] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. Improving ssd lifetime with byte-addressable metadata. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'17)*, page 374–384, 2017.

[29] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*, 2017.

[30] Myoungsoo Jung and Mahmut T Kandemir. Sprinkler: maximizing resource utilization in many-chip solid state disks. In *Proceedings of the 20st IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*, 2014.

[31] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, 2013.

[32] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*, 2019.

[33] Bryan S. Kim, Jongmoo Choi, and Sang Lyul Min. Design tradeoffs for SSD reliability. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*, 2019.

[34] Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. Deduplication in SSDs: Model and quantitative analysis. In *Proceedings of the 28th IEEE Symposium on Mass Storage Systems and Technologies (MSST'12)*, 2012.

[35] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: a new file system for flash storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'15)*, 2015.

[36] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, 2016.

[37] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proceedings of the USENIX Annual Technical Conference (ATC'15)*, 2015.

[38] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, 2018.

[39] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. Cachededup: In-line deduplication for flash caching. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 301–314, 2016.

[40] Lloyd Liu. MRAM based NVMe SSD Architecture. Flash Memory Summit, 2019.

[41] Dongzhe Ma, Jianhua Feng, and Guoliang Li. LazyFTL: A page-level flash translation layer optimized for NAND flash memory. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2011.

[42] Micron. How Micron SSDs Handle Unexpected Power Loss. https://www.micron.com/-/media/client/global/documents/products/white-paper/ssd_power_loss_protection_white_paper_lo.pdf, 2014.

[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 USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[44] Sparsh Mittal and Jeffrey S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2016.

[45] Fan Ni, Xingbo Wu, Weijun Li, Lei Wang, and Song Jiang. Leveraging ssd's flexible address mapping to accelerate data copy operations. In *Proceedings of the IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS'19)*, pages 1051–1059, 2019.

[46] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. Share interface in flash storage for relational and nosql databases. In *Proceedings of the International Conference on Management of Data (SIGMOD'16)*, page 343–354, 2016.

[47] Michael Oros. Analysts Weigh In On Persistent Memory. Persistent Memory Summit, 2018.

[48] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, 2014.

[49] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*, 2011.

[50] Jisung Park, Sungjin Lee, and Jihong Kim. DAC: Dedup-assisted compression scheme for improving lifetime of NAND storage systems. In *roceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*, 2017.

[51] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys*, 47(1), 2014.

[52] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[53] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable SSD. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[54] Scott Shadley. NAND flash media management through RAIN. https://www.micron.com/-/media/client/global/documents/products/technical-marketing-brief/brief_ssd_rain.pdf, 2011.

[55] Kai Shen, Stan Park, and Men Zhu. Journaling of journal is (almost) free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, pages 287–293, 2014.

[56] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a flash with ioSnap. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys'14)*, 2014.

[57] Guangyu Sun, Yongsoo Joo, Yibo Chen, Dimin Niu, Yuan Xie, Yiran Chen, and Hai Li. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA'10)*, 2010.

[58] Ying Y. Tai. High Performance FTL for PCIe/NVMe SSDs. Flash Memory Summit, 2016.

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

[60] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. ANViL: advanced virtualization for modern non-volatile memory devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, 2015.

[61] Sung-Ming Wu, Kai-Hsiang Lin, and Li-Pin Chang. KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE'2018)*, 2018.

[62] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[63] Zhichao Yan, Hong Jiang, Song Jiang, Yujuan Tan, and Hao Luo. SES-Dedup: a case for low-cost ECC-based SSD deduplication. In *Proceedings of the 35th Symposium on Mass Storage Systems and Technologies (MSST'19)*, 2019.

[64] Qirui Yang, Runyu Jin, and Ming Zhao. SmartDedup: Optimizing deduplication for resource-constrained devices. In *Proceedings of the USENIX Annual Technical Conference (ATC'19)*, pages 633–646, 2019.

[65] Miao-Chiang Yen, Shih-Yi Chang, and Li-Pin Chang. Lightweight, integrated data deduplication for write stress reduction of mobile flash storage. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2590–2600, 2018.

[66] Yang Zhan, Alexander Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. How to copy files. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 75–89, 2020.

[67] You Zhou, Fei Wu, Weizhou Huang, and Changsheng Xie. LiveSSD: A low-interference RAID scheme for hardware virtualized SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[68] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, 2018.

CheckFreq: Frequent, Fine-Grained DNN Checkpointing

Jayashree Mohan [*]
UT Austin

Amar Phanishayee
Microsoft Research

Vijay Chidambaram
UT Austin and VMware research

Abstract

Training Deep Neural Networks (DNNs) is a resource-hungry and time-consuming task. During training, the model performs computation at the GPU to learn weights, repeatedly, over several epochs. The learned weights reside in GPU memory, and are occasionally checkpointed (written to persistent storage) for fault-tolerance. Traditionally, model parameters are checkpointed at epoch boundaries; for modern deep networks, an epoch runs for several hours. An interruption to the training job due to preemption, node failure, or process failure, therefore results in the loss of several hours worth of GPU work on recovery.

We present CheckFreq, an automatic, fine-grained checkpointing framework that (1) algorithmically determines the checkpointing frequency at the granularity of iterations using *systematic online profiling*, (2) dynamically tunes checkpointing frequency at runtime to bound the checkpointing overhead using *adaptive rate tuning*, (3) maintains the training data invariant of using each item in the dataset exactly once per epoch by checkpointing data loader state using a *light-weight resumable iterator*, and (4) carefully pipelines checkpointing with computation to reduce the checkpoint cost by introducing *two-phase checkpointing*. Our experiments on a variety of models, storage backends, and GPU generations show that CheckFreq can reduce the recovery time from hours to seconds while bounding the runtime overhead within 3.5%.

1 Introduction

Deep Neural Networks (DNNs) are widely used in many AI applications including image classification [20, 23, 41], language translation [46], and speech recognition [17]. While DNNs have facilitated state-of-the-art accuracy in these tasks, they come at the cost of high computational complexity, taking up to several days to train [8, 35].

Training starts with a randomly chosen set of learnable parameters (such as weights and biases) and proceeds in iterations consisting forward and backward pass over a minibatch of data. At the end of each backward pass, the learnable parameters are recomputed using the gradients obtained, and updated *in GPU memory*. Training is performed for several epochs, where one epoch is a complete pass over the dataset. At the end of training, the learned parameters are saved to persistent storage for inference.

Due to the large runtime of DNN training, the model weights and optimizer state (collectively, model state) are occasionally written to persistent storage, for fault tolerance; else, an interruption to the job due to process failure, or node crash can wipe out all the job state, resulting in loss of several hours of GPU work. This is termed *checkpointing*. Traditionally, models are checkpointed at epoch boundaries [30].

Interruptions to DNN training jobs are common. Be it dedicated enterprise clusters or cloud instances, failures due to software and hardware errors are inevitable. Prior work has shown that infrastructure and process failures are common in large-scale big data clusters, with a mean time between failure (MTBF) of 4 – 22 hours [19, 27]. Similarly, for GPU clusters, recent study of large-scale DNN training clusters at Microsoft [22] highlight that DNN training jobs encounter interruptions due to infrastructure failure, node crashes, software bugs, and user errors. Over the span of the analysis period (2 months), the mean time between job failures was 45 minutes on average (excluding early failures) in the Microsoft cluster.

Furthermore, a recent trend with cloud providers is the emergence of cost-effective preemptible VMs which are priced 6-8× cheaper than dedicated VMs [9, 16, 29]; such VMs may be preempted at any time. Recent work shows that GPU VMs may be preempted as frequent as every 15 minutes and atleast every 24 hours on the Google Cloud [31].

When interruptions occur, the long running, stateful, DNN job terminates abruptly, wiping out the model parameters in-memory. For instance, training ResNext101 to accuracy on ImageNet-1K dataset using a V100 GPU takes 270 hours (~3.9 hours per epoch) [35]; if checkpointing is performed at epoch boundaries, about two hours of GPU computation is wasted on average for every interruption. More generally, there is a trend of growing size of datasets [3, 7, 24], and larger, complex model architectures [8, 10, 35], consequently increasing DNN epoch time and overall training time. Therefore, it is critical to frequently checkpoint training progress, at a finer granularity than epochs *i.e.,* at iteration level. In this work, we explore how to perform fine-grained checkpointing automatically in a model- and hardware-agnostic manner, without intrusive changes to the training workload.

We present CheckFreq, a fine-grained checkpointing framework for DNN training. CheckFreq strikes a balance between ensuring a low runtime overhead and providing a high checkpointing frequency, so that there is minimal loss of GPU time in the event of job interruptions or failures by performing iteration-level checkpointing. CheckFreq has two major components; a checkpointing policy that automatically deter-

[*] Work done as part of MSR internship

mines *when to checkpoint*, and a checkpointing mechanism that performs *correct, low-cost checkpointing*. To this end, we build upon a set of techniques from the High Performance Computing (HPC) and storage community, alongside novel DNN-specific optimizations such as pipelined in-memory snapshots, utilizing spare GPU capabilities for fast snapshot, and a DNN-aware systematic profiling for dynamic tuning of checkpointing frequency. Using CheckFreq, we show that the recovery time reduces from hours to seconds during job interruptions.

Fine-grained checkpointing for DNNs at iteration granularity poses several unique challenges which CheckFreq addresses as described below.

1. **Checkpointing frequency**. There is no single checkpointing frequency that works across models, hardware, and training environments. The frequency of checkpointing depends on several factors; *e.g.,* model size, storage bandwidth, and training iteration time. Moreover, a job could face interference while writing checkpoints due to reading the dataset from the same storage device, or due to concurrently running jobs that share the storage bandwidth to write checkpoints. Statically determining a checkpointing frequency is sub-optimal for runtime if a job faces interference in its training environment.

Therefore, CheckFreq algorithmically determines an initial checkpointing frequency by profiling the job characteristics during runtime. CheckFreq uses *systematic online profiling* to determine the best-case checkpointing frequency for the model in the given training environment. However, in practice, the job might incur additional overheads due to intereference which slows down the checkpointing process. To tackle this, CheckFreq introduces *adaptive rate tuning* to dynamically monitor the job runtime between checkpoint intervals, and appropriately scale up or scale down the checkpointing frequency, so that the end-to-end runtime overhead is within a user-given bound.

2. **Checkpoint stalls**. The model state to be checkpointed is updated every iteration. Therefore, training has to briefly pause to accurately checkpoint the current state; the GPU (or any accelerator) remains idle until checkpoint completes, introducing *checkpoint stalls* in training. Naively increasing the frequency of checkpointing (*e.g.,* every iteration) results in high runtime overhead due to checkpoint stalls.

CheckFreq reduces checkpoint stalls using a *DNN-aware two-phase checkpointing* strategy. The checkpointing operation is split into a `snapshot()` and a `persist()` phase. In the `snapshot()` phase, CheckFreq performs a consistent in-memory copy of all the learnable model state. This operation is *pipelined* with compute until the weight update of the subsequent iteration which is the latest point when the model parameters are updated. In the `persist()` phase, the snapshot is asynchronously written to the storage device. CheckFreq guarantees that a checkpoint is reliably persisted on disk (using `fsync()`) before the subsequent checkpoint operation begins. Therefore, in the event of an unexpected interruption, the job state will rollback at most one checkpoint.

3. **Data invariant**. For a large class of models that perform random data pre-processing operations in every epoch of training (eg CNNs), it is crucial to ensure the following data invariant holds: every epoch must process *all* the items in the dataset *exactly once*, in a random order, with random pre-processing like crop, resize etc. Existing data iterators in frameworks like PyTorch, and MxNet do not support resumability. When the job is interrupted, these iterators can either miss out, or repeat data items in an epoch, resulting in loss in model accuracy when resuming at iteration granularity.

To address this challenge, CheckFreq introduces a resumable data iterator that respects the data invariant even in the presence of interruptions. The iterator uses epoch seeded psuedo-random transformations, that can reconstruct the iterator state as it was prior to interruption. CheckFreq's iterator thus makes correct, iteration-level checkpointing feasible.

We implement CheckFreq as a pluggable module for PyTorch, with minimal (< 10 LOC) changes to the original job's script. Our evaluation across a variety of models, GPUs, and storage types confirms that CheckFreq reduces the wasted GPU time from order of hours to just under a minute, while incurring less than 3.5% runtime overhead, as compared to the existing epoch-based checkpointing schemes. CheckFreq reduces the end-to-end training time by 2× when training a ResNet50 job on a 1080Ti GPU, and by 1.6× for a ResNext101 job on a V100 GPU, when the job is interrupted every 5 hours in both cases. We further demonstrate the importance of CheckFreq's recoverable iterator by training ResNet18 to accuracy using ImageNet dataset with frequent interruptions (once every 2 epochs) and iteration-level checkpointing; Existing state-of-the-art data loaders like DALI [4] result in up to 13% drop in accuracy while CheckFreq is able to train the model to target accuracy.

In summary, this paper makes the following contributions.

- Analyzes the state of DNN checkpointing today and highlights the need for fine-grained checkpointing and the challenges involved in achieveing it (§3)
- The design and implementation of CheckFreq, an automatic, fine-grained checkpointing framework for DNN training that exploits the DNN computational model to provide low-cost, pipelined checkpointing (§4)
- Experimental results demonstrating the efficacy of CheckFreq in reducing the recovery time from hours to seconds, across a range of models and hardware configurations (§5)

2 Background

This section provides a brief overview of the DNN computational model and the role of checkpointing in DNN training.

DNN computational model. Training a Deep Neural Network (DNN) is the process of determining the set of weights and bias in the network, collectively called the *learnable pa-*

rameters. Once trained, the DNN computes the output using the weights learned during the training phase.

DNN Training starts with a randomly chosen set of learnable parameters and proceeds iteratively in steps called *iterations*. Every iteration processes a small disjoint subset of the dataset called a *minibatch*. When the entire dataset is processed exactly once, an *epoch* is said to be complete. Each iteration of training performs the following steps in order.

- **Data augmentation**. Fetches a minibatch of data from storage and applies random pre-processing operations. For *e.g.,* in popular image classification models like the ResNets, pre-processing includes randomly cropping the input image, resizing, rotating, and flipping it.
- **Forward pass**. The model function is applied on the minibatch of data to obtain the prediction.
- **Backward pass**. A loss function is used to determine how much the prediction deviates from the correct answer; each layer in the DNN computes a gradient of the loss.
- **Weight update**. Using the gradients computed in the backward pass, the learnable model parameters are updated.

At the end of training (typically after a fixed number of epochs), the final learned parameters are saved to persistent storage. To perform inference on the model, the DNN is initialized with the learned parameters and the output is predicted.

Checkpointing. Training a DNN is a highly time-consuming task. For instance, BERT-large, the state-of-the-art language modeling network, takes 2.5 days to train [8], when trained in parallel across 16 V100 GPUs. Since the learnable parameters are maintained in GPU memory during training, any interruption to the training job due to a process crash, server crash, job or VM preemption, or job migration, results in the *loss* of model state learned so far. This state is typically a few hundred MBs to a few hundred GBs in size [36] (§5.4). Consequently, several hours of GPU time spent on training will be lost. To overcome this, the model state is typically *checkpointed* at epoch boundaries; *i.e.,* written out to persistent storage for fault-tolerance. This checkpoint can then be loaded when the training job resumes to ensure that progress is not entirely lost.

Recovery Time. When a DNN training job is interrupted, it rolls back to the last completed epoch that was checkpointed as shown in Figure 1. Note that, all the GPU work performed between the last checkpoint and the point of interruption is *lost* and has to be *redone* when training resumes. The amount of GPU time lost due to an interruption is termed the *recovery time*. In other words, this is the time spent to bring the model to the same state as it was prior to the interruption.

3 The Current State of Checkpointing

We analyze the current state of checkpointing in popular open source ML training frameworks like PyTorch [5], TensorFlow [6], and MxNet [11]. We analyze training workloads from MLPerf submissions v0.7, and the official workloads released by NVIDIA, TensorFlow and PyTorch. We find that checkpointing in open source ML training frameworks is incorrect and inefficient.

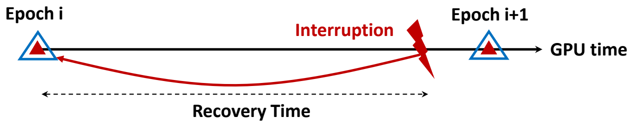


Figure 1: **Recovery time**. The amount of GPU work lost and has to be *redone* on recovery is termed the recovery time.

- **Correctness**. The checkpointing mechanism used in the training scripts could result in loss or corruption of checkpoint files in the event of job failure or interruption.
- **Efficiency**. Checkpointing is inefficient. The frequency of checkpointing is determined in an ad-hoc fashion, typically at epoch-boundaries which results in loss of several hours of GPU time for recovery. Furthermore, there is lack of support for checkpointing at fine granularity; existing data iterators do not support resuming training state at iteration boundaries and results in high checkpoint stalls.

3.1 Checkpointing is Incorrect

Corruption due to overwrites. Some of the official training workloads maintained by PyTorch [38], overwrite the same checkpoint file at the end of each epoch to reduce storage utilization. However, this exposes the risk of corrupting the checkpoint file in the event of a crash during the checkpoint operation. Prior work [37] has shown that different filesystems treat overwrites differently; a crash could result in non-atomic data update in the writeback mode of ext3 resulting in data corruption, while it could truncate the file on ext4, resulting in data loss. In either case, the checkpoint file becomes unusable; training has to restart from the first epoch.

The checkpoint file may not persist. Analyzing the primitives used by training frameworks for checkpointing, such as `torch.save` reveal that they do not `fsync()` the checkpoint file. We verified that this can lead to data loss. Moreover, naively performing frequent synchronous `fsync()` affects training performance significantly (§5.3.1).

3.2 Checkpointing is Inefficient

Checkpointing is performed sparingly in an ad-hoc fashion. There is no systematic checkpointing policy in the training jobs; checkpointing interval is chosen in an ad-hoc fashion. For example, some jobs do not checkpoint during training, while some others start checkpointing only after a large number of epochs (60% of training) have elapsed. In general, we observe that checkpointing is typically performed at epoch boundaries, providing only modest fault-tolerance; in the event of a job interruption, the training will resume from the last completed epoch, which potentially loses several hours of GPU training time that has to be redone. For instance, when ResNext101 is trained using ImageNet on a V100 GPU, *two* hours of GPU time is lost on average if the job is interrupted (§5.5).

A naive frequent checkpointing schedule results in checkpoint stalls. Providing higher fault-tolerance requires checkpointing to be performed more frequently than at epoch boundaries; *i.e.,* at iteration boundaries. However, naively increasing the frequency of checkpointing introduces a large checkpoint stall in training. Since model weights are constantly updated between iterations, checkpointing requires the training to briefly pause to capture the model weights accurately. We term this overhead (i.e, the time GPU is idle, waiting for the checkpoint to complete) as the *checkpoint stall*. Therefore, it is crucial to find the correct checkpointing frequency given a DNN (because the size of checkpoint varies from 100MBs to 100GBs across DNNs), and the storage bandwidth, to minimize checkpoint stalls.

Violating the data invariant during training can affect model accuracy. Each epoch performs a full pass over the dataset, in a random order and holds the invariant that *each data item is seen exactly once per epoch*. One of the benefits of checkpointing at epoch boundaries is that, the data iterator state need not be persisted, as it is reset at the end of epoch. Checkpointing at a finer granularity (i.e. at iterations), requires infrastructure support to resume the state of data iterator as well. We note that the support to persist iterator state exists in some custom dataloaders of NLP models which do not perform random pre-processing operations for every batch. However, for image and video models that apply random transformations on the input data every batch, the existing dataloaders in PyTorch, MxNet, and state-of-the-art data pipelines like NVIDIA's DALI are *not resumable* at iteration boundaries. As a result, they violate the data invariant in the presence of interruptions, resulting in upto the 13% drop in accuracy for popular models ResNet18 (Fig 6).

3.3 Summary

In summary, we observe that the checkpointing mechanism today is incorrect; resulting in potential checkpoint data loss or corruption. Additionally, the checkpointing policy is adhoc; there is no systematic way of determining how frequently one must checkpoint, to both minimize recovery time and incur low checkpoint stalls.

The solution to minimize recovery time is to perform frequent, iteration-level checkpointing. However, performing correct and efficient fine-grained checkpointing is challenging. We need (1) low-cost checkpointing mechanisms, (2) light-weight, resumable data iterators that preserve the model accuracy, and (3) a way to systematically determine the frequency of checkpointing.

4 CheckFreq: Design and Implementation

We present the goals of CheckFreq and the recovery guarantees it provides. We then present an overview of the overall architecture of CheckFreq, and discuss the techniques used by CheckFreq to achieve the enlisted goals.

Technique	***Benefits***
Checkpointing mechanism (How to checkpoint?)	
2-phase checkpointing	Splits checkpointing into two phases and pipelines them carefully with compute to make checkpoints cheap
Recoverable data iterator	Maintains data invariant, allows resuming training at iteration boundaries without affecting accuracy
Checkpointing policy (When to checkpoint?)	
Systematic online profiling	Automatically determines checkpointing frequency, cognizant of model characteristics
Adaptive rate tuning	Dynamically tunes checkpointing frequency to reduce overhead due to interference

Table 1: **Overview of techniques used by CheckFreq**.

4.1 Goals

Correctness. CheckFreq aims to provide frequent, iteration-level checkpointing that is consistent, and persistent.

No impact on model accuracy. CheckFreq aims to not impact the statistical efficiency of the model by ensuring that the data invariant holds when training resumes after interruption.

Automatic frequency selection. CheckFreq aims to determine and tune the frequency of checkpointing *automatically* based on the model being trained, and the training environment (GPU gen, storage type, iteration time). Checkpointing frequency influences the recovery time, *i.e.,* time to bring model state to what it was prior to the interruption.

Low checkpoint stalls. CheckFreq aims to reduce checkpoint stalls during training, so that there is low runtime overhead to frequent checkpointing (*e.g.,* $< 5\%$).

Minimal code changes. CheckFreq aims to require minimal changes to the training code to automate checkpoint management and restoration.

4.2 CheckFreq Recovery Guarantees

An interrupted job resumes training from the latest available checkpoint on disk. In the traditional epoch-based checkpointing, irrespective of when the job is interrupted, training resumes from the previous epoch boundary as shown in Fig 1. If a job performs n iterations per epoch and takes time t_i per iteration, then the average recovery time R_{avg} for this job is :

$$R_{avg} = \frac{n}{2} * t_i$$

This is because, when interrupted in the middle of an epoch, work done so far in the epoch must be redone when resumed, as the state is reset to the end of previous epoch. Thus, recovery time R for epoch-based checkpointing is bounded by:

$$0 \leq R \leq n * t_i$$

Note that $n * t_i$ is the duration of an epoch; it can be as large as a few hours. CheckFreq aims to provide a tight bound

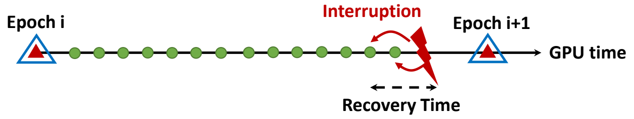


Figure 2: **Bounding recovery time**. CheckFreq guarantees that training rolls back at most one checkpoint.

on recovery time and takes a more fine-grained approach to checkpointing at iteration boundaries. CheckFreq guarantees that *there is at most one ongoing checkpoint operation in the system at any point in time*. When interrupted, it *rolls back at most one checkpoint* - either the last initiated checkpoint (if it completes), or the one prior as shown in Fig 2. If the frequency automatically determined by CheckFreq is k iterations, then CheckFreq guarantees that the recovery time R is bounded by

$$0 \leq R \leq 2*k*t_i$$

$$R_{avg} = k*t_i \quad (k << n)$$

The chosen checkpointing frequency k is $100-300\times$ less than n, as we show later in evaluation (§5.4), thereby resulting in orders of magnitude reduction in recovery time compared to epoch based checkpointing.

4.3 Design

We now present an overview of the architecture of CheckFreq and how it uses various techniques to provide frequent checkpointing at a bounded cost described in §4.2. Table 1 lists the different techniques used by CheckFreq and the benefit of each technique.

Overview. The architecture of CheckFreq is shown in Figure 3. CheckFreq has three major components; a recoverable data iterator that returns a minibatch of data to the training job, a feedback-driven checkpointing policy that determines when to trigger a checkpoint, and a low-cost checkpointing mechanism that is split into a `snapshot()` and a `persist()` phase. CheckFreq monitors the runtime overhead incurred in each checkpoint interval; this is used as feedback to dynamically tune the checkpointing frequency to ensure that the runtime overhead does not exceed a user-given limit p (*e.g.,* 5%). When interrupted, CheckFreq restores the latest available checkpoint and resumes training. We describe each component in detail below.

4.3.1 Checkpointing Mechanism

DNN checkpointing today is performed synchronously; training is paused until the checkpoint operation is complete. However, synchronous checkpointing introduces large *checkpoint stalls*, which results in large runtime overhead if performed frequently. In other words, the cost of a checkpoint (T_c) is high for synchronous checkpointing. For example, consider a policy that checkpoints every three iterations. The model state is written to disk after the weight update phase which updates weights based on the gradients computed in the backward pass. As shown in Figure 4a, the checkpoint cost is incurred in the critical path, resulting in high checkpoint stalls, which can significantly slow down the end-to-end training time. To mask such high checkpoint costs within an overhead p, checkpointing needs to be performed infrequently, which in turn results in high recovery cost.

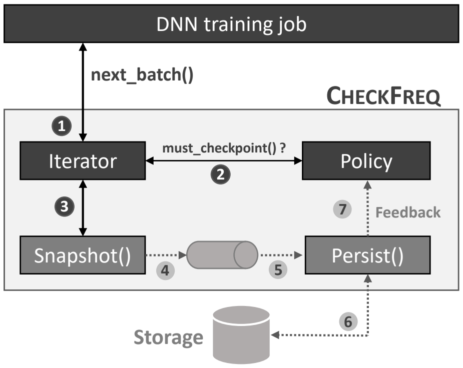


Figure 3: **Training with CheckFreq**. CheckFreq's policy determines the checkpointing frequency. The checkpointing mechanism then snapshots and persists the model and iterator state at the identified frequency in a pipelined manner. If a failure occurs, CheckFreq rolls back the model and iterator state to the latest available checkpoint and resumes training.

Two-phase checkpointing. CheckFreq aims to reduce the recovery cost in the event of an interruption by reducing checkpoint stalls. To achieve low checkpoint cost, CheckFreq introduces a *DNN-aware* two-phase checkpointing mechanism. CheckFreq splits checkpointing into two phases; `snapshot()` and `persist()` and pipelines each phase with computation. The main insight behind CheckFreq's two-phase checkpointing is that it exploits the DNN computational model (§2) to pipeline checkpointing operations on modern accelerators such as the GPUs.

1. **Phase 1 : `snapshot()`**. The first is a `snapshot()` phase, performed after the weight update step of the iteration. Here, a copy of the model state is captured in memory, so that it can be written out to storage asynchronously. Since the model state resides in GPU memory, `snapshot()` involves copying the model parameters from GPU to CPU memory. Performing this operation synchronously in the critical path results in non-trivial `snapshot()` overhead as shown in Figure 4b. Therefore, CheckFreq carefully *pipelines* `snapshot()` with compute.

   Pipelining `snapshot()` with compute has to be performed cautiously to ensure consistency of model parameters and preserve correctness of Stochastic Gradient Descent (SGD), which is a popular optimization technique used by learning algorithms. Naively pipelining them can result in an inconsistent snapshot that contains part of the weight updates from one iteration and the rest from the other. CheckFreq exploits the *DNN learning structure* to achieve correct, pipelined snapshots.

   We observe that the learnable model parameters are updated in GPU memory after the backward pass of an

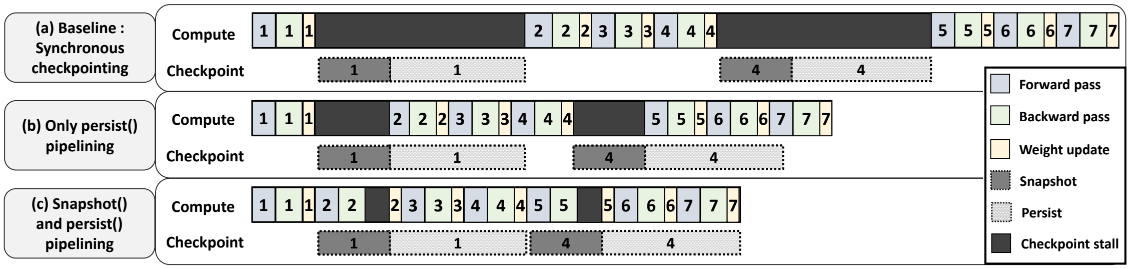

(a) Baseline : Synchronous checkpointing
(b) Only persist() pipelining
(c) Snapshot() and persist() pipelining
Compute
Checkpoint
1
4
Forward pass
Backward pass
Weight update
Snapshot
Persist
Checkpoint stall

Figure 4: **Pipelining checkpoint with compute**. This figure contrasts three checkpointing mechanisms, when checkpointing is performed every 3 iterations. (a) performs checkpointing synchronously and incurs a high checkpoint stall. (b) takes a snapshot of the model state synchronously but pipelines disk IO (`persist()`) with compute, allowing it to proceed in the background. CheckFreq takes a more nuanced approach by carefully pipelining `snapshot()` with the subsequent iteration's forward and backward pass and incurs lower checkpointing stalls as shown in (c)

iteration; in a step called the *weight update*. Therefore, we can pipeline `snapshot()` of iteration i with compute, until the weight update of iteration $i+1$. If `snapshot()` does not complete by then, then iteration $i+1$ waits until the ongoing `snapshot()` successfully completes as shown in Figure 4c. This tight coupling is required to ensure a consistent snapshot; else we might capture a state that is partially updated by the subsequent iteration that in turn affects the correctness of the learning algorithm [28].

GPU-based `snapshot()`. Although `snapshot()` is pipelined with compute of the following iteration, it may result in checkpointing stalls in cases where it is not possible to completely hide the cost of copying model state from GPU to CPU. Therefore, CheckFreq further optimizes this operation using a GPU-based `snapshot()` when feasible. We observe that the cost of performing a `snapshot()` in GPU memory is an order of magnitude cheaper than performing it to CPU memory, as the latter involves a GPU to CPU copy in the critical path. Therefore CheckFreq takes the following approach.

(a) When spare GPU memory is available in the training environment to hold a copy of the snapshot, we `snapshot()` in the GPU on GPU memory. The `persist()` phase then asynchronously copies the snapshot to CPU memory and then to disk.

(b) If not, CheckFreq snapshots directly into CPU memory. This can introduce stalls in critical path.

(c) CheckFreq adjusts the frequency of checkpointing appropriately to minimize the overhead of `snapshot()`, which can be especially large in (b), and stalls in `persist()`.

2. **Phase 2 : `persist()`**. The second phase in checkpointing is the `persist()` phase which asynchronously writes the snapshot to persistent storage similar to well explored asynchronous checkpointing techniques [33, 34, 40, 45]. However, to provide bounded rollback guarantees discussed in §4.2, `persist()` is *tightly coupled with compute*. CheckFreq performs the `persist()` operation as a background process; and monitors its progress. When a subsequent checkpoint is triggered as determined by the policy, the progress of the ongoing `persist()` operation is checked. If the `persist()` has not completed, then the compute process waits until the ongoing checkpoint operation is complete. This ensures that there is at most one ongoing checkpoint operation at any point in time, and if the job is interrupted, it rolls back to at most one prior checkpoint.

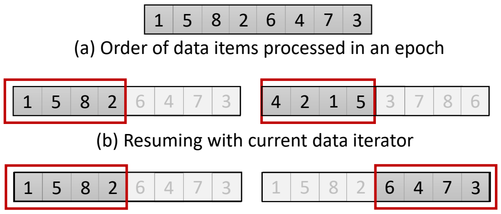
While it may be tempting to abandon an ongoing checkpoint if the next one is triggered, it is a tricky and risky operation. Suppose we abandon the current checkpoint and begin writing the next one, a failure at this point may end up losing both the checkpoints. This could be a chain reaction; a failure could result in rolling back to a significantly old checkpoint if all the recent ones were abandoned, resulting in a high recovery time. Since CheckFreq aims to guarantee that we roll back to at most one prior checkpoint, it does not abandon any running checkpoints.

Resumable light-weight data iterator. The DNN training workload interacts with CheckFreq using a thin API provided by a data iterator. The function of a data iterator in DNN training is to return a pre-processed batch of data items to the GPU, such that the data invariant holds - *each epoch processes all the data items exactly once, in a random order*. While the native iterator in PyTorch and those provided by state-of-the-art data pipelines like DALI [4] support this in the common case, they lack *resumability* if the training is interrupted.

For example, consider a dataset with eight data items from 1 – 8. In an epoch, the order of data items processed could be as shown in Fig 5a. Assume that we checkpoint the model state at the end of every iteration which processes *one* data item. If training is interrupted in the middle of this epoch, the data iterator loses state, and resumes with a random shuffled order of the dataset as shown in Fig 5b, resulting in data items being repeated and missed in a epoch, violating the data invariant.

CheckFreq's data iterator uses the following techniques to support resumption:

- It shuffles data items every epoch using a seed that is a function of the epoch number. Therefore, to recreate the



(a) Order of data items processed in an epoch

(b) Resuming with current data iterator

(c) Resuming with CheckFreq data iterator

Figure 5: **Resuming iterator state**. When iterator state is not resumable, an epoch might miss data items when job is interrupted (items 3,6,7 are missed in b). CheckFreq (c) ensures that training resumes from exactly where it left off.

same shuffle order, it is sufficient to persist the current epoch ID, and the number of data items processed so far (which makes iterator checkpointing lightweight).

- When training resumes, the iterator reconstructs the shuffle order, and deterministically restarts from where it left off at the last checkpoint as shown in Fig 5c.

Summary. Two-phase checkpointing mechanism along with the resumable data iterator provides correct, low-cost checkpointing. The next important question to answer is, *how frequently should we checkpoint the model?*

4.3.2 Checkpointing Policy

To perform automatic, iteration-level checkpointing, we must determine the frequency at which checkpointing is performed. On one hand, we can checkpoint after every iteration, providing low recovery cost but possibly high runtime overhead. On the other hand, we can perform coarse grained checkpointing at epoch boundaries, resulting in high recovery cost but low runtime overhead. An effective checkpointing policy must find the right balance between recovery cost and runtime overhead, minimizing both. The main idea behind CheckFreq's checkpointing policy is to initiate checkpoints every k iterations (called the checkpointing frequency), such that the overhead of one checkpointing operation can be amortized over k iterations. While prior work in HPC have explored ways of identifying the checkpointing frequency based on failure distribution in the cluster [12, 14, 15], CheckFreq finds the shortest interval that masks the overhead of checkpointing based on the DNN and hardware characteristics.

Systematic online profiling. CheckFreq takes a systematic profile-based approach to determine the checkpointing frequency. It should be chosen such that the runtime overhead introduced due to checkpointing is within a percentage p of the actual compute time, where p is the permissible overhead decided by the user (say 5%).

CheckFreq determines the initial checkpointing frequency as follows. When a training job starts, CheckFreq's data iterator (§4.3.1) automatically profiles several iteration-level and checkpoint-specific metrics which influences the checkpointing frequency - the iteration time (T_i), time to perform weight update (T_w), time to create an in-memory GPU copy (T_g), time to create an in-memory CPU copy (T_c), time to write to storage (T_s), size of checkpoint (m), peak GPU memory utilization (M), and total GPU memory (M_{max}). Based on CheckFreq's 2-phase checkpointing mechanism, the frequency determination algorithm is as shown in Algorithm 1.

Algorithm 1 : Checkpointing frequency determination

```
Input: $Ti, Tw, Tc, Tg, Ts, m, M, M_{max}, p$
  $T_{oc} \leftarrow max(0, T_c - (T_i - T_w))$
  $T_{og} \leftarrow T_g$
  if $M_{max} - M > m$ and $T_{og} \leq T_{oc}$ then
      $T_o \leftarrow T_{og}$
      $mode \leftarrow GPU$
  else
      $T_o \leftarrow T_{oc}$
      $mode \leftarrow CPU$
  end if
  $k \leftarrow \frac{T_c + T_s - T_o}{T_i}$
  $k_{min} \leftarrow \left\lceil \frac{T_o}{p * T_i} \right\rceil$
  $k \leftarrow max(k, k_{min})$
Output: $k, mode$
```

The algorithm provides two outputs; 1) the checkpointing frequency k which is the number of iterations elapsed between every checkpoint, and 2) the `snapshot()` *mode* (CPU or GPU-based). The algorithm first determines the snapshot mode based on available free GPU memory; if there is enough space to snapshot the model state in GPU memory, then the mode is set to GPU, else the preferred mode is set to CPU-based snapshotting. Based on the chosen mode, the algorithm estimates the overhead in the critical path incurred after pipelining checkpointing and compute in a tightly coupled manner as described earlier (§4.3.1). It then determines the number of iterations required to amortize this overhead such that the total runtime overhead incurred is below the threshold p. For example, consider the cost of a checkpoint operation and the duration of an iteration are both 1 time unit. If the threshold on runtime overhead is set to 5%, then CheckFreq chooses to checkpoint every 20 iterations.

Adaptive rate tuning. A static, profile-based frequency determination works well when the training environment of the model remains unchanged throughout the runtime of the job. However, in practice, the checkpoint cost estimated by the online profiler can deviate, resulting in higher than estimated runtime overheads. For instance, a job could face write interference by concurrently running jobs sharing storage for read/write, which affects the time to write a checkpoint.

Therefore, CheckFreq uses an adaptive rate tuning technique to perform feedback-driven frequency changes. CheckFreq's iterator monitors the runtime of the job and the actual cost of checkpointing during runtime (after the initial frequency determination). If the observed runtime exceeds the desired overhead, then these values are used to recalculate the checkpointing frequency. The idea is to ensure that the overall runtime overhead does not exceed the threshold p.

4.4 Implementation

We implement CheckFreq as a pluggable module for PyTorch. The data iterator of CheckFreq is implemented on top of the state-of-the-art data pipeline DALI for PyTorch. CheckFreq can be used as a drop-in replacement to the existing data loader in PyTorch.

CheckFreq determines the initial checkpointing frequency by profiling the first 1% of the iterations in the first epoch, or the first 50 iterations, whichever is the minimum. Therefore, no checkpointing is performed during this initial phase, which is a very small fraction of the total runtime. Additionally, we cache the profiled metrics and the determined policy on persistent storage so that profiling can be skipped when the job resumes after a crash.

CheckFreq internally uses `torch.save()`, followed by a `fsync()` to perform `persist()`, and thus guarantees persistence. To eliminate chances of data corruption, CheckFreq always writes checkpoints to a new file. However, to keep space utilization bounded, CheckFreq only maintains two checkpoints on disk at any given time; one completed checkpoint and the other in-flight. Additionally, checkpoints performed at epoch boundary are preserved (can be turned off by the user). CheckFreq wraps the weight update step in the optimizer with a semaphore that waits on the ongoing `snapshot()` to ensure that a copy of the model state is completed before it is updated by the next iteration.

5 Evaluation

In this section we use a number of microbenchmarks and end-to-end training to accuracy with interruptions to evaluate the efficacy of CheckFreq with respect to the current epoch-based checkpointing scheme across a variety of DNNs. Our evaluation seeks to answer the following questions.

- Can CheckFreq's iterator make iteration-level checkpointing feasible without affecting the accuracy? (§5.2)
- Does CheckFreq's 2-phase checkpoint mechanism reduce checkpoint stalls compared to the existing synchronous strategy? (§5.3)
- Can CheckFreq checkpoint more frequently than epoch-based checkpointing, while incurring low runtime overhead? (§5.4)
- Does CheckFreq reduce the recovery cost when DNN training is interrupted? (§5.5)
- What is the end-to-end benefit of training to accuracy with CheckFreq in the presence of job interruptions in a real preemptive training environment? (§5.6)

5.1 Experimental setup

We evaluate the efficacy of CheckFreq against the state-of-the-art epoch-based checkpointing in PyTorch using the state-of-the-art data pipeline DALI [4].

Servers. We evaluate CheckFreq on two generations of GPU; a Volta V100 GPU with a 1.8TB SSD for persistent storage, and a Pascal 1080Ti GPU with a 1.8TB HDD for persistent storage as shown in Table 2. Both these servers have 8 GPUs, 24 CPU cores and 500GB of DRAM. Both servers run 64-bit Ubuntu 16.04 with CUDA toolkit 10.0 and PyTorch 1.1.0.

	GPU Type	GPU Mem(GB)	CPU Mem(GB)	Storage Media
`Conf-Pascal`	1080Ti	11	500	HDD
`Conf-Volta`	V100	32	500	SSD

Table 2: **Server configurations**. We use two ML server SKUs; each with 24 CPU cores, 500GB DRAM, and 8 GPUs

Models. We use 7 DNNs in our evaluation. ResNet18 [20], ResNet50 [20], ResNext101 [48], DenseNet121 [21], VGG16 [41], InceptionV3 [42] all on Imagenet-1k dataset [39], and Bert-Large pretraining [13] on Wikipedia & BookCorpus dataset [49]. For each model, we use the default minibatch size reported in the literature for these models.

Baseline. We use the epoch-boundary checkpointing as the baseline for all the models except BERT. BERT trains in units of iterations; therefore we use the default checkpointing interval of 200 iterations as the baseline [8]. To perform persistent and correct checkpoints, we explicitly flush the checkpoint file after the checkpoint operation returns.

5.2 Accuracy implications

We first show the need for resumable data iterator to make fine grained iteration-level checkpointing feasible. Using the existing state-of-the-art data iterators to perform iteration-level checkpointing results in violation of the DNN data invariant as described in (§4.3.1). To demonstrate this, we perform the following experiment. We train a ResNet18 job for 70 epochs or to a target accuracy of 69.5% (whichever is earliest) in three different scenarios;

- **No interrupt**. This is the normal training scenario where the job is not interrupted until its completion. There is no checkpointing performed here.
- **Baseline-interrupt**. This scenario uses the existing DALI iterator (same with the native PyTorch iterator) to perform checkpoints at the iteration right before the job is interrupted. We interrupt the job once very 7 minutes (approx every two epochs). This corresponds to commonly used round durations in preemptive schedulers [18, 26, 32, 47].
- **CheckFreq-interrupt**. This setting uses the CheckFreq data iterator that is capable of performing a light-weight checkpoint of iterator state and correctly resuming it. We checkpoint, interrupt, and resume the job exactly as described in the prior setting.

We plot the Top-1 validation accuracy against cumulative training time. Figure 6 shows that it is not possible to perform iteration-level checkpointing using existing iterator, without affecting the model accuracy. This is because, the model state

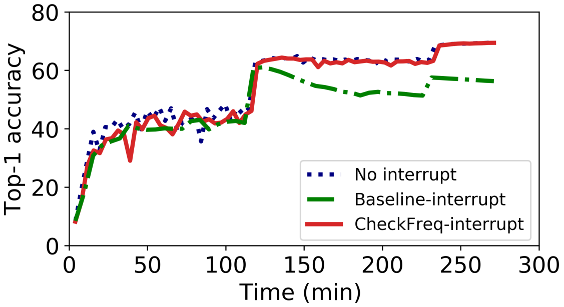


Figure 6: **Impact of resumable data iterator on accuracy**. Performing iteration-level checkpointing with baseline non-resumable data iterator violates the data invariant, results in significant loss of accuracy if job is interrupted. However, CheckFreq's iterator does not affect the final accuracy.

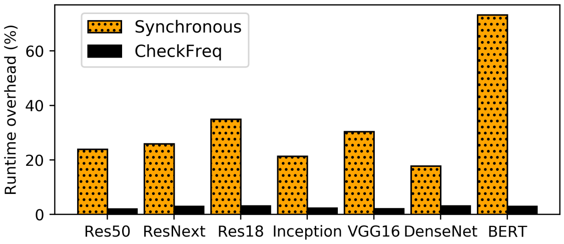


Figure 7: **Runtime overhead for various models**. At a frequency chosen by CheckFreq, synchronous checkpointing incurs upto 70% overhead while CheckFreq's pipelined checkpointing reduces runtime overhead to under 3.5%

is checkpointed at iteration boundaries, but the data loader state is lost. However, with CheckFreq's iterator, the model reaches the target accuracy in the almost the same time as the setting where the job ran without any interruption.

Storage overhead. Checkpointing data iterator state does not have a significant space overhead; it requires persisting two integers - epoch and iteration number, that take up a few bytes on disk. CheckFreq thus provides light-weight, resumable data iterators that do not affect the accuracy of DNNs.

5.3 Performance of checkpointing mechanism

We now evaluate the performance of the two-phase checkpointing strategy of CheckFreq, and compare it against the synchronous strategy. We further provide a split of benefits due to pipelining `persist()` and `snapshot()` operations.

5.3.1 Checkpoint stalls

Figure 7 shows the runtime overhead incurred due to checkpoint stalls with CheckFreq and the baseline checkpointing mechanism while checkpointing at a frequency chosen for that model by CheckFreq on `Conf-Pascal`. The frequency varies across models, but is kept constant for CheckFreq and baseline for a given model. While CheckFreq is able to bound the runtime overheads to about 3.5%, the baseline incurs 17 – 73% runtime overhead due to frequent checkpointing. The reduction in runtime overhead is due to the two-phase checkpointing and pipelining it with computation.

	Checkpoint stall (seconds)		
	Synchronous	*IO pipelining*	*CheckFreq*
`Conf-Volta`	3.6	1.5	0.3
`Conf-Pascal`	10.7	1.3	0.07

Table 3: **Breakdown of benefits**. This table shows the split of checkpoint stall incurred in critical path for VGG16 on two different hardwares

Model	Res18	Res50	ResNext	VGG16	BERT
Freq	147	125	238	83	100
Size(MB)	90	195	482	1055	5000

Table 4: **Checkpoint frequency**. This table shows the number of checkpoints per epoch and the size of each checkpoint

5.3.2 Breakdown of benefits

To understand how much each phase of the checkpointing mechanism contributes to the reduction of checkpoint stalls, we train VGG16 on the two servers using identical batch size of 64 that is the maximum that can fit on `Conf-Pascal`. Checkpointing is performed at a frequency chosen independently for the two servers. We evaluate three settings in Table 3; 1) The baseline synchronous mode, 2) CheckFreq with only `persist()` pipelining (indicated by IO pipelining) and `snapshot()` performed synchronously, 3) CheckFreq with both `persist()` and `snapshot()` pipelining.

On both hardware, CheckFreq is able to significantly reduce the checkpoint cost by $5-18\times$ by pipelining both phases of checkpointing with compute as compared to only pipelining `persist()`. On `Conf-Pascal`, the benefit due to pipelining `persist()` is prominent due to the slower storage device. On `Conf-Volta` with fast storage, the CPU cost of `snapshot()` and the storage cost of `persist()` contribute equally to the checkpointing cost. Therefore, pipelining `snapshot()` with compute provides significant speedup.

5.4 Checkpointing policy

We compare the checkpointing frequency determined by CheckFreq for a threshold overhead p of 3.5%. Table 4 shows the number of checkpoints performed per epoch for various models along with per-checkpoint size when performing distributed data parallel training across across 8 GPUs on `Conf-Pascal`. There are two main takeaways here. First, the checkpointing frequency varies with model; therefore frequency selection must take into account the model characteristics. Second, CheckFreq is able to perform $83-278\times$ more frequent checkpointing when compared to that performed at epoch boundaries, while incurring $\leq 3.5\%$ overhead. On `Conf-Volta`, CheckFreq resulted in $25-100\times$ more frequent checkpointing than the epoch-based policy. More frequent checkpoints directly translate to faster recovery times which we evaluate in Section 5.5.

Adaptive tuning of frequency. To demonstrate the importance of adaptive frequency tuning, we perform the follow-

Setting	Isolated	Static	Adaptive
Overhead	5%	35%	5%
Frequency (# iterations)	14	14	19

Table 5: **Adpative frequency tuning**. Adaptive frequency tuning is able to dynamically adjust checkpointing frequency to maintain the same overhead as if the job is run in isolation.

Model	*Recovery (seconds)*		*Recovery (seconds)*	
	Baseline	*CF*	*Baseline*	*CF*
ResNet18	840	5	180	3
ResNet50	2100	24	540	8
VGG16	5700	25	1320	31
ResNext101	7080	32	1680	14
DenseNet121	2340	7	600	4
Inceptionv3	3000	27	780	42
BERT	4920	85	4500	43

(a) 1 GPU (V100) (b) 8 GPU (1080Ti)

Table 6: **Average recovery time (CF - CheckFreq)**.

ing experiment. We run a VGG16 training job on a single GPU (Job-A), allowing it to checkpoint at an initial frequency chosen by CheckFreq (with an overhead of 5%). After 100 iterations have elapsed, we trigger another VGG16 job on a different GPU on the same machine (Job-B), so that the two jobs contend for storage bandwidth to write checkpoints. We measure the runtime for 500 iterations of Job-A with and without adaptive frequency tuning. The results are as shown in Table 5. When Job-A runs in isolation, it incurs an overhead of 5% while checkpointing every 14 iterations. However, when Job-B is introduced after 100 iterations of Job-A, if there is no adaptation across the two jobs, the checkpointing frequency is statically fixed to 14 iterations and the runtime overhead for Job-A increases to 35% (indicated as static in Table 5). This is because, the jobs compete for storage bandwidth, increasing checkpoint cost. In contrast, CheckFreq's adaptive rate tuning dynamically adjusts the checkpointing frequency and keeps the overhead bounded at 5%.

5.5 Recovery time

To understand the benefits of using CheckFreq in the presence of job interruptions, we evaluate the recovery time with the epoch-based checkpointing and CheckFreq. With epoch-based checkpointing, irrespective of when during the epoch the job is interrupted, the job rolls back to the previously completed epoch. Therefore, in the best case, if a failure occurs immediately after the finish of an epoch, then the recovery time is the same as CheckFreq. However, on average, half an epoch's worth of work can be lost if the job is interrupted in the middle of an epoch. And in the worst case, the entire epoch must be redone if the job fails just before the completion of an epoch. For the seven different models, we compare the average case recovery time in two distinct scenarios; 1) a single-GPU training job on `Conf-Volta` in Table 6a and 2) a 8 GPU data-parallel job on `Conf-Pascal` in Table 6b.

As can be seen, CheckFreq is able to reduce recovery time from several minutes (and hours) to just a few seconds, all while incurring less than 3.5% runtime overhead. For instance, when training ResNext101 on a V100 GPU, on average, CheckFreq reduces the recovery time from 2 hours to 32 seconds on average.

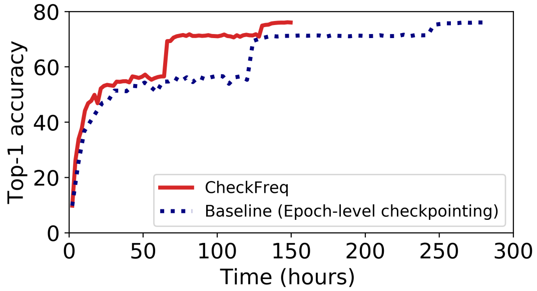


Figure 8: **End-to-end training**. We train Resnet50 using a `Conf-Pascal` GPU with interruptions every 5 hours. CheckFreq trains to state-of-the-art accuracy (76.1%) 2× faster than epoch-based checkpointing by reducing recovery time.

5.6 End-to-end training

We evaluate the end-to-end benefit of training with CheckFreq by simulating a preemptive cluster scenario. We consider a cluster with a premeptive scheduler similar to the one in large production clusters like Philly [2,22]. We consider an average preemption interval of 5 hours. Figure 8 plots the total training duration against top-1 validation accuracy for the epoch-based baseline checkpointing strategy and CheckFreq for training ResNet50 using a GPU on `Conf-Pascal` to state-of-the-art accuracy. CheckFreq results in 2× faster training by reducing recovery time from 1.9 hours to under a minute for every interruption. A similar experiment on `Conf-Volta` resulted in 1.6× faster training time to accuracy for ResNext101.

6 Discussion

Applicability to distributed cluster training. CheckFreq currently works with the distributed data parallel (DDP) mode, where only one GPU per node (rank 0) is responsible for checkpointing. While we show results for single- and multi-GPU training, extending it to multi-node settings is straightforward; checkpointing in multi-GPU and multi-node settings is the same for DDP in frameworks such as PyTorch. Model weights are synchronized across different workers (same node or in the distributed cluster) typically every iteration, or accumulated over a few tens of iterations before synchronizing; therefore each node sees the same version of weights at these synchronization points. Hence, one instance of CheckFreq runs on each node, and persists an identical checkpoint for local recovery at synchronization boundaries. Since each node persists checkpoints independently, and in parallel, there is no additional synchronization overhead for checkpointing.

Generality. CheckFreq focuses on optimizing checkpointing, which is by far the predominant way in which DNN training jobs recover from failures. While our paper focuses on data parallel training, prior work in model or pipeline parallelism, also rely on checkpointing. Using CheckFreq, checkpointing at minibatch boundaries (every n iterations), each pipeline stage only persists a subset of parameters and optimizer state hosted by that worker. CheckFreq also enables checkpointing within minibatch boundaries during pipeline parallel training (every m microbatches), as CheckFreq's iterator controls the introduction of each microbatch into the pipeline. Checkpointing at the microbatch granularity requires storing additional model state – specifically accumulated weight gradients at every stage in addition to parameter and optimizer state. We leave it to future work to integrate CheckFreq's implementation into frameworks supporting pipeline parallelism.

While we implement CheckFreq in PyTorch, we can extend it to other frameworks like TF and MxNet by wrapping the framework-specific APIs into those exposed by CheckFreq.

7 Related Work

Asynchronous DNN checkpointing. While recent work like DeepFreeze [33] that perform asynchronous DNN checkpointing employ techniques similar to CheckFreq for IO pipelining, it only considers CPU clusters. It does not consider the cost of snapshotting the model state in memory when trained using state-of-the-art GPUs. Our work shows that on modern ML optimized servers, the cost of snapshotting the model state (copying from GPU to CPU) is significant, demonstrating how to pipeline this transfer with compute, and use spare GPU capabilities to enable fast snapshotting.

Furthermore, DeepFreeze requires manual intervention to tune the checkpointing frequency for a given model, hardware and training environment while CheckFreq masks these complexities from the user and analytically identifies the best parameters for checkpointing. Unlike DeepFreeze that uses a static checkpointing frequency, CheckFreq is also beneficial in shared cluster settings, as it adapts the checkpointing frequency based on memory and storage interference due to other jobs to minimize checkpoint stalls.

Asynchronous checkpointing in HPC. Prior work in HPC [34, 40, 45] uses asynchronous checkpointing to mask the IO latency. A key challenge that differentiates DNN checkpointing from traditional HPC ones is that, performing a synchronous in-memory copy of the model state from GPU to CPU is expensive due to the increasingly fast compute capabilities of the GPU. CheckFreq exploits the DNN learning structure to carefully pipeline even the in-memory snapshot with computation to perform correct, consistent checkpointing. Moreover, CheckFreq further reduces the latency of checkpointing by utilizing spare GPU memory and compute capabilities when possible to perform fast snapshots.

Checkpoint interval estimation in HPC. Prior work [12, 14, 15] determine checkpointing interval for large scale HPC applications based on failure distributions observed in the system. CheckFreq does this in a DNN-aware fashion by exploiting the deterministic, repetitive structure of DNN training to systematically profile resource utilization at runtime.

Adaptive checkpointing. The idea of using adaptation for fault management has been used in HPC applications [25] to decide when to checkpoint, based on a failure prediction module. CheckFreq introduces adaptivity in DNN checkpointing frequency. It identifies and dynamically adapts the checkpointing frequency, based on the characteristics of the model being trained, system hardware, and interference due to other jobs.

TensorFlow Checkpoint Manager. TF checkpoint manager [43] allows checkpointing at a user-given time interval, and supports persisting iterator state. However, it has three shortcomings. First, the checkpointing frequency is decided in an ad-hoc fashion by the user; this introduces large checkpoint stalls if not chosen carefully. Second, it cannot checkpoint the iterator state if random data transformation is involved; this is common for most image based models [44]. Finally, even in cases where it can persist iterator state, TF writes the entire operator graph to storage along with prefetched items resulting in large checkpoint size. CheckFreq addresses these challenges by automatically adapting the checkpointing frequency and using a light-weight, resumable data iterator.

Framework-transparent checkpointing. Transparent checkpointing techniques such as CRIU [1] can backup entire VM state for fault-tolerance; however they do not checkpoint GPU or accelerator state. Even if they were to capture entire device state, device state alone is an order of magnitude larger than the model state captured at iteration boundaries, making frequent CRIU checkpoints impractical. Thus, in this work, we focus on the dominant approach to DNN fault-tolerance - framework-assisted checkpointing of model state.

8 Conclusion

This paper presents CheckFreq, an automatic, fine-grained checkpointing framework for DNN training. CheckFreq achieves consistent, low-cost checkpoints at iteration level using a resumable data iterator, a pipelined two-phase checkpointing mechanism, and automatic determination and tuning of checkpointing frequency. When the job is interrupted, CheckFreq reduces recovery time for popular DNNs from hours to seconds, while incurring low runtime overhead.

Acknowledgements

This work was done during an internship at Microsoft Research as part of Project Fiddle. We thank our shepherd Mehul Shah, the anonymous FAST reviewers, members of the UT SaSLab, fellow Project Fiddle interns Youjie Li, Kshiteej Mahajan, Andrew Or, and many of our MSR colleagues for their invaluable feedback that made this work better. We sincerely thank MSR Labs for their generous support in procuring the many resources required for this work. This work was supported by NSF CAREER #1751277 and donations from VMware, Google, and Facebook.

References

[1] CRIU checkpointing. https://criu.org/Main_Page.

[2] Microsoft Philly Traces. https://github.com/msr-fiddle/philly-traces.

[3] Training a Champion: Building Deep Neural Nets for Big Data Analytics. https://www.kdnuggets.com/training-a-champion-building-deep-neural-nets-for-big-data-analytics.html/.

[4] NVIDIA DALI. https://github.com/NVIDIA/DALI, 2018.

[5] PyTorch. https://github.com/pytorch/pytorch, 2019.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016.

[7] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675*, 2016.

[8] NVIDIA AI. BERT Meets GPUs. hhttps://medium.com/future-vision/bert-meets-gpus-403d3fbed848.

[9] Amazon. Amazon EC2 spot instances. https://aws.amazon.com/ec2/spot/?cards.sort-by=item.additionalFields.startDateTime&cards.sort-order=asc.

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[12] John T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[14] Sheng Di, Mohamed-Slim Bouguerra, Leonardo Arturo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 1181–1190. IEEE Computer Society, 2014.

[15] Sheng Di, Mohamed-Slim Bouguerra, Leonardo Arturo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 1181–1190. IEEE Computer Society, 2014.

[16] Google. Preemptible VM instances. https://cloud.google.com/compute/docs/instances/preemptible#preemptible_with_gpu.

[17] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[18] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 485–500. USENIX Association, 2019.

[19] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis,*

SC 2017, Denver, CO, USA, November 12 - 17, 2017, pages 44:1–44:12. ACM, 2017.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[21] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 2261–2269. IEEE Computer Society, 2017.

[22] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[24] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, et al. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *arXiv preprint arXiv:1811.00982*, 2018.

[25] Zhiling Lan and Yawei Li. Adaptive fault management of parallel applications for high-performance computing. *IEEE Trans. Computers*, 57(12):1647–1660, 2008.

[26] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 289–304. USENIX Association, 2020.

[27] Catello Di Martino, Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 610–621. IEEE Computer Society, 2014.

[28] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. *Neurocomputing*, 337:46–57, 2019.

[29] Microsoft. Use low priority VMs. https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms.

[30] MLPerf. MLPerf Training Results v0.7. https://github.com/mlperf/training_results_v0.7.

[31] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Analysis and exploitation of dynamic pricing in the public cloud for ml training. *DISPA*, 2020.

[32] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. *arXiv preprint arXiv:2008.09213*, 2020.

[33] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*, pages 172–181. IEEE, 2020.

[34] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. Veloc: Towards high performance adaptive asynchronous checkpointing at large scale. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 911–920. IEEE, 2019.

[35] NVIDIA. ResNext101 Training. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Classification/ConvNets/resnext101-32x4d.

[36] OpenAI. GPT-3 Checkpoint. https://github.com/openai/gpt-3/issues/1.

[37] 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 *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 433–448. USENIX Association, 2014.

[38] PyTorch. PyTorch Training Examples. https://github.com/pytorch/examples/tree/master/imagenet.

[39] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[40] Faisal Shahzad, Markus Wittmann, Thomas Zeiser, Georg Hager, and Gerhard Wellein. An evaluation of different I/O techniques for checkpoint/restart. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*, pages 1708–1716. IEEE, 2013.

[41] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[42] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2818–2826. IEEE Computer Society, 2016.

[43] TensorFlow. Tensorflow checkpoint manager. https://www.tensorflow.org/api_docs/python/tf/train/CheckpointManager.

[44] TensorFlow. Tensorflow iterator checkpointing. https://www.tensorflow.org/guide/data#iterator_checkpointing.

[45] Devesh Tiwari, Saurabh Gupta, and Sudharshan S. Vazhkudai. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 25–36. IEEE Computer Society, 2014.

[46] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[47] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.

[48] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 5987–5995. IEEE Computer Society, 2017.

[49] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.

Facebook's Tectonic Filesystem: Efficiency from Exascale

Satadru Pan[1], Theano Stavrinos[1,2], Yunqiao Zhang[1], Atul Sikaria[1], Pavel Zakharov[1], Abhinav Sharma[1], Shiva Shankar P[1], Mike Shuey[1], Richard Wareing[1], Monika Gangapuram[1], Guanglei Cao[1], Christian Preseau[1], Pratap Singh[1], Kestutis Patiejunas[1], JR Tipton[1], Ethan Katz-Bassett[3], and Wyatt Lloyd[2]

[1]*Facebook, Inc.,* [2]*Princeton University,* [3]*Columbia University*

Abstract

Tectonic is Facebook's exabyte-scale distributed filesystem. Tectonic consolidates large tenants that previously used service-specific systems into general multitenant filesystem instances that achieve performance comparable to the specialized systems. The exabyte-scale consolidated instances enable better resource utilization, simpler services, and less operational complexity than our previous approach. This paper describes Tectonic's design, explaining how it achieves scalability, supports multitenancy, and allows tenants to specialize operations to optimize for diverse workloads. The paper also presents insights from designing, deploying, and operating Tectonic.

1 Introduction

Tectonic is Facebook's distributed filesystem. It currently serves around ten tenants, including blob storage and data warehouse, both of which store exabytes of data. Prior to Tectonic, Facebook's storage infrastructure consisted of a constellation of smaller, specialized storage systems. Blob storage was spread across Haystack [11] and f4 [34]. Data warehouse was spread across many HDFS instances [15].

The constellation approach was operationally complex, requiring many different systems to be developed, optimized, and managed. It was also inefficient, stranding resources in the specialized storage systems that could have been reallocated for other parts of the storage workload.

A Tectonic cluster scales to exabytes such that a single cluster can span an entire datacenter. The multi-exabyte capacity of a Tectonic cluster makes it possible to host several large tenants like blob storage and data warehouse on the same cluster, with each supporting hundreds of applications in turn. As an exabyte-scale multitenant filesystem, Tectonic provides operational simplicity and resource efficiency compared to federation-based storage architectures [8, 17], which assemble smaller petabyte-scale clusters.

Tectonic simplifies operations because it is a single system to develop, optimize, and manage for diverse storage needs. It is resource-efficient because it allows resource sharing among all cluster tenants. For instance, Haystack was the storage system specialized for new blobs; it bottlenecked on hard disk IO per second (IOPS) but had spare disk capacity. f4, which stored older blobs, bottlenecked on disk capacity but had spare IO capacity. Tectonic requires fewer disks to support the same workloads through consolidation and resource sharing.

In building Tectonic, we confronted three high-level challenges: scaling to exabyte-scale, providing performance isolation between tenants, and enabling tenant-specific optimizations. Exabyte-scale clusters are important for operational simplicity and resource sharing. Performance isolation and tenant-specific optimizations help Tectonic match the performance of specialized storage systems.

To scale metadata, Tectonic disaggregates the filesystem metadata into independently-scalable layers, similar to ADLS [42]. Unlike ADLS, Tectonic hash-partitions each metadata layer rather than using range partitioning. Hash partitioning effectively avoids hotspots in the metadata layer. Combined with Tectonic's highly scalable chunk storage layer, disaggregated metadata allows Tectonic to scale to exabytes of storage and billions of files.

Tectonic simplifies performance isolation by solving the isolation problem for groups of applications in each tenant with similar traffic patterns and latency requirements. Instead of managing resources among hundreds of applications, Tectonic only manages resources among tens of traffic groups.

Tectonic uses tenant-specific optimizations to match the performance of specialized storage systems. These optimizations are enabled by a client-driven microservice architecture that includes a rich set of client-side configurations for controlling how tenants interact with Tectonic. Data warehouse, for instance, uses Reed-Solomon (RS)-encoded writes to improve space, IO, and networking efficiency for its large writes. Blob storage, in contrast, uses a replicated quorum append protocol to minimize latency for its small writes and later RS-encodes them for space efficiency.

Tectonic has been hosting blob storage and data warehouse in single-tenant clusters for several years, completely replacing Haystack, f4, and HDFS. Multitenant clusters are being methodically rolled out to ensure reliability and avoid performance regressions.

Adopting Tectonic has yielded many operational and efficiency improvements. Moving data warehouse from HDFS onto Tectonic reduced the number of data warehouse clusters by 10×, simplifying operations from managing fewer clusters. Consolidating blob storage and data warehouse into multitenant clusters helped data warehouse handle traffic spikes with spare blob storage IO capacity. Tectonic manages these efficiency improvements while providing comparable or better performance than the previous specialized storage systems.

2 Facebook's Previous Storage Infrastructure

Before Tectonic, each major storage tenant stored its data in one or more specialized storage systems. We focus here on two large tenants, blob storage and data warehouse. We discuss each tenant's performance requirements, their prior storage systems, and why they were inefficient.

2.1 Blob Storage

Blob storage stores and serves **b**inary **l**arge **ob**jects. These may be media from Facebook apps (photos, videos, or message attachments) or data from internal applications (core dumps, bug reports). Blobs are immutable and opaque. They vary in size from several kilobytes for small photos to several megabytes for high-definition video chunks [34]. Blob storage expects low-latency reads and writes as blobs are often on path for interactive Facebook applications [29].

Haystack and f4. Before Tectonic, blob storage consisted of two specialized systems, Haystack and f4. Haystack handled "hot" blobs with a high access frequency [11]. It stored data in replicated form for durability and fast reads and writes. As Haystack blobs aged and were accessed less frequently, they were moved to f4, the "warm" blob storage [34]. f4 stored data in RS-encoded form [43], which is more space-efficient but has lower throughput because each blob is directly accessible from two disks instead of three in Haystack. f4's lower throughput was acceptable because of its lower request rate.

However, separating hot and warm blobs resulted in poor resource utilization, a problem exacerbated by hardware and blob storage usage trends. Haystack's ideal effective replication factor was 3.6× (i.e., each logical byte is replicated 3×, with an additional 1.2× overhead for RAID-6 storage [19]). However, because IOPS per hard drive has remained steady as drive density has increased, IOPS per terabyte of storage capacity has declined over time.

As a result, Haystack became IOPS-bound; extra hard drives had to be provisioned to handle the high IOPS load of hot blobs. The spare disk capacity resulted in Haystack's effective replication factor increasing to 5.3×. In contrast, f4 had an effective replication factor of 2.8× (using RS(10,4) encoding in two different datacenters). Furthermore, blob storage usage shifted to more ephemeral media that was stored in Haystack but deleted before moving to f4. As a result, an increasing share of the total blob data was stored at Haystack's high effective replication factor.

Finally, since Haystack and f4 were separate systems, each *stranded* resources that could not be shared with other systems. Haystack overprovisioned storage to accommodate peak IOPS, whereas f4 had an abundance of IOPS from storing a large volume of less frequently-accessed data. Moving blob storage to Tectonic harvested these stranded resources and resulted in an effective replication factor of ~2.8×.

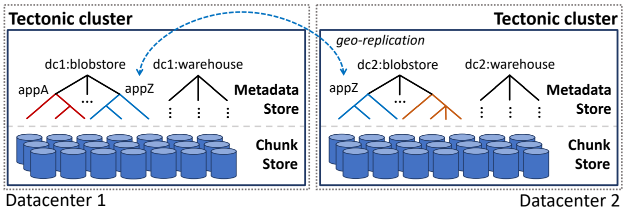


Figure 1: Tectonic provides durable, fault-tolerant storage inside a datacenter. Each tenant has one or more separate namespaces. Tenants implement geo-replication.

2.2 Data Warehouse

Data warehouse provides storage for data analytics. Data warehouse applications store objects like massive map-reduce tables, snapshots of the social graph, and AI training data and models. Multiple compute engines, including Presto [3], Spark [10], and AI training pipelines [4] access this data, process it, and store derived data. Warehouse data is partitioned into datasets that store related data for different product groups like Search, Newsfeed, and Ads.

Data warehouse storage prioritizes read and write throughput over latency, since data warehouse applications often batch-process data. Data warehouse workloads tend to issue larger reads and writes than blob storage, with reads averaging multiple megabytes and writes averaging tens of megabytes.

HDFS for data warehouse storage. Before Tectonic, data warehouse used the Hadoop Distributed File System (HDFS) [15, 50]. However, HDFS clusters are limited in size because they use a single machine to store and serve metadata.

As a result, we needed tens of HDFS clusters per datacenter to store analytics data. This was operationally inefficient; every service had to be aware of data placement and movement among clusters. Single data warehouse datasets are often large enough to exceed a single HDFS cluster's capacity. This complicated compute engine logic, since related data was often split among separate clusters.

Finally, distributing datasets among the HDFS clusters created a two-dimensional bin-packing problem. The packing of datasets into clusters had to respect each cluster's capacity constraints and available throughput. Tectonic's exabyte scale eliminated the bin-packing and dataset-splitting problems.

3 Architecture and Implementation

This section describes the Tectonic architecture and implementation, focusing on how Tectonic achieves exabyte-scale single clusters with its scalable chunk and metadata stores.

3.1 Tectonic: A Bird's-Eye View

A *cluster* is the top-level Tectonic deployment unit. Tectonic clusters are datacenter-local, providing durable storage that is resilient to host, rack, and power domain failures. Tenants can build geo-replication on top of Tectonic for protection against datacenter failures (Figure 1).

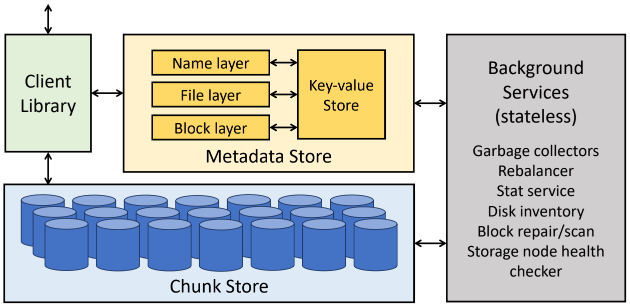


Figure 2: Tectonic architecture. Arrows indicate network calls. Tectonic stores filesystem metadata in a key-value store. Apart from the Chunk and Metadata Stores, all components are stateless.

A Tectonic cluster is made up of storage nodes, metadata nodes, and stateless nodes for background operations. The Client Library orchestrates remote procedure calls to the metadata and storage nodes. Tectonic clusters can be very large: a single cluster can serve the storage needs of all tenants in a single datacenter.

Tectonic clusters are *multitenant*, supporting around ten tenants on the same storage fabric (§4). *Tenants* are distributed systems that will never share data with one another; tenants include blob storage and data warehouse. These tenants in turn serve hundreds of *applications*, including Newsfeed, Search, Ads, and internal services, each with varying traffic patterns and performance requirements.

Tectonic clusters support any number of arbitrarily-sized *namespaces*, or filesystem directory hierarchies, on the same storage and metadata components. Each tenant in a cluster typically owns one namespace. Namespace sizes are limited only by the size of the cluster.

Applications interact with Tectonic through a hierarchical filesystem API with append-only semantics, similar to HDFS [15]. Unlike HDFS, Tectonic APIs are configurable at runtime, rather than being pre-configured on a per-cluster or per-tenant basis. Tectonic tenants leverage this flexibility to match the performance of specialized storage systems (§4).

Tectonic components. Figure 2 shows the major components of Tectonic. The foundation of a Tectonic cluster is the *Chunk Store* (§3.2), a fleet of storage nodes which store and access data chunks on hard drives.

On top of the Chunk Store is the *Metadata Store* (§3.3), which consists of a scalable key-value store and stateless metadata services that construct the filesystem logic over the key-value store. Their scalability enables Tectonic to store exabytes of data.

Tectonic is a client-driven microservices-based system, a design that enables tenant-specific optimizations. The Chunk and Metadata Stores each run independent services to handle read and write requests for data and metadata. These services are orchestrated by the *Client Library* (§3.4); the library converts clients' filesystem API calls into RPCs to Chunk and Metadata Store services.

Finally, each cluster runs stateless background services to maintain cluster consistency and fault tolerance (§3.5).

3.2 Chunk Store: Exabyte-Scale Storage

The Chunk Store is a flat, distributed object store for *chunks*, the unit of data storage in Tectonic. Chunks make up blocks, which in turn make up Tectonic files.

The Chunk Store has two features that contribute to Tectonic's scalability and ability to support multiple tenants. First, the Chunk Store is flat; the number of chunks stored grows linearly with the number of storage nodes. As a result, the Chunk Store can scale to store exabytes of data. Second, it is oblivious to higher-level abstractions like blocks or files; these abstractions are constructed by the Client Library using the Metadata Store. Separating data storage from filesystem abstractions simplifies the problem of supporting good performance for a diversity of tenants on one storage cluster (§5). This separation means reading to and writing from storage nodes can be specialized to tenants' performance needs without changing filesystem management.

Storing chunks efficiently. Individual chunks are stored as files on a cluster's storage nodes, which each run a local instance of XFS [26]. Storage nodes expose core IO APIs to get, put, append to, and delete chunks, along with APIs for listing chunks and scanning chunks. Storage nodes are responsible for ensuring that their own local resources are shared fairly among Tectonic tenants (§4).

Each storage node has 36 hard drives for storing chunks [5]. Each node also has a 1 TB SSD, used for storing XFS metadata and caching hot chunks. Storage nodes run a version of XFS that stores local XFS metadata on flash [47]. This is particularly helpful for blob storage, where new blobs are written as appends, updating the chunk size. The SSD hot chunk cache is managed by a cache library which is flash endurance-aware [13].

Blocks as the unit of durable storage. In Tectonic, blocks are a logical unit that hides the complexity of raw data storage and durability from the upper layers of the filesystem. To the upper layers, a block is an array of bytes. In reality, blocks are composed of chunks which together provide block durability.

Tectonic provides per-block durability to allow tenants to tune the tradeoff between storage capacity, fault tolerance, and performance. Blocks are either Reed-Solomon encoded [43] or replicated for durability. For $\mathrm{RS}(r,k)$ encoding, the block data is split into r equal chunks (potentially by padding the data), and k parity chunks are generated from the data chunks. For replication, data chunks are the same size as the block and multiple copies are created. Chunks in a block are stored in different fault domains (e.g., different racks) for fault tolerance. Background services repair damaged or lost chunks to maintain durability (§3.5).

Layer	Key	Value	Sharded by	Mapping
Name	(`dir_id`, *subdirname*)	`subdir_info`, `subdir_id`	`dir_id`	dir → list of subdirs (expanded)
	(`dir_id`, *filename*)	`file_info`, `file_id`	`dir_id`	dir → list of files (expanded)
File	(`file_id`, `blk_id`)	`blk_info`	`file_id`	file → list of blocks (expanded)
Block	`blk_id`	list<`disk_id`>	`blk_id`	block → list of disks (i.e., chunks)
	(`disk_id`, `blk_id`)	`chunk_info`	`blk_id`	disk → list of blocks (expanded)

Table 1: Tectonic's layered metadata schema. *dirname* and *filename* are application-exposed strings. `dir_id`, `file_id`, and `block_id` are internal object references. Most mappings are expanded for efficient updating.

3.3 Metadata Store: Naming Exabytes of Data

Tectonic's Metadata Store stores the filesystem hierarchy and the mapping of blocks to chunks. The Metadata Store uses a fine-grained partitioning of filesystem metadata for operational simplicity and scalability. Filesystem metadata is first *disaggregated*, meaning the naming, file, and block layers are logically separated. Each layer is then hash partitioned (Table 1). As we describe in this section, scalability and load balancing come for free with this design. Careful handling of metadata operations preserves filesystem consistency despite the fine-grained metadata partitioning.

Storing metadata in a key-value store for scalability and operational simplicity. Tectonic delegates filesystem metadata storage to ZippyDB [6], a linearizable, fault-tolerant, sharded key-value store. The key-value store manages data at the shard granularity: all operations are scoped to a shard, and shards are the unit of replication. The key-value store nodes internally run RocksDB [23], a SSD-based single-node key-value store, to store shard replicas. Shards are replicated with Paxos [30] for fault tolerance. Any replica can serve reads, though reads that must be strongly consistent are served by the primary. The key-value store does not provide cross-shard transactions, limiting certain filesystem metadata operations.

Shards are sized so that each metadata node can host several shards. This allows shards to be redistributed in parallel to new nodes in case a node fails, reducing recovery time. It also allows granular load balancing; the key-value store will transparently move shards to control load on each node.

Filesystem metadata layers. Table 1 shows the filesystem metadata layers, what they map, and how they are sharded. The Name layer maps each directory to its sub-directories and/or files. The File layer maps file objects to a list of blocks. The Block layer maps each block to a list of disk (i.e., chunk) locations. The Block layer also contains the reverse index of disks to the blocks whose chunks are stored on that disk, used for maintenance operations. Name, File, and Block layers are hash-partitioned by directory, file, and block IDs, respectively.

As shown in Table 1, the Name and File layer and disk to block list maps are *expanded*. A key mapped to a list is expanded by storing each item in the list as a key, prefixed by the true key. For example, if directory `d1` contains files *foo* and *bar*, we store two keys (`d1`, *foo*) and (`d1`, *bar*) in `d1`'s Name shard. Expanding allows the contents of a key to be modified without reading and then writing the entire list. In a filesystem where mappings can be very large, e.g., directories may contain millions of files, expanding significantly reduces the overhead of some metadata operations such as file creation and deletion. The contents of a expanded key are listed by doing a prefix scan over keys.

Fine-grained metadata partitioning to avoid hotspots. In a filesystem, directory operations often cause hotspots in metadata stores. This is particularly true for data warehouse workloads where related data is grouped into directories; many files from the same directory may be read in a short time, resulting in repeated accesses to the directory.

Tectonic's layered metadata approach naturally avoids hotspots in directories and other layers by separating searching and listing directory contents (Name layer) from reading file data (File and Block layers). This is similar to ADLS's separation of metadata layers [42]. However, ADLS range-partitions metadata layers whereas Tectonic hash-partitions layers. Range partitioning tends to place related data on the same shard, e.g., subtrees of the directory hierarchy, making the metadata layer prone to hotspots if not carefully sharded.

We found that hash partitioning effectively load-balances metadata operations. For example, in the Name layer, the immediate directory listing of a single directory is always stored in a single shard. But listings of two subdirectories of the same directory will likely be on separate shards. In the Block layer, block locator information is hashed among shards, independent of the blocks' directory or file. Around two-thirds of metadata operations in Tectonic are served by the Block layer, but hash partitioning ensures this traffic is evenly distributed among Block layer shards.

Caching sealed object metadata to reduce read load. Metadata shards have limited available throughput, so to reduce read load, Tectonic allows blocks, files, and directories to be *sealed*. Directory sealing does not apply recursively, it only prevents adding objects in the immediate level of the directory. The contents of sealed filesystem objects cannot change; their metadata can be cached at metadata nodes and at clients without compromising consistency. The exception is the block-to-chunk mapping; chunks can migrate among disks, invalidating the Block layer cache. A stale Block layer cache can be detected during reads, triggering a cache refresh.

Providing consistent metadata operations. Tectonic relies on the key-value store's strongly-consistent operations and atomic read-modify-write in-shard transactions for strongly-consistent same-directory operations. More specifically, Tectonic guarantees read-after-write consistency for data operations (e.g., appends, reads), file and directory operations involving a single object (e.g., create, list), and move operations where the source and destination are in the same parent directory. Files in a directory reside in the directory's shard (Table 1), so metadata operations like file create, delete, and moves within a parent directory are consistent.

The key-value store does not support consistent cross-shard transactions, so Tectonic provides non-atomic cross-directory move operations. Moving a directory to another parent directory on a different shard is a two-phase process. First, we create a link from the new parent directory, and then delete the link from the previous parent. The moved directory keeps a backpointer to its parent directory to detect pending moves. This ensures only one move operation is active for a directory at a time. Similarly, cross directory file moves involve copying the file and deleting it from the source directory. The copy step creates a new file object with the underlying blocks of the source file, avoiding data movement.

In the absence of cross-shard transactions, multi-shard metadata operations on the same file must be carefully implemented to avoid race conditions. An example of such a race condition is when a file named *f1* in directory `d` is renamed to *f2*. Concurrently, a new file with the same name is created, where creates overwrite existing files with the same name. The metadata layer and shard lookup key (*shard*(`x`)) are listed for each step in parentheses.

A file rename has the following steps:

R1: get file ID `fid` for *f1* (Name, *shard*(`d`))

R2: add *f2* as an owner of *fid* (File, *shard*(`fid`))

R3: create the mapping *f2* → `fid` and delete *f1* → `fid` in an atomic transaction (Name, *shard*(`d`))

A file create with overwriting has the following steps:

C1: create new file ID `fid_new` (File, *shard*(`fid_new`))

C2: map *f1* → `fid_new`; delete *f1* → `fid` (Name, *shard*(`d`))

Interleaving the steps in these transactions may leave the filesystem in an inconsistent state. If steps C1 and C2 are executed after R1 but before R3, then R3 will erase the newly-created mapping from the create operation. Rename step R3 uses a within-shard transaction to ensure that the file object pointed to by *f1* has not been modified since R1.

3.4 Client Library

The Tectonic Client Library orchestrates the Chunk and Metadata Store services to expose a filesystem abstraction to applications, which gives applications per-operation control over how to configure reads and writes. Moreover, the Client Library executes reads and writes at the chunk granularity, the finest granularity possible in Tectonic. This gives the Client Library nearly free reign to execute operations in the most performant way possible for applications, which might have different workloads or prefer different tradeoffs (§5).

The Client Library replicates or RS-encodes data and writes chunks directly to the Chunk Store. It reads and reconstructs chunks from the Chunk Store for the application. The Client Library consults the Metadata Store to locate chunks, and updates the Metadata Store for filesystem operations.

Single-writer semantics for simple, optimizable writes. Tectonic simplifies the Client Library's orchestration by allowing a single writer per file. Single-writer semantics avoids the complexity of serializing writes to a file from multiple writers. The Client Library can instead write directly to storage nodes in parallel, allowing it to replicate chunks in parallel and to hedge writes (§5). Tenants needing multiple-writer semantics can build serialization semantics on top of Tectonic.

Tectonic enforces single-writer semantics with a write token for every file. Any time a writer wants to add a block to a file, it must include a matching token for the metadata write to succeed. A token is added in the file metadata when a process opens a file for appending, which subsequent writes must include to update file metadata. If a second process attempts to open the file, it will generate a new token and overwrite the first process's token, becoming the new, and only, writer for the file. The new writer's Client Library will seal any blocks opened by the previous writer in the open file call.

3.5 Background Services

Background services maintain consistency between metadata layers, maintain durability by repairing lost data, rebalance data across storage nodes, handle rack drains, and publish statistics about filesystem usage. Background services are layered similar to the Metadata Store, and they operate on one shard at a time. Figure 2 lists important background services.

A garbage collector between each metadata layer cleans up (acceptable) metadata inconsistencies. Metadata inconsistencies can result from failed multi-step Client Library operations. Lazy object deletion, a real-time latency optimization that marks deleted objects at delete time without actually removing them, also causes inconsistencies.

A rebalancer and a repair service work in tandem to relocate or delete chunks. The rebalancer identifies chunks that need to be moved in response to events like hardware failure, added storage capacity, and rack drains. The repair service handles the actual data movement by reconciling the chunk list to the disk-to-block map for every disk in the system. To scale horizontally, the repair service works on a per-Block layer shard, per-disk basis, enabled by the reverse index mapping disks to blocks (Table 1).

Copysets at scale. Copysets are combinations of disks that provide redundancy for the same block (e.g., a copyset for an RS(10,4)-encoded block consists of 14 disks) [20]. Having too many copysets risks data unavailability if there is an unexpected spike in disk failures. On the other hand, having too few copysets results in high reconstruction load to peer disks when one disk fails, since they share many chunks.

The Block Layer and the rebalancer service together attempt to maintain a fixed copyset count that balances unavailability and reconstruction load. They each keep in memory about one hundred consistent shuffles of all the disks in the cluster. The Block Layer forms copysets from contiguous disks in a shuffle. On a write, the Block Layer gives the Client Library a copyset from the shuffle corresponding to that block ID. The rebalancer service tries to keep the block's chunks in the copyset specified by that block's shuffle. Copysets are best-effort, since disk membership in the cluster changes constantly.

4 Multitenancy

Providing comparable performance for tenants as they move from individual, specialized storage systems to a consolidated filesystem presents two challenges. First, tenants must share resources while giving each tenant its fair share, i.e., at least the same resources it would have in a single-tenant system. Second, tenants should be able to optimize performance as in specialized systems. This section describes how Tectonic supports resource sharing with a clean design that maintains operational simplicity. Section 5 describes how Tectonic's tenant-specific optimizations allow tenants to get performance comparable to specialized storage systems.

4.1 Sharing Resources Effectively

As a shared filesystem for diverse tenants across Facebook, Tectonic needs to manage resources effectively. In particular, Tectonic needs to provide approximate (weighted) fair sharing of resources among tenants and performance isolation between tenants, while elastically shifting resources among applications to maintain high resource utilization. Tectonic also needs to distinguish latency-sensitive requests to avoid blocking them behind large requests.

Types of resources. Tectonic distinguishes two types of resources: non-ephemeral and ephemeral. Storage capacity is the *non-ephemeral* resource. It changes slowly and predictably. Most importantly, once allocated to a tenant, it cannot be given to another tenant. Storage capacity is managed at the tenant granularity. Each tenant gets a predefined capacity quota with strict isolation, i.e., there is no automatic elasticity in the space allocated to different tenants. Reconfiguring storage capacity between tenants is done manually. Reconfiguration does not cause downtime, so in case of an urgent capacity crunch, it can be done immediately. Tenants are responsible for distributing and tracking storage capacity among their applications.

Ephemeral resources are those where demand changes from moment to moment, and allocation of these resources can change in real time. Storage IOPS capacity and metadata query capacity are two ephemeral resources. Because ephemeral resource demand changes quickly, these resources need finer-grained real-time automated management to ensure they are shared fairly, tenants are isolated from one another, and resource utilization is high. For the rest of this section, we describe how Tectonic shares ephemeral resources effectively.

Distributing ephemeral resources among and within tenants. Ephemeral resource sharing is challenging in Tectonic because not only are tenants diverse, but each tenant serves many applications with varied traffic patterns and performance requirements. For example, blob storage includes production traffic from Facebook users and background garbage collection traffic. Managing ephemeral resources at the tenant granularity would be too coarse to account for the varied workloads and performance requirements within a tenant. On the other hand, because Tectonic serves hundreds of applications, managing resources at the application granularity would be too complex and resource-intensive.

Ephemeral resources are therefore managed within each tenant at the granularity of groups of applications. These application groups, called *TrafficGroups*, reduce the cardinality of the resource sharing problem, reducing the overhead of managing multitenancy. Applications in the same TrafficGroup have similar resource and latency requirements. For example, one TrafficGroup may be for applications generating background traffic while another is for applications generating production traffic. Tectonic supports around 50 TrafficGroups per cluster. Each tenant may have a different number of TrafficGroups. Tenants are responsible for choosing the appropriate TrafficGroup for each of their applications. Each TrafficGroup is in turn assigned a *TrafficClass*. A TrafficGroup's TrafficClass indicates its latency requirements and decides which requests should get spare resources. The TrafficClasses are Gold, Silver, and Bronze, corresponding to latency-sensitive, normal, and background applications. Spare resources are distributed according to TrafficClass priority within a tenant.

Tectonic uses tenants and TrafficGroups along with the notion of TrafficClass to ensure isolation and high resource utilization. That is, tenants are allocated their fair share of resources; within each tenant, resources are distributed by TrafficGroup and TrafficClass. Each tenant gets a guaranteed quota of the cluster's ephemeral resources, which is subdivided between a tenant's TrafficGroups. Each TrafficGroup gets its guaranteed resource quota, which provides isolation between tenants as well as isolation between TrafficGroups.

Any ephemeral resource surplus within a tenant is shared with its own TrafficGroups by descending TrafficClass. Any remaining surplus is given to TrafficGroups in other tenants by descending TrafficClass. This ensures spare resources are used by TrafficGroups of the same tenant first before being distributed to other tenants. When one TrafficGroup uses resources from another TrafficGroup, the resulting traffic gets the minimum TrafficClass of the two TrafficGroups. This ensures the overall ratio of traffic of different classes does not change based on resource allocation, which ensures the node can meet the latency profile of the TrafficClass.

Enforcing global resource sharing. The Client Library uses a rate limiter to achieve the aforementioned elasticity. The rate limiter uses high-performance, near-realtime distributed counters to track the demand for each tracked resource in each tenant and TrafficGroup in the last small time window. The rate limiter implements a modified leaky bucket algorithm. An incoming request increments the demand counter for the bucket. The Client Library then checks for spare capacity in its own TrafficGroup, then other TrafficGroups in the same tenant, and finally other tenants, adhering to TrafficClass priority. If the client finds spare capacity, the request is sent to the backend. Otherwise, the request is delayed or rejected depending on the request's timeout. Throttling requests at clients puts backpressure on clients before they make a potentially wasted request.

Enforcing local resource sharing. The client rate limiter ensures approximate global fair sharing and isolation. Metadata and storage nodes also need to manage resources to avoid local hotspots. Nodes provide fair sharing and isolation with a weighted round-robin (WRR) scheduler that provisionally skips a TrafficGroup's turn if it will exceed its resource quota. In addition, storage nodes need to ensure that small IO requests (e.g., blob storage operations) do not see higher latency from colocation with large, spiky IO requests (e.g., data warehouse operations). Gold TrafficClass requests can miss their latency targets if they are blocked behind lower-priority requests on storage nodes.

Storage nodes use three optimizations to ensure low latency for Gold TrafficClass requests. First, the WRR scheduler provides a greedy optimization where a request from a lower TrafficClass may cede its turn to a higher TrafficClass if the request will have enough time to complete after the higher-TrafficClass request. This helps prevent higher-TrafficClass requests from getting stuck behind a lower-priority request. Second, we limit how many non-Gold IOs may be in flight for every disk. Incoming non-Gold traffic is blocked from scheduling if there are any pending Gold requests and the non-Gold in-flight limit has been reached. This ensures the disk is not busy serving large data warehouse IOs while blob storage requests are waiting. Third, the disk itself may re-arrange the IO requests, i.e., serve a non-Gold request before an earlier Gold request. To manage this, Tectonic stops scheduling non-Gold requests to a disk if a Gold request has been pending on that disk for a threshold amount of time. These three techniques combined effectively maintain the latency profile of smaller IOs, even when outnumbered by larger IOs.

4.2 Multitenant Access Control

Tectonic follows common security principles to ensure that all communications and dependencies are secure. Tectonic additionally provides coarse access control between tenants (to prevent one tenant from accessing another's data) and fine-grained access control within a tenant. Access control must be enforced at each layer of Tectonic, since the Client Library talks to each layer directly. Since access control is on path for every read and write, it must also be lightweight.

Tectonic uses a token-based authorization mechanism that includes which resources can be accessed with the token [31]. An authorization service authorizes top-level client requests (e.g., opening a file), generating an authorization token for the next layer in the filesystem; each subsequent layer likewise authorizes the next layer. The token's payload describes the resource to which access is given, enabling granular access control. Each layer verifies the token and the resource indicated in the payload entirely in memory; verification can be performed in tens of microseconds. Piggybacking token-passing on existing protocols reduces the access control overhead.

5 Tenant-Specific Optimizations

Tectonic supports around ten tenants in the same shared filesystem, each with specific performance needs and workload characteristics. Two mechanisms permit tenant-specific optimizations. First, clients have nearly full control over how to configure an application's interactions with Tectonic; the Client Library manipulates data at the chunk level, the finest possible granularity (§3.4). This Client Library-driven design enables Tectonic to execute operations according to the application's performance needs.

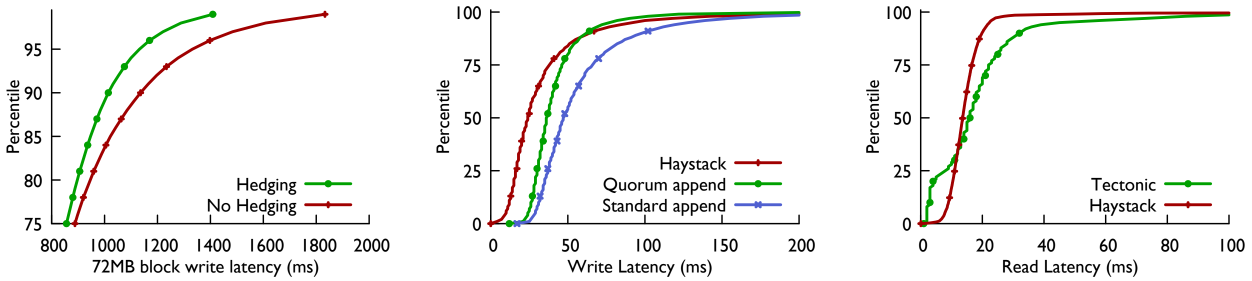
Second, clients enforce configurations on a per-call basis. Many other filesystems bake configurations into the system or apply them to entire files or namespaces. For example, HDFS configures durability per directory [7], whereas Tectonic configures durability per block write. Per-call configuration is enabled by the scalability of the Metadata Store: the Metadata Store can easily handle the increased metadata for this approach. We next describe how data warehouse and blob storage leverage per-call configurations for efficient writes.

5.1 Data Warehouse Write Optimizations

A common pattern in data warehouse workloads is to write data once that will be read many times later. For these workloads, the file is visible to readers only once the creator closes the file. The file is then immutable for its lifetime. Because the file is only read after it is written completely, applications prioritize lower file write time over lower append latency.

Full-block, RS-encoded asynchronous writes for space, IO, and network efficiency. Tectonic uses the write-once-read-many pattern to improve IO and network efficiency, while minimizing total file write time. The absence of partial file reads in this pattern allows applications to buffer writes up to the block size. Applications then RS-encode blocks in memory and write the data chunks to storage nodes. Long-lived data is typically RS(9,6) encoded; short-lived data, e.g., map-reduce shuffles, is typically RS(3,3)-encoded.

Writing RS-encoded full blocks saves storage space, network bandwidth, and disk IO over replication. Storage and bandwidth are lower because less total data is written. Disk IO is lower because disks are more efficiently used. More



(a) Data warehouse full block writes **(b) Blob storage write latency** **(c) Blob storage read latency**

Figure 3: Tail latency optimizations in Tectonic. (a) shows the improvement in data warehouse tail latency from hedged quorum writes (72MB blocks) in a test cluster with ~80% load. (b) and (c) show Tectonic blob storage write latency (with and without quorum appends) and read latency compared to Haystack.

IOPS are needed to write chunks to 15 disks in RS(9,6), but each write is small and the total amount of data written is much smaller than with replication. This results in more efficient disk IO because block sizes are large enough that disk bandwidth, not IOPS, is the bottleneck for full-block writes.

The write-once-read-many pattern also allows applications to write the blocks of a file asynchronously in parallel, which decreases the file write latency significantly. Once the blocks of the file are written, the file metadata is updated all together. There is no risk of inconsistency with this strategy because a file is only visible once it is completely written.

Hedged quorum writes to improve tail latency. For full-block writes, Tectonic uses a variant of quorum writing which reduces tail latency without any additional IO. Instead of sending the chunk write payload to extra nodes, Tectonic first sends reservation requests ahead of the data and then writes the chunks to the first nodes to accept the reservation. The reservation step is similar to hedging [22], but it avoids data transfers to nodes that would reject the request because of lack of resources or because the requester has exceeded its resource share on that node (§4).

As an example, to write a RS(9,6)-encoded block, the Client Library sends a reservation request to 19 storage nodes in different failure domains, four more than required for the write. The Client Library writes the data and parity chunks to the first 15 storage nodes that respond to the reservation request. It acknowledges the write to the client as soon as a quorum of 14 out of 15 nodes return success. If the 15th write fails, the corresponding chunk is repaired offline.

The hedging step is more effective when the cluster is highly loaded. Figure 3a shows ~20% improvement in 99th percentile latency for RS(9,6) encoded, 72 MB full-block writes, in a test cluster with 80% throughput utilization.

5.2 Blob Storage Optimizations

Blob storage is challenging for filesystems because of the quantity of objects that need to be indexed. Facebook stores tens of trillions of blobs. Tectonic manages the size of blob storage metadata by storing many blobs together into log-structured files, where new blobs are appended at the end of a file. Blobs are located with a map from blob ID to the location of the blob in the file.

Blob storage is also on path for many user requests, so low latency is desirable. Blobs are usually much smaller than Tectonic blocks (§2.1). Blob storage therefore writes new blobs as small, replicated partial block appends for low latency. The partial block appends need to be read-after-write consistent so blobs can be read immediately after successful upload. However, replicated data uses more disk space than full-block RS-encoded data.

Consistent partial block appends for low latency. Tectonic uses *partial block quorum appends* to enable durable, low-latency, consistent blob writes. In a quorum append, the Client Library acknowledges a write after a subset of storage nodes has successfully written the data to disk, e.g., two nodes for three-way replication. The temporary decrease of durability from a quorum write is acceptable because the block will soon be reencoded and because blob storage writes a second copy to another datacenter.

The challenge with partial block quorum appends is that straggler appends could leave replica chunks at different sizes. Tectonic maintains consistency by carefully controlling who can append to a block and when appends are made visible. Blocks can only be appended to by the writer that created the block. Once an append completes, Tectonic commits the post-append block size and checksum to the block metadata before acknowledging the partial block quorum append.

This ordering of operations with a single appender provides consistency. If block metadata reports a block size of *S*, then all preceeding bytes in the block were written to at least two storage nodes. Readers will be able to access data in the block up to offset *S*. Similarly, any writes acknowledged to the application will have been updated in the block metadata and so will be visible to future reads. Figures 3b and 3c demonstrate that Tectonic's blob storage read and write latency is comparable to Haystack, validating that Tectonic's generality does

Capacity	Used bytes	Files	Blocks	Storage Nodes
1590 PB	1250 PB	10.7 B	15 B	4208

Table 2: Statistics from a multitenant Tectonic production cluster. File and block counts are in billions.

not have a significant performance cost.

Reencoding blocks for storage efficiency. Directly RS-encoding small partial-block appends would be IO-inefficient. Small disk writes are IOPS-bound and RS-encoding results in many more IOs (e.g, 14 IOs with RS(10, 4) instead of 3). Instead of RS-encoding after each append, the Client Library reencodes the block from replicated form to RS(10,4) encoding once the block is sealed. Reencoding is IO-efficient compared to RS-encoding at append time, requiring only a single large IO on each of the 14 target storage nodes. This optimization, enabled by Tectonic's Client Library-driven design, provides nearly the best of both worlds with fast and IO-efficient replication for small appends that are quickly transitioned to the more space-efficient RS-encoding.

6 Tectonic in Production

This section shows Tectonic operating at exabyte scale, demonstrates benefits of storage consolidation, and discusses how Tectonic handles metadata hotspots. It also discusses tradeoffs and lessons from designing Tectonic.

6.1 Exabyte-Scale Multitenant Clusters

Production Tectonic clusters run at exabyte scale. Table 2 gives statistics on a representative multitenant cluster. All results in this section are for this cluster. The 1250 PB of storage, ~70% of the cluster capacity at the time of the snapshot, consists of 10.7 billion files and 15 billion blocks.

6.2 Efficiency from Storage Consolidation

The cluster in Table 2 hosts two tenants, blob storage and data warehouse. Blob storage uses ~49% of the used space in this cluster and data warehouse uses ~51%. Figures 4a and 4b show the cluster handling storage load over a three-day period. Figure 4a shows the cluster's aggregate IOPS during that time, and Figure 4b shows its aggregate disk bandwidth. The data warehouse workload has large, regular load spikes triggered by very large jobs. Compared to the spiky data warehouse workload, blob storage traffic is smooth and predictable.

Sharing surplus IOPS capacity. The cluster handles spikes in storage load from data warehouse using the surplus IOPS capacity unlocked by consolidation with blob storage. Blob storage requests are typically small and bound by IOPS while data warehouse requests are typically large and bound by bandwidth. As a result, neither IOPS nor bandwidth can fairly account for disk IO usage. The bottleneck resource in serving storage operations is *disk time*, which measures how often a given disk is busy. Handling a storage load spike requires Tectonic to have enough free disk time to serve the spike. For example, if a disk does 10 IOs in one second with each taking 50 ms (seek and fetch), then the disk was busy for 500 out of 1000 ms. We use disk time to fairly account for usage by different types of requests.

	Warehouse	Blob storage	Combined
Supply	**0.51**	**0.49**	**1.00**
Peak 1	0.60	0.12	0.72
Peak 2	0.54	0.14	0.68
Peak 3	0.57	0.11	0.68

Table 3: Consolidating data warehouse and blob storage in Tectonic allows data warehouse to use what would otherwise be stranded surplus disk time for blob storage to handle large load spikes. This figure shows the normalized disk time demand vs. supply in three daily peaks in the representative cluster.

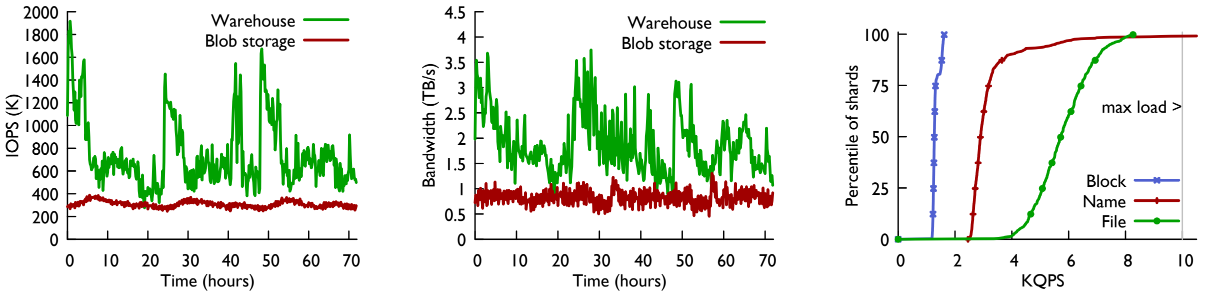
For the representative production cluster, Table 3 shows normalized disk time demand for data warehouse and blob storage for three daily peaks and the supply of disk time each would have if running on its own cluster. We normalize by total disktime corresponding to used space in the cluster. The daily peaks correspond to the same three days of traffic as in Figures 4a and 4b. Data warehouse's demand exceeds its supply in all three peaks and handling it on its own would require disk overprovisioning. To handle peak data warehouse demand over the three day period, the cluster would have needed ~17% overprovisioning. Blob storage, on the other hand, has surplus disk time that would be stranded if it ran in its own cluster. Consolidating these tenants into a single Tectonic cluster allows the blob storage's surplus disk time to be used for data warehouse's storage load spikes.

6.3 Metadata Hotspots

Load spikes to the Metadata Store may result in hotspots in metadata shards. The bottleneck resource in serving metadata operations is queries per second (QPS). Handling load spikes requires the Metadata Store to keep up with the QPS demand on every shard. In production, each shard can serve a maximum of 10 KQPS. This limit is imposed by the current isolation mechanism on the resources of the metadata nodes. Figure 4c shows the QPS across metadata shards in the cluster for the Name, File, and Block layers. All shards in the File and Block layers are below this limit.

Over this three-day period, around 1% of Name layer shards hit the QPS limit because they hold very hot directories. The small unhandled fraction of metadata requests are retried after a backoff. The backoff allows the metadata nodes to clear most of the initial spike and successfully serve retried requests. This mechanism, combined with all other shards being below their maximum, enables Tectonic to successfully handle the large spikes in metadata load from data warehouse.

The distribution of load across shards varies between the Name, File, and Block layers. Each higher layer has a larger distribution of QPS per shard because it colocates more of a



(a) Aggregate cluster IOPS **(b) Aggregate cluster bandwidth** **(c) Peak metadata load (CDF)**

Figure 4: IO and metadata load on the representative production cluster over three days. (a) and (b) show the difference in blob storage and data warehouse traffic patterns and show Tectonic successfully handling spikes in storage IOPS and bandwidth over 3 days. Both tenants occupy nearly the same space in this cluster. (c) is a CDF of peak metadata load over three days on this cluster's metadata shards. The maximum load each shard can handle is 10 KQPS (grey line). Tectonic can handle all metadata operations at the File and Block layers. It can immediately handle almost all Name layer operations; the remaining operations are handled on retry.

tenant's operations. For instance, all directory-to-file lookups for a given directory are handled by one shard. An alternative design that used range partitioning like ADLS [42] would colocate many more of a tenant's operations together and result in much larger load spikes. Data warehouse jobs often read many similarly-named directories, which would lead to extreme load spikes if the directories were range-partitioned. Data warehouse jobs also read many files in a directory, which causes load spikes in the Name layer. Range-partitioning the File layer would colocate files in a directory on the same shard, resulting in a much larger load spike because each job does many more File layer operations than Name layer operations. Tectonic's hash partitioning reduces this colocation, allowing Tectonic to handle metadata load spikes using fewer nodes than would be necessary with range partitioning.

Tectonic also codesigns with data warehouse to reduce metadata hotspots. For example, compute engines commonly use an orchestrator to list files in a directory and distribute the files to workers. The workers open and process the files in parallel. In Tectonic, this pattern sends a large number of nearly-simultaneous file open requests to a single directory shard (§3.3), causing a hotspot. To avoid this anti-pattern, Tectonic's list-files API returns the file IDs along with the file names in a directory. The compute engine orchestrator sends the file IDs and names to its workers, which can open the files directly by file ID without querying the directory shard again.

6.4 The simplicity-performance tradeoffs

Tectonic's design generally prioritizes simplicity over efficiency. We discuss two instances where we opted for additional complexity in exchange for performance gains.

Managing reconstruction load. RS-encoded data may be stored *contiguously*, where a data block is divided into chunks that are each written contiguously to storage nodes, or *striped*, where a data block is divided into much smaller chunks that are distributed round-robin across storage nodes [51]. Because Tectonic uses contiguous RS encoding and the majority of reads are smaller than a chunk size, reads are usually *direct*: they do not require RS reconstruction and so consist of a single disk IO. Reconstruction reads require 10× more IOs than direct reads (for RS(10,4) encoding). Though common, it is difficult to predict the fraction of reads that will be reconstructed, since reconstruction is triggered by hardware failures as well as node overload. We learned that such a wide variability in resource requirements, if not controlled, can cause cascading failures that affect system availability and performance.

If some storage nodes are overloaded, direct reads fail and trigger reconstructed reads. This increases load to the rest of the system and triggers yet more reconstructed reads, and so forth. The cascade of reconstructions is called a *reconstruction storm*. A simple solution would be to use striped RS encoding where all reads are reconstructed. This avoids reconstruction storms because the number of IOs for reads does not change when there are failures. However, it makes normal-case reads much more expensive. We instead prevent reconstruction storms by restricting reconstructed reads to 10% of all reads. This fraction of reconstructed reads is typically enough to handle disk, host, and rack failures in our production clusters. In exchange for some tuning complexity, we avoid over-provisioning disk resources.

Efficiently accessing data within and across datacenters. Tectonic allows clients to directly access storage nodes; an alternative design might use front-end proxies to mediate all client access to storage. Making the Client Library accessible to clients introduces complexity because bugs in the library become bugs in the application binary. However, direct client access to storage nodes is vastly more network- and hardware resource efficient than a proxy design, avoiding an extra network hop for terabytes of data per second.

Unfortunately, direct storage node access is a poor fit for remote requests, where the client is geographically distant from the Tectonic cluster. The additional network overhead makes the orchestration round trips prohibitively inefficient. To solve this problem, Tectonic handles remote data access differently from local data access: remote requests get forwarded to a stateless proxy in the same datacenter as the storage nodes.

6.5 Tradeoffs and Compromises

Migrating to Tectonic was not without tradeoffs and compromises. This subsection describes a few areas where Tectonic is either less flexible or less performant than Facebook's previous infrastructure. We also describe the impact of using a hash-partitioned metadata store.

The impact of higher metadata latency. Migrating to Tectonic meant data warehouse applications saw higher metadata latency. HDFS metadata operations are in-memory and all metadata for a namespace is stored on a single node. In contrast, Tectonic stores its metadata in a sharded key-value store instance and disaggregates metadata layers (§3.3). This means Tectonic metadata operations may require one or more network calls (e.g., a file open operation will interact with the Name and File layers). Data warehouse had to adjust how it handled certain metadata operations given the additional metadata latency. For instance, compute engines rename a set of files one by one, in sequence, after computation is done. In HDFS each rename was fast, but with Tectonic, compute engines parallelize this step to hide the extra latency of individual Tectonic rename operations.

Working around hash-partitioned metadata. Because Tectonic directories are hash sharded, listing directories recursively involves querying many shards. In fact, Tectonic does not provide a recursive list API; tenants need to build it as a client-side wrapper over individual *list* calls. As a result, unlike HDFS, Tectonic does not have *du* (directory utilization) functionality to query aggregate space usage of a directory. Instead, Tectonic periodically aggregates per-directory usage statistics, which can be stale.

6.6 Design and Deployment Lessons

Achieving high scalability is an iterative process enabled by a microservice architecture. Several Tectonic components have been through multiple iterations to meet increasing scalability requirements. For example, the first version of the Chunk Store grouped blocks to reduce metadata. A number of blocks with the same redundancy scheme were grouped and RS-encoded as one unit to store their chunks together. Each block group mapped to a set of storage nodes. This is a common technique since it significantly reduces metadata [37, 53], but it was too inflexible for our production environment. For example, with only 5% of storage nodes unavailable, 80% of the block groups became unavailable for writes. This design also precluded optimizations like hedged quorum writes and quorum appends (§5).

Additionally, our initial Metadata Store architecture did not separate the Name and File layers; clients consulted the same shards for directory lookups and for listing blocks in a file. This design resulted in unavailability from metadata hotspots, prompting us to further disaggregate metadata.

Tectonic's evolution shows the importance of trying new designs to get closer to performance goals. Our development experience also shows the value of a microservices-based architecture for experimentation: we could iterate on components transparently to the rest of the system.

Memory corruption is common at scale. At Tectonic's scale, with thousands of machines reading and writing a large amount of data every day, in-memory data corruption is a regular occurrence, a phenomenon observed in other large-scale systems [12, 27]. We address this by enforcing checksum checks within and between process boundaries.

For data D and checksum C_D, if we want to perform an in-memory transformation F such that $D' = F(D)$, we generate checksum $C_{D'}$ for D'. To check D', we must convert D' back to D with G, the inverse function of F, and compare $C_{G(D')}$ with C_D. The inverse function, G, may be expensive to compute (e.g., for RS encoding or encryption), but it is an acceptable cost for Tectonic to preserve data integrity.

All API boundaries involving moving, copying, or transforming data had to be retrofitted to include checksum information. Clients pass a checksum with data to the Client Library when writing, and Tectonic needs to pass the checksum not just across process boundaries (e.g., between the client library and the storage node) but also within the process (e.g., after transformations). Checking the integrity of transformations prevents corruptions from propagating to reconstructed chunks after storage node failure.

6.7 Services that do not use Tectonic

Some services within Facebook do not use Tectonic for storage. Bootstrap services, e.g., the software binary package deployment system, which must have no dependencies, cannot use Tectonic because it depends on many other services (e.g., the key-value store, configuration management system, deployment management system). Graph storage [16] also does not use Tectonic, as Tectonic is not yet optimized for key-value store workloads which often need the low latencies provided by SSD storage.

Many other services do not use Tectonic directly. They instead use Tectonic through a major tenant like blob storage or data warehouse. This is because a core design philosophy of Tectonic is separation of concerns. Internally, Tectonic aims for independent software layers which each focus on a narrow set of a storage system's core responsibilities (e.g., storage nodes only know about chunks but not blocks or files). This philosophy extends to how Tectonic fits in with the rest of the storage infrastructure. For example, Tectonic focuses on providing fault tolerance within a datacenter; it does not protect against datacenter failures. Geo-replication is a separate problem that Tectonic delegates to its large tenants, who solve it to provide transparent and easy-to-use shared storage for applications. Tenants are also expected to know details of capacity management and storage deployments and rebalancing across different datacenters. For smaller applications, the complexity and implementation needed to interface directly with Tectonic in a way that meets their storage needs would amount to re-implementing features that tenants have already implemented. Individual applications therefore use Tectonic via tenants.

7 Related Work

Tectonic adapts techniques from existing systems and the literature, demonstrating how they can be combined into a novel system that realizes exabyte-scale single clusters which support a diversity of workloads on a shared storage fabric.

Distributed filesystems with a single metadata node. HDFS [15], GFS [24], and others [38, 40, 44] are limited by the metadata node to tens of petabytes of storage per instance or cluster, compared to Tectonic's exabytes per cluster.

Federating namespaces for increased capacity. Federated HDFS [8] and Windows Azure Storage (WAS) [17] combine multiple smaller storage clusters (with a single metadata node) into larger clusters. For instance, a federated HDFS [8] cluster has multiple independent single-namenode namespaces, even though the storage nodes are shared between namespaces. Federated systems still have the operational complexity of bin-packing datasets (§2). Also, migrating or sharing data between instances, e.g., to load-balance or add storage capacity, requires resource-heavy data copying among namespaces [33, 46, 54]

Hash-based data location for metadata scalability. Ceph [53] and FDS [36] eliminate centralized metadata, instead locating data by hashing on object ID. Handling failures in such systems is a scalability bottleneck. Failures are more frequent with larger clusters, requiring frequent updates to the hash-to-location map that must propagate to all nodes. Yahoo's Cloud Object Store [41] federates Ceph instances to isolate the effects of failures. Furthermore, adding hardware and draining is complicated, as Ceph lacks support for controlled data migration [52]. Tectonic explicitly maps chunks to storage nodes, allowing controlled migration.

Disaggregated or sharded metadata for scalability. Like Tectonic, ADLS [42] and HopsFS [35] increase filesystem capacity by disaggregating metadata into layers in separate sharded data stores. Tectonic hash-partitions directories, while ADLS and HopsFS store some related directory metadata on the same shards, causing metadata for related parts of the directory tree to be colocated. Hash partitioning helps Tectonic avoid hotspots local to part of the directory tree. ADLS uses WAS's federated architecture [17] for block storage. In contrast, Tectonic's block storage is flat.

Like Tectonic, Colossus [28, 32] provides cluster-wide multi-exabyte storage where client libraries directly access storage nodes. Colossus uses Spanner [21], a globally consistent database to store filesystem metadata. Tectonic metadata is built on a sharded key-value store, which only provides within-shard strong consistency and no cross-shard operations. These limitations have not been a problem in practice.

Blob and object stores. Compared to distributed filesystems, blob and object stores [14, 18, 36, 37] are easier to scale, as they do not have a hierarchical directory tree or namespace to keep consistent. Hierarchical namespaces are required for most warehouse workloads.

Other large-scale storage systems. Lustre [1] and GPFS [45] are tuned for high-throughput parallel access. Lustre limits the number of metadata nodes, limiting scalability. GPFS is POSIX-compliant, introducing unnecessary metadata management overhead for our setting. HBase [9] is a key-value store based on HDFS, but its HDFS clusters are not shared with a warehouse workload. We could not compare with AWS [2] as its design is not public.

Multitenancy techniques. Tectonic's multitenancy techniques were co-designed with the filesystem as well as the tenants, and does not aim to achieve optimal fair sharing. It is thus easier to provide performance isolation compared to other systems in the literature. Other systems use more complex resource management techniques to accommodate changes in tenancy and resource use policies, or to provide optimal fair resource sharing among tenants [25, 48, 49].

Some details of Tectonic have previously been described in talks [39, 47] where the system is called Warm Storage.

8 Conclusion

This paper presents Tectonic, Facebook's distributed filesystem. A single Tectonic instance can support all Facebook's major storage tenants in a datacenter, enabling better resource utilization and less operational complexity. Tectonic's hash-sharded disaggregated metadata and flat data chunk storage allow it to address and store exabytes. Its cardinality-reduced resource management allows it to efficiently and fairly share resources and distribute surplus resources for high utilization. Tectonic's client-driven tenant-specific optimizations allow it to match or exceed the performance of the previous specialized storage systems.

Acknowledgements. We are grateful to our shepherd, Peter Macko, and the anonymous reviewers of the FAST program committee whose extensive comments substantially improved this work. We are also grateful to Nar Ganapathy, Mihir Gorecha, Morteza Ghandehari, Bertan Ari, John Doty, and other colleagues at Facebook who contributed to the project. We also thank Jason Flinn and Qi Huang for suggestions for improving the paper. Theano Stavrinos was supported by the National Science Foundation grant CNS-1910390 while at Princeton University.

References

[1] Lustre Wiki. https://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf, 2017.

[2] AWS Documentation. https://docs.aws.amazon.com/, 2020.

[3] Presto. https://prestodb.io/, 2020.

[4] Aditya Kalro. Facebook's FBLearner Platform with Aditya Kalro. https://twimlai.com/twiml-talk-197-facebooks-fblearner-platform-with-aditya-kalro/, 2018.

[5] J. Adrian. Introducing Bryce Canyon: Our next-generation storage platform. https://tinyurl.com/yccx2x7v, 2017.

[6] M. Annamalai. ZippyDB - A Distributed key value store. https://www.youtube.com/embed/ZRP7z0HnClc, 2015.

[7] Apache Software Foundation. HDFS Erasure Coding. https://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, 2018.

[8] Apache Software Foundation. HDFS Federation. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html, 2019.

[9] Apache Software Foundation. Apache HBase. https://hbase.apache.org/, 2020.

[10] Apache Software Foundation. Apache Spark. https://spark.apache.org/, 2020.

[11] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, Vancouver, BC, Canada, 2010. USENIX Association.

[12] D. Behrens, M. Serafini, F. P. Junqueira, S. Arnautov, and C. Fetzer. Scalable error isolation for distributed systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, Oakland, CA, USA, 2015. USENIX Association.

[13] B. Berg, D. S. Berger, S. McAllister, I. Grosof, J. Gunasekar, Sathya Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Online, 2020. USENIX Association.

[14] A. Bigian. Blobstore: Twitter's in-house photo storage system. https://blog.twitter.com/engineering/en_us/a/2012/blobstore-twitter-s-in-house-photo-storage-system.html, 2012.

[15] D. Borthakur. HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2019.

[16] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*. USENIX, 2013.

[17] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, Cascais, Portugal, 2011. Association for Computing Machinery (ACM).

[18] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears. Walnut: a unified cloud object store. 2012.

[19] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[20] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, San Jose, CA, USA, 2013. USENIX Association.

[21] 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. *ACM Trans. Comput. Syst.*, 31(3), Aug. 2013. ISSN 0734-2071. doi: 10.1145/2491245. URL https://doi.org/10.1145/2491245.

[22] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013. ISSN 0001-

0782. doi: 10.1145/2408776.2408794. URL http://doi.acm.org/10.1145/2408776.2408794.

[23] Facebook Open Source. RocksDB. https://rocksdb.org/, 2020.

[24] 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)*, Bolton Landing, NY, USA, 2003. Association for Computing Machinery (ACM).

[25] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom. Crystal: Software-defined storage for multi-tenant object stores. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Santa Clara, CA, USA, 2017. USENIX Association.

[26] X. F. Group. The XFS Linux wiki. https://xfs.wiki.kernel.org/, 2018.

[27] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Dhoot, A. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB'14)*, Hangzhou, China, 2014. VLDB Endowment.

[28] D. Hildebrand and D. Serenyi. A peek behind the VM at the Google Storage infrastructure. https://www.youtube.com/watch?v=q4WC_6SzBz4, 2020.

[29] Q. Huang, P. Ang, P. Knowles, T. Nykiel, I. Tverdokhlib, A. Yajurvedi, P. Dapolito IV, X. Yan, M. Bykov, C. Liang, M. Talwar, A. Mathur, S. Kulkarni, M. Burke, and W. Lloyd. SVE: Distributed video processing at Facebook scale. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, Shanghai, China, 2017. Association for Computing Machinery (ACM).

[30] L. Leslie. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[31] K. Lewi, C. Rain, S. A. Weis, Y. Lee, H. Xiong, and B. Yang. Scaling backend authentication at facebook. *IACR Cryptol. ePrint Arch.*, 2018:413, 2018. URL https://eprint.iacr.org/2018/413.

[32] M. K. McKusick and S. Quinlan. GFS: Evolution on Fast-forward. *Queue*, 7(7):10:10–10:20, Aug. 2009. ISSN 1542-7730. doi: 10.1145/1594204.1594206. URL http://doi.acm.org/10.1145/1594204.1594206.

[33] P. A. Misra, I. n. Goiri, J. Kace, and R. Bianchini. Scaling Distributed File Systems in Resource-Harvesting Datacenters. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, Santa Clara, CA, USA, 2017. USENIX Association.

[34] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's Warm BLOB Storage System. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, USA, 2014. USENIX Association.

[35] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström. HopsFS: Scaling hierarchical file system metadata using NewSQL databases. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Santa Clara, CA, USA, 2017. USENIX Association.

[36] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat Datacenter Storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, Hollywood, CA, USA, 2012. USENIX Association.

[37] S. A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, M. Zadeh, T. Li, I. Gupta, and R. H. Campbell. Ambry: Linkedin's scalable geo-distributed object store. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, San Francisco, California, USA, 2016. Association for Computing Machinery (ACM).

[38] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB'13)*, Riva del Garda, Italy, 2013. VLDB Endowment.

[39] K. Patiejunas and A. Jaiswal. Facebook's disaggregated storage and compute for Map/Reduce. https://atscaleconference.com/videos/facebooks-disaggregated-storage-and-compute-for-mapreduce/, 2016.

[40] A. J. Peters and L. Janyst. Exabyte scale storage at CERN. *Journal of Physics: Conference Series*, 331 (5):052015, dec 2011. doi: 10.1088/1742-6596/331/5/052015. URL https://doi.org/10.1088/1742-6596/331/5/052015.

[41] N. P.P.S, S. Samal, and S. Nanniyur. Yahoo Cloud Object Store - Object Storage at Exabyte Scale. https://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at, 2015.

[42] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan. Azure Data Lake Store: a hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD'17)*, Chicago, IL, USA, 2017. Association for Computing Machinery (ACM).

[43] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[44] Rousseau, Hervé, Chan Kwok Cheong, Belinda, Contescu, Cristian, Espinal Curull, Xavier, Iven, Jan, Gonzalez Labrador, Hugo, Lamanna, Massimo, Lo Presti, Giuseppe, Mascetti, Luca, Moscicki, Jakub, and van der Ster, Dan. Providing large-scale disk storage at cern. *EPJ Web Conf.*, 214:04033, 2019. doi: 10.1051/epjconf/201921404033. URL https://doi.org/10.1051/epjconf/201921404033.

[45] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, USA, 2002. USENIX Association.

[46] R. Shah. Enabling HDFS Federation Having 1B File System Objects. https://tech.ebayinc.com/engineering/enabling-hdfs-federation-having-1b-file-system-objects/, 2020.

[47] S. Shamasunder. Hybrid XFS—Using SSDs to Supercharge HDDs at Facebook. https://www.usenix.org/conference/srecon19asia/presentation/shamasunder, 2019.

[48] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, Hollywood, CA, USA, 2012. USENIX Association.

[49] A. K. Singh, X. Cui, B. Cassell, B. Wong, and K. Daudjee. Microfuge: A middleware approach to providing performance isolation in cloud storage systems. In *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems (ICDCS'14)*, Madrid, Spain, 2014. IEEE Computer Society.

[50] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*, Indianapolis, IN, USA, 2010. Association for Computing Machinery (ACM).

[51] A. Wang. Introduction to HDFS Erasure Coding in Apache Hadoop. https://blog.cloudera.com/introduction-to-hdfs-erasure-coding-in-apache-hadoop/, 2015.

[52] L. Wang, Y. Zhang, J. Xu, and G. Xue. MAPX: Controlled Data Migration in the Expansion of Decentralized Object-Based Storage Systems. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, Santa Clara, CA, USA, 2020. USENIX Association.

[53] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, Seattle, WA, USA, 2006. USENIX Association.

[54] A. Zhang and W. Yan. Scaling Uber's Apache Hadoop Distributed File System for Growth. https://eng.uber.com/scaling-hdfs/, 2018.

Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage

Yuchong Hu[†], Liangfeng Cheng[†], Qiaori Yao[†], Patrick P. C. Lee[‡], Weichun Wang[*], Wei Chen[*]
[†]*Huazhong University of Science & Technology* [‡]*The Chinese University of Hong Kong* [*]*HIKVISION*

Abstract

Erasure coding is a low-cost redundancy mechanism for distributed storage systems by storing stripes of data and parity chunks. *Wide stripes* are recently proposed to suppress the fraction of parity chunks in a stripe to achieve extreme storage savings. However, wide stripes aggravate the repair penalty, while existing repair-efficient approaches for erasure coding cannot effectively address wide stripes. In this paper, we propose *combined locality*, the first mechanism that systematically addresses the wide-stripe repair problem via the combination of both parity locality and topology locality. We further augment combined locality with efficient encoding and update schemes. Experiments on Amazon EC2 show that combined locality reduces the single-chunk repair time by up to 90.5% compared to locality-based state-of-the-arts, with only a redundancy of as low as $1.063\times$.

1 Introduction

Erasure coding is an established low-cost redundancy mechanism for protecting data storage against failures in modern distributed storage systems [25, 34, 47]; in particular, Reed-Solomon (RS) codes [61] are widely adopted in today's erasure coding deployment [26, 45, 47, 59, 72]. At a high level, for some configurable parameters n and k (where $k < n$), RS codes compose multiple *stripes* of n chunks, including k original uncoded data chunks and $n-k$ coded parity chunks, such that any k out of n chunks of the same stripe suffice to reconstruct the original k data chunks (see §2.1 for details). Each stripe of n chunks is distributed across n nodes to tolerate any $n-k$ node failures. RS codes incur a *minimum* redundancy of $\frac{n}{k}\times$ (i.e., no other erasure codes can have a lower redundancy than RS codes while tolerating any $n-k$ node failures). In contrast, traditional replication incurs a redundancy of $(n-k+1)\times$ to tolerate the same number of any $n-k$ node failures. For example, Facebook f4 [47] uses $(14,10)$ RS codes to tolerate any four node failures with a redundancy of $1.4\times$, while replication needs a redundancy of $5\times$ for the same four-node fault tolerance. With proper parameterization of (n,k), erasure coding can limit the redundancy to at most $1.5\times$ (see Table 1).

Storage systems	(n,k)	Redundancy
Google Colossus [25]	(9,6)	1.50
Quantcast File System [49]	(9,6)	1.50
Hadoop Distributed File System [3]	(9,6)	1.50
Baidu Atlas [36]	(12,8)	1.50
Facebook f4 [47]	(14,10)	1.40
Yahoo Cloud Object Store [48]	(11,8)	1.38
Windows Azure Storage [34]	(16,12)	1.33
Tencent Ultra-Cold Storage [8]	(12,10)	1.20
Pelican [12]	(18,15)	1.20
Backblaze Vaults [13]	(20,17)	1.18

Table 1: Common parameters of (n,k) in state-of-the-art erasure coding deployment. Note that a similar table is also presented in [22], while we add Azure and Pelican here.

Conventional wisdom suggests that erasure coding parameters should be configured in a *medium* range [53]. Table 1 lists the parameters (n,k) used by state-of-the-art production systems. We see that the number of tolerable failures $n-k$ is typically three or four, while the stripe size n is no more than 20. One major reason of choosing a moderate stripe size is to limit the *repair penalty* of erasure coding, in which repairing any single lost chunk needs to retrieve multiple available chunks of the same stripe for decoding the lost chunk (e.g., k chunks are retrieved in (n,k) RS codes). A larger stripe size n, and hence a larger k for tolerating the same $n-k$ node failures, implies more severe bandwidth and I/O amplifications in repair and hence compromises storage reliability.

While erasure coding effectively mitigates storage redundancy, we explore further redundancy reduction under erasure coding to achieve *extreme* storage savings; for example, a redundancy reduction of 14% (from $1.5\times$ to $1.33\times$) can translate to millions of dollar savings in production [52]. This motivates us to explore *wide stripes*, in which n and k are very large, while the number of tolerable failures $n-k$ remains three to four as in state-of-the-art production systems. Wide stripes are studied in storage industry (e.g., VAST [9]), and provide an opportunity to achieve *near-optimal redundancy* (i.e., $\frac{n}{k}$ approaches one) with the maximum possible storage savings. For example, VAST [9] considers a setting of $(n,k) = (154,150)$, thereby incurring only a redundancy of $1.027\times$. We argue that the significant storage efficiency of wide stripes is attractive for both *cold* and *hot* distributed storage systems. Erasure coding is traditionally used by cold storage systems (e.g., backup and archival applications), in which data needs to be persistently stored but is rarely accessed [2, 10, 12]. Wide stripes allow cold storage systems to achieve long-term data durability at extremely low cost. Erasure coding is also adopted by hot storage systems (e.g., in-memory key-value stores) to provide data availability for key-value objects that are frequently accessed in the face of failures and stragglers [18, 57, 73, 74]. Wide stripes allow hot storage systems to significantly reduce expensive hardware footprints (e.g., DRAM for in-memory key-value stores).

While wide stripes achieve extreme storage savings, they further aggravate the repair penalty, as the repair bandwidth (i.e., the amount of data transfers during repair) increases with k. Many existing repair-efficient approaches for erasure-coded storage leverage *locality* to reduce the repair bandwidth. There are two types of locality: (i) *parity locality*, which introduces extra local parity chunks to reduce the number of available chunks to retrieve for repairing a lost chunk [14, 27, 34, 39, 51, 63]; and (ii) *topology locality*, which takes into account the hierarchical nature of the system topology and performs local repair operations to mitigate the cross-rack (or cross-cluster) repair bandwidth [31, 32, 56, 65, 66, 68].

However, existing locality-based repair approaches still mainly focus on stripes with a small k (e.g., $k = 12$ [34] and $k = 6$ [32]). They inevitably increase the redundancy or degrade the repair performance for wide stripes as k increases (§2.3). The reason is that the near-optimal redundancy of wide stripes reduces the benefits brought by either parity locality or topology locality (§3.5).

In this paper, we present *combined locality*, a new repair mechanism that systematically combines both parity locality and topology locality to address the repair problem in wide-stripe erasure coding. Combined locality associates local parity chunks with a small subset of data chunks (i.e., parity locality) and localizes a repair operation in a limited number of racks (i.e., topology locality), so as to provide better trade-offs between redundancy and repair performance than existing locality-based state-of-the-arts. In addition, we revisit the classical encoding and update problems for wide-stripe erasure coding under combined locality and design the corresponding efficient schemes. Our contributions include:

- We are the *first* to systematically address the wide-stripe repair problem. We propose combined locality, which mitigates the cross-rack repair bandwidth under ultra-low storage redundancy. We examine the trade-off between redundancy and cross-rack repair bandwidth for different locality-based schemes (§3).
- We design ECWide, which realizes combined locality to address two types of repair: single-chunk repair and full-node repair. We also design (i) an efficient encoding scheme that allows the parity chunks of a wide stripe to be encoded across multiple nodes in parallel, and (ii) an inner-rack parity update scheme that allows parity chunks to be locally updated within racks to reduce cross-rack transfers (§4).
- We implement two ECWide prototypes, namely ECWide-C and ECWide-H, to realize combined locality. The former is designed for cold storage, while the latter builds on a Memcached-based [5, 6] in-memory key-value store for hot storage (§5). The source code of our prototypes is now available at **https://github.com/yuchonghu/ecwide**.
- We compare via Amazon EC2 experiments ECWide-C and ECWide-H with two existing locality-based schemes: (i) Azure's Local Reconstruction Codes (Azure-LRC) [34] adopted in production, and (ii) the recently proposed topology-locality-based repair approach [32, 65] that minimizes the cross-rack repair bandwidth for fast repair. We show that combined locality significantly reduces the single-chunk repair time by up to 87.9% and 90.5% of the above two schemes, respectively, while incurring a redundancy of as low as 1.063× only. We also validate the efficiency of our encoding and update schemes (§6).

2 Background and Motivation

We provide the background details of erasure coding for distributed storage (§2.1), and state the challenges of deploying wide-stripe erasure coding (§2.2). We describe how existing studies exploit locality to address the repair problem (§2.3), and motivate the idea of our combined locality design (§2.4).

2.1 Erasure Coding for Distributed Storage

Consider a distributed storage system that organizes data in fixed-size *chunks* spanning across a number of storage nodes, such that erasure coding operates in units of chunks. Depending on the types of storage workloads, the chunk size used for erasure coding can vary significantly, ranging from as small as 4 KiB in in-memory key-value storage (i.e., hot storage) [18, 73, 74], to as large as 256 MiB [59] in persistent file storage (i.e., cold storage) for small I/O seek costs. Erasure coding can be constructed in different forms, among which RS codes [61] are the most popular erasure codes and widely deployed (§1).

To deploy RS codes in distributed storage, we configure two integer parameters n and k (where $k < n$). An (n,k) RS code works by encoding k fixed-size (uncoded) *data chunks* into $n-k$ (coded) *parity chunks* of the same size. RS codes are *storage-optimal* (a.k.a. *maximum distance separable (MDS)* in coding theory terms), meaning that any k out of the n chunks suffice to reconstruct all k data chunks (i.e., any $n-k$ lost chunks can be tolerated for data availability), while the redundancy (i.e., $\frac{n}{k}$ times the original data size) is the minimum among all possible erasure code constructions. We call each set of n chunks a *stripe*. A distributed storage system contains multiple stripes that are independently encoded, and the n chunks of each stripe are stored in n different nodes to provide fault tolerance against any $n-k$ node failures.

Mathematically, each parity chunk in an (n,k) RS code is formed by a linear combination of the k data chunks of the same stripe based on the arithmetic of the Galois Field $GF(2^w)$ in w-bit words [53] (where $n \leq 2^w$). Specifically, let $D_1, D_2, \cdots, D_k$ be the k data chunks of a stripe, and $P_1, P_2, \cdots, P_{n-k}$ be the $n-k$ parity chunks of the same stripe. Each parity chunk P_i ($1 \leq i \leq n-k$) can be expressed as $P_i = \sum_{j=1}^{k} \alpha_{i,j} D_j$, where $\alpha_{i,j}$ denotes some encoding coefficient. In this work, we focus on *Cauchy RS codes* [15, 55], where the encoding coefficients are defined based on the Cauchy matrix, so that we can construct *systematic* RS codes (i.e., the k data chunks are included in a stripe for direct access).

2.2 Challenges of Wide-Stripe Erasure Coding

We explore wide-stripe erasure coding with both large n and k, so as to achieve an ultra-low redundancy $\frac{n}{k}$ (i.e., approaching one). However, it poses three performance challenges.

Expensive repair. Erasure coding is known to incur the repair penalty, and it is even more severe for wide stripes. For an (n,k) RS code, the conventional approach for repairing a single lost chunk is to retrieve k available chunks from other non-failed nodes, implying that the bandwidth and I/O costs are amplified k times. Even though new erasure code constructions can mitigate the repair bandwidth and I/O costs (e.g., regenerating codes [23] or locally repairable codes [27, 33, 34, 51, 63]), the repair bandwidth and I/O amplifications still exist and become more prominent as k increases, as proven by theoretical analysis [23].

The high repair penalty of wide stripes manifests differently in cold and hot storage workloads. For cold storage workloads with large chunk sizes, the repair bandwidth is much more significant for large k. For example, if we configure a wide stripe with $k = 128$ and the chunk size is 256 MiB [59], the single-chunk repair bandwidth becomes 32 GiB. We may interpolate that the daily repair bandwidth of 180 TiB for the $(14,10)$ RS code [59] will increase to 2.25 PiB for $k = 128$. For hot storage workloads with small chunk sizes, although its single-chunk repair bandwidth is much less than in cold storage, a large k incurs a significant tail latency under frequent accesses, as the repair is now more likely bottlenecked by any straggler node out of the k non-failed nodes.

Expensive encoding. The (per-stripe) encoding overhead of erasure coding becomes more prominent as k increases (the same arguments hold for decoding). In an (n,k) RS code, each parity chunk is a linear combination of k data chunks (§2.1), so the computational overhead increases linearly with k. Most importantly, as k increases, it becomes more difficult for the encoding process to fit the input data of a wide stripe into CPU cache, leading to significant encoding performance degradations. Figure 1 shows the encoding throughput on three Intel CPU families versus k, using the Intel ISA-L encoding APIs [4]. Here, we fix a chunk size of 64 MiB and $n-k = 4$. We see that the encoding throughput remains high from $k = 4$ to $k = 16$, but drops dramatically as k further increases from $k = 32$ onwards; for example, the throughput drops by 43-70% from $k = 4$ to $k = 128$.

Expensive updates. The (per-stripe) update overhead of erasure coding is significant: if any data chunk of the same stripe has been updated, all $n-k$ parity chunks need to be updated. Wide stripes suffer the same expensive update issue as in traditional stripes of moderate sizes.

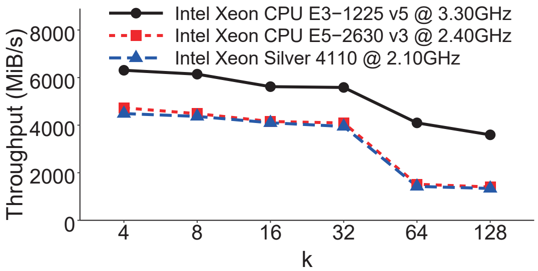


Figure 1: Encoding throughput on different Intel CPU families versus k for a chunk size of 64 MiB and $n-k = 4$.

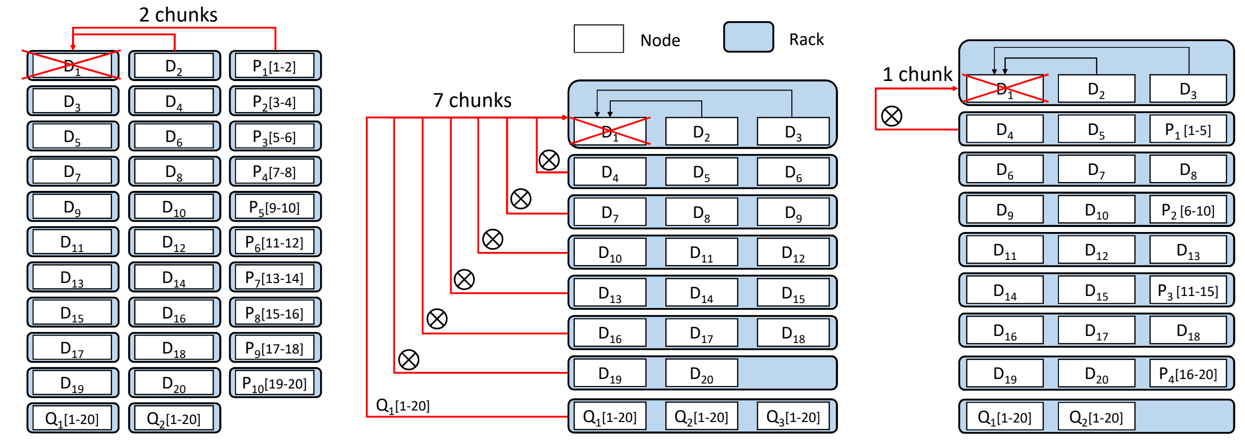
2.3 Locality in Erasure-coded Repair

The main challenge of wide-stripe erasure coding is the repair problem. Existing studies on the erasure-coded repair problem have led to a rich body of literature, and many of them focus on using *locality* to reduce the repair bandwidth, including *parity locality* and *topology locality*.

Parity locality. Recall that an (n,k) RS code needs to retrieve k chunks for repairing a lost chunk. Parity locality adds *local parity chunks* to reduce the number of surviving chunks (and hence the repair bandwidth and I/O) for repairing a lost chunk. Its representative erasure code construction is the *locally repairable codes (LRCs)* [27, 33, 34, 51, 63]. Take Azure's Local Reconstruction Codes (Azure-LRC) [34] as an example. Given three configurable parameters n, k, and r (where $r < k < n$), an (n,k,r) Azure-LRC encodes each local group of r data chunks (except the last group, which may have fewer than r data chunks) into a local parity chunk, so that the repair of a lost chunk now only accesses r surviving chunks ($r < k$). It also contains $n-k-\lceil \frac{k}{r} \rceil$ *global parity chunks* encoded from all data chunks. Azure-LRC satisfies the *Maximally Recoverable* property [34] and can tolerate any $n-k-\lceil \frac{k}{r} \rceil + 1$ node failures.

Figure 2(a) shows the $(32,20,2)$ Azure-LRC [34]. It has 20 data chunks (denoted by $D_1, D_2, \ldots, D_{20}$). It has 10 local parity chunks, in which the ℓ-th local parity chunk $P_\ell[i\text{-}j]$ (where $1 \leq \ell \leq 10$) is a linear combination of data chunks $D_i, D_{i+1}, \ldots, D_j$. It also has two global parity chunks $Q_1[1\text{-}20]$ and $Q_2[1\text{-}20]$, each of which is a linear combination of all 20 data chunks. All the above 32 chunks are placed in 32 nodes to tolerate any three node failures. Thus, the $(32,20,2)$ Azure-LRC has a single-chunk repair bandwidth of two chunks (e.g., repairing D_1 needs to access D_2 and $P_1[1\text{-}2]$), while incurring a redundancy of 1.6×. In contrast, the $(23,20)$ RS code also has 20 data chunks and is tolerable against any three node failures. Its single-chunk repair bandwidth is 20 chunks, yet its redundancy is only 1.15×. In short, parity locality *reduces the repair bandwidth but incurs high redundancy*.

Topology locality. Existing erasure-coded storage systems [34, 47, 58, 60, 63] (including Azure-LRC) place each chunk of a stripe in a distinct node residing in a distinct rack. This



(a) Parity locality: (32, 20, 2) Azure-LRC **(b) Topology locality: (23, 20, 8) TL** **(c) Combined locality: (26, 20, 5, 9) CL**

Figure 2: Examples of three locality-based schemes, each of which stores 20 data chunks and can tolerate any three node failures.

provides tolerance against the same numbers of node failures and rack failures, but the repair incurs substantial cross-rack bandwidth, which is often much more constrained than inner-rack bandwidth [20].

Recent studies [31, 32, 65] exploit *topology locality* to reduce the cross-rack repair bandwidth by localizing the repair operations within racks, at the expense of reduced rack-level fault tolerance. They store the chunks of a stripe in multiple nodes within a rack, and split a repair operation into inner-rack and cross-rack repair sub-operations. The cross-rack repair bandwidth is *provably* minimized, subject to the minimum redundancy [31, 32, 65]. Some similar studies focus on minimizing the cross-cluster repair bandwidth via inner-cluster repair sub-operations [56, 66, 68]. We define a topology locality scheme as (n,k,z) TL, in which (n,k) RS-coded chunks are placed in z racks (or clusters).

Figure 2(b) shows the $(23,20,8)$ TL that places 20 data chunks and three RS-coded parity chunks in 23 nodes that reside in eight racks, so as to tolerate any three node failures and one rack failure. The $(23,20,8)$ TL has the minimum redundancy of $1.15\times$, but transfers seven cross-rack chunks to repair a lost chunk. For example, repairing D_1 needs to retrieve Q_1[1-20] and six chunks that are linear parts of Q_1[1-20] from other racks, so that D_1 can be solved from Q_1[1-20] by canceling out the linear parts, D_2, and D_3. The single-chunk repair bandwidth is higher than that of the $(32,20,2)$ Azure-LRC (i.e., two chunks). In short, topology locality *achieves the minimum redundancy, but incurs high cross-rack repair bandwidth.*

2.4 Motivating Example

For wide stripes with a large k, neither parity locality (high redundancy) nor topology locality (high repair penalty) can effectively balance the trade-off between redundancy and repair penalty. This motivates us to combine both types of locality to obtain a better trade-off and hence make wide stripes practically applicable.

Figure 2(c) shows the idea. We encode 20 data chunks into 26 chunks via the $(26,20,5)$ Azure-LRC. We place the chunks across nine racks, and denote the scheme by the $(26,20,5,9)$ CL (see §3.1 for definition). In this case, repairing the lost chunk D_1 can be solved by canceling out D_2, D_3, D_4, and D_5 from P_1[1-5]. The single-chunk repair bandwidth is only one cross-rack chunk, less than both the $(32,20,2)$ Azure-LRC (two chunks) and the $(23,20,8)$ TL (seven chunks). Meanwhile, the redundancy is $1.3\times$, much closer to the minimum redundancy than the $(32,20,2)$ Azure-LRC $(1.6\times)$.

3 Combined Locality

In this section, we present *combined locality*, which exploits the combination of parity locality and topology locality to reduce the cross-rack repair bandwidth subject to limited redundancy for wide-stripe erasure coding. We provide definitions and state our design objective (§3.1), and show our design idea of combined locality (§3.2). We analyze and select the suitable LRC construction for combined locality (§3.3). We present the details of the combined locality mechanism (§3.4), and analyze its trade-off between redundancy and cross-rack repair bandwidth (§3.5). Finally, we present reliability analysis on combined locality (§3.6). Table 2 summarizes the notation.

3.1 Design Objective

We define the combined locality mechanism as (n,k,r,z) CL, which combines (n,k,r) Azure-LRC and (n,k,z) TL across z racks (note that we justify our choice of Azure-LRC in §3.3). Our primary objective of the combined locality mechanism is to determine the parameters (n,k,r,z) that minimize the cross-rack repair bandwidth, subject to: (i) the number of tolerable node failures (denoted by f) and (ii) the maximum allowed redundancy (denoted by γ). For wide-stripe erasure coding, we consider a large k (e.g., $k=128$) for a typical fault tolerance level shown in Table 1 (e.g., $f=4$).

Notation	Description
n	total number of chunks of a stripe
k	number of data chunks of a stripe
r	number of retrieved chunks to repair a lost chunk
z	number of racks to store a stripe
c	number of chunks of a stripe in a rack
f	number of tolerable node failures of a stripe
γ	maximum allowed redundancy

Table 2: Notation for combined locality.

Here, we ensure that the maximum number of chunks of a stripe residing in each rack (denoted by c) cannot be larger than the number of tolerable node failures f of a stripe; otherwise, a rack failure can lead to data loss. Thus, we require:

$$c \leq f. \tag{1}$$

Each of the first $z-1$ racks stores c chunks of a stripe and the last rack stores the $n-c(z-1)$ $(\leq c)$ remaining chunks.

We focus on optimizing two types of repair operations: single-chunk repair and full-node repair (§4.1). Both repair operations assume that each failed stripe has exactly one failed chunk as in most prior studies (§7), including those on parity locality [27,33,34,51,63] and topology locality [31,32,65]. For the failed stripes with multiple failed chunks, we resort to the conventional repair that retrieves k available chunks for reconstructing all failed chunks as in RS codes.

3.2 Design Idea

To achieve the objective of combined locality, we observe from Figure 2 that combined locality repairs a data chunk by downloading $r-1$ data chunks plus one local parity chunk (i.e., the repair bandwidth is r chunks). Since combined locality places some of the r chunks in identical racks, it can apply a local repair to the chunks in each rack, so as to reduce the cross-rack repair bandwidth. Intuitively, if c increases (i.e., more chunks of a stripe can reside in one rack), a local repair can include more chunks, thereby further reducing the cross-rack repair bandwidth. Thus, we aim to find the largest possible c. Recall that $c \leq f$ (Equation (1)). If $c = f$, then the cross-rack repair bandwidth can be minimized.

Thus, the construction of (n,k,r,z) CL is to ensure $c = f$. However, there are different constructions of (n,k,r) LRCs that provide different levels of fault tolerance f [35]. Thus, our idea is to select the appropriate LRC construction that has the highest fault tolerance (§3.3).

3.3 LRC Selection

We consider four representative LRCs discussed in [35].

- **Azure-LRC [34]:** It computes a local parity chunk as a linear combination of r data chunks of each local group, and computes the global parity chunks via RS codes. Note that repairing a global parity chunk needs to retrieve k chunks.

	(n,k,r)	$(16,10,5)$
Azure-LRC [34]	$f = n-k-\lceil k/r \rceil + 1$	$f = 5$
Xorbas [63]	$f \leq n-k-\lceil k/r \rceil + 1$	$f = 4$
Optimal-LRC [69]	$f \leq n-k-\lceil k/r \rceil + 1$	$f = 4$
Azure-LRC+1 [35]	$f = n-k-\lceil k/r \rceil$	$f = 4$

Table 3: Number of tolerable node failures f for different LRCs for (n,k,r) = $(16,10,5)$ [35].

- **Xorbas [63]:** It differs from Azure-LRC in that it allows each global parity chunk to be repairable by at most r chunks, which may include the other global parity chunks and the local parity chunks.
- **Optimal-LRC [69]:** It divides all data chunks and global parity chunks into local groups of size r, and adds a local parity to each local group to allow the repair of any lost chunk by at most r chunks.
- **Azure-LRC+1 [35]:** It builds on Azure-LRC by adding a new local parity chunk for all global parity chunks, allowing the local repair of any lost global parity chunk.

Table 3 shows the number of tolerable node failures f for a practical setting $(n,k,r) = (16,10,5)$ [35]. Note that Xorbas and Optimal-LRC give their upper bounds of f, but in fact the bounds are not attainable for some parameters, including (n,k,r) = $(16,10,5)$ [35]. Table 3 shows that Azure-LRC has the largest f under the same (n,k,r), so it can be the appropriate selection of LRC for combined locality.

The reason why Azure-LRC achieves the highest fault tolerance f is that it neither introduces extra local parity chunks that are linearly dependent on the global parity chunks (e.g., Optimal-LRC and Azure-LRC+1), nor makes the global parity chunks linearly dependent on the local parity chunks (e.g., Xorbas). In fact, for a given level of redundancy, adding linear dependency does not improve fault tolerance.

Note that Azure-LRC needs to download k chunks to repair a global parity chunk, which may be inefficient in repairing a failed node that stores multiple global parity chunks. Nevertheless, we argue that the number of global parity chunk accounts for a small fraction for wide stripes with a large k. For example, in the $(128,120,24)$ Azure-LRC, which contains three global parity chunks, only 3/128 = 2.34% of the chunks stored in each node are global parity chunks. Also, the cross-rack repair bandwidth of a single global parity chunk can be significantly reduced via topology locality. In our following discussion, unless otherwise specified, we focus on a single-chunk repair for a data chunk or a local parity chunk.

3.4 Construction of (n,k,r,z) CL

We provide the construction of (n,k,r,z) CL as follows. Here, we focus on one stripe that has k data chunks with a fixed number of tolerable node failures f subject to the maximum allowed redundancy γ (i.e., $\frac{n}{k} \leq \gamma$). The construction comprises two steps: (i) finding the parameters for (n,k,r) Azure-LRC, and (ii) placing all n chunks across z racks for local repair operations.

Step 1. Given (n,k,r) Azure-LRC, Table 3 states that:

$$n = k + \lceil k/r \rceil + f - 1. \tag{2}$$

Due to $\frac{n}{k} \le \gamma$, we have:

$$\lceil k/r \rceil \le k(\gamma - 1) - f + 1. \tag{3}$$

We can obtain the minimum value of r that satisfies Equation (3), denoted by r_{min}. Since r represents the single-chunk repair bandwidth (§3.2), r_{min} refers to the minimum single-chunk repair bandwidth.

Step 2. Based on r_{min}, we proceed to minimize the cross-rack repair bandwidth. First, we can obtain the value of n from Equation (2) and r_{min}. Next, we place these n chunks across n nodes that reside in z racks as follows. For each local group, we put $r+1$ chunks (note that $r = r_{min}$ here), including r data chunks and the corresponding local parity chunk, into $(r+1)/c$ different racks (for the simplicity of discussion, we assume that $r+1$ is divisible by c to have a symmetric distribution of chunks across racks). Thus, for any lost chunk in a rack, we can perform a local repair over the $(r+1)/c$ racks, such that the cross-rack repair bandwidth is $(r+1)/c-1$ chunks collected from the other $(r+1)/c-1$ racks. By setting $c = f$ to minimize the cross-rack repair bandwidth (§3.2), the minimum cross-rack repair bandwidth is $(r+1)/f-1$ chunks.

Figure 2(c) illustrates the $(26,20,5,9)$ CL with $k = 20$ and $f = 3$. Each local group of $r+1 = 6$ chunks (where $r = r_{min} = 5$) is stored in $(r+1)/f = 2$ racks. The cross-rack repair bandwidth is only one chunk (i.e., $(r+1)/f-1 = 1$).

3.5 Trade-off Analysis

Each set of the parameters (n,k,r,z) in combined locality yields the corresponding set of values of redundancy and cross-rack repair bandwidth based on the results in §3.4. We can also derive the values for Azure-LRC and topology locality in terms of k and f. For Azure-LRC, we obtain its redundancy via Equation (2) and cross-rack repair bandwidth as r chunks (assuming each chunk is stored in a distinct rack). For topology locality, we obtain its redundancy subject to $f = n-k$ and cross-rack repair bandwidth as the number of racks minus one (in chunks) (i.e., $\lceil n/f \rceil - 1$) (Figure 2(b)). Table 4 lists the redundancy and cross-rack repair bandwidth for Azure-LRC, topology locality, and combined locality, represented as (n,k,r) Azure-LRC, (n,k,z) TL, and (n,k,r,z) CL, respectively.

Figure 3 plots the results of Table 4 for $k = 128$ and $f = 2,3,4$ subject to the maximum allowed redundancy $\gamma = 1.1$. We set k as a sufficiently large value for wide stripes, and set f as in state-of-the-arts (Table 1). We set γ to close to one to achieve extreme storage savings with wide stripes.

Each point in Figure 3 represents a trade-off between redundancy and cross-rack repair bandwidth for a specific set of parameters. Note that topology locality has three points for the three respective values of f; in contrast, Azure-LRC and combined locality have three curves for the three respective values of f, since they have an additional parameter r that leads to different points along each curve for different values of r. We only plot the points that satisfy $r = r_{min}$ for the minimum cross-rack repair bandwidth.

	Redundancy	**Cross-rack repair bandwidth**
(n,k,r) Azure-LRC	$\frac{k+\lceil k/r \rceil+f-1}{k}$	r
(n,k,z) TL	$\frac{k+f}{k}$	$\lceil (k+f)/f \rceil - 1$
(n,k,r,z) CL	$\frac{k+\lceil k/r \rceil+f-1}{k}$	$(r+1)/f-1$

Table 4: Redundancy and cross-rack repair bandwidth (in chunks) given k and f for (n,k,r) Azure-LRC, (n,k,z) TL, and (n,k,r,z) CL.

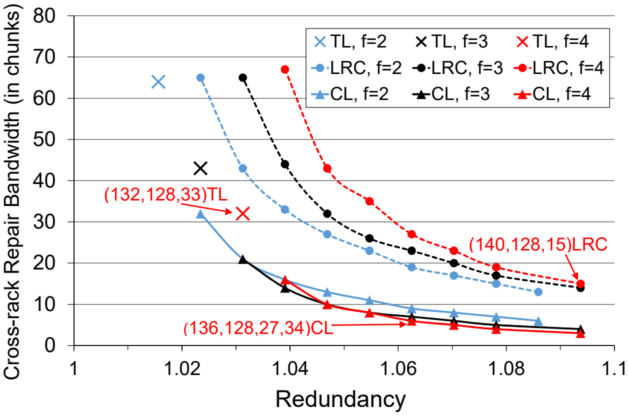


Figure 3: Trade-off between redundancy and cross-rack repair bandwidth for Azure-LRC (LRC), topology locality (TL), and combined locality (CL) for $k = 128$ and $f = 2,3,4$.

Combined locality outperforms both Azure-LRC and topology locality in terms of the trade-off between redundancy and cross-rack repair bandwidth via the combination of both parity locality and topology locality. Take $f = 4$ as an example. For topology locality, the $(132,128,33)$ TL has the minimum redundancy $1.031\times$, yet its cross-rack repair bandwidth reaches 32 chunks, even though many racks perform local repair operations. The $(140,128,15)$ Azure-LRC largely reduces the cross-rack repair bandwidth to $r = 15$ chunks via parity locality, yet its redundancy $(1.094\times)$ is not close to the minimum one. The reason is that Azure-LRC's redundancy is $\propto (1/r)$, while its cross-rack repair bandwidth is $\propto r$ (Table 4), so r should be small for small cross-rack repair bandwidth, at the expense of incurring higher redundancy. In contrast, for combined locality, the $(136,128,27,34)$ CL not only has closer redundancy (i.e., $1.063\times$) to the minimum one, but also further significantly reduces the cross-rack repair bandwidth to at most $(r+1)/f-1 = 6$ chunks (we show a more precise calculation in §3.6), a reduction of 60% compared to Azure-LRC. The reason is that the cross-rack repair bandwidth of combined locality is $\propto (r/f)$ (Table 4), so it has lower cross-rack repair bandwidth under limited redundancy.

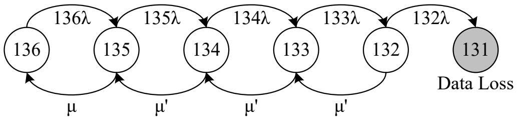


Figure 4: Markov model for (136,128,27,34) CL.

3.6 Reliability Analysis

We analyze the mean-time-to-data-loss (MTTDL) metric via Markov modeling as in prior studies [19,26,32,34,63,67]. We compare six different codes with $f = 4$: (i) $(16,12)$ RS, (ii) $(16,12,6)$ Azure-LRC, (iii) $(132,128)$ RS, (iv) $(132,128,33)$ TL, (v) $(140,128,15)$ Azure-LRC, and (vi) $(136,128,27,34)$ CL. The former two codes are moderate-stripe codes, while the latter four codes are wide-stripe codes.

Figure 4 shows the Markov model for $(136,128,27,34)$ CL; other codes are modeled similarly. Each state represents the number of available nodes of a stripe. For example, State 136 means that all nodes are healthy, while State 131 means data loss. We make two assumptions to simplify our analysis. First, we assume that data loss always occurs whenever there exist five failed nodes, yet in reality some combinations of five failed nodes remain repairable [34] (e.g., the loss of five local parity chunks). Thus, the reliability of $(136,128,27,34)$ CL is an underestimate; we make similar treatments when we model Azure-LRC. Second, we only focus on independent node failures, but do not consider rack failures with multiple nodes failing simultaneously (e.g., a power outage [19]). Our justification is that node failures are much more common than rack failures [47]. We plan to relax the assumptions in our future work.

Our reliability modeling follows the prior work [34]. Let λ be the failure rate of each node. Thus, the state transition rate from State i to State $i-1$ (where $132 \leq i \leq 136$) is $i\lambda$, since any one of the i nodes in State i fails independently. To model repair, let μ be the repair rate of a failed node from State 135 to State 136, and μ' be the repair rate for each node from State i to State $i+1$ (where $132 \leq i \leq 134$). We assume that the repair time of a single-node failure is proportional to the amount of repair traffic. Specifically, let N be the total number of nodes in a storage system, S be the capacity of each node, B be the network bandwidth of each node, and ε be the fraction of available network bandwidth of each node for repair due to rate throttling. If a single node fails, the repair load is evenly distributed over the remaining $N-1$ nodes, and the total available network bandwidth for repair is $\varepsilon(N-1)B$. Thus, we have $\mu = \varepsilon(N-1)B/(CS)$, where C is the single-node repair cost (which is derived below). If multiple nodes fail, we set $\mu' = 1/T$, where T denotes the time of detecting multiple node failures and triggering a multi-node repair, based on the assumption that the multi-node repair is prioritized over the single-node repair [34].

We compute C as the average cross-rack repair bandwidth. Take $(136,128,27,34)$ CL as an example. There exist $\lceil \frac{k}{r} \rceil = 5$ local groups, in which the first four local groups (each with $r+1 = 28$ chunks) span seven racks (i.e., the cross-rack repair bandwidth is six chunks), while the last local group (with 21 chunks) spans six racks (i.e., the cross-rack repair bandwidth is five chunks). For the remaining $n - k - \lceil \frac{k}{r} \rceil = 3$ global parity chunks (which reside in one rack), we repair each of them by accessing the other $z - 1 = 33$ racks, each of which sends one cross-rack chunk computed from an inner-rack repair sub-operation as in topology locality. Thus, we have $C = (6 \times 112 + 5 \times 21 + 33 \times 3)/136 = 6.44$ chunks.

$1/\lambda$ (years)	2	4	10
(16,12) RS	2.47e+11	7.87e+12	7.66e+14
(16,12,6) Azure-LRC	4.38e+11	1.40e+13	1.36e+15
(132,128) RS	6.33e+05	1.53e+07	1.20e+09
(132,128,33) TL	1.61e+06	4.64e+07	4.24e+09
(140,128,15) Azure-LRC	2.06e+06	6.20e+07	5.82e+09
(136,128,27,34) CL	5.82e+06	1.82e+08	1.75e+10

Table 5: MTTDLs of codes (in years) for varying $1/\lambda$ (years) and $B = 1$ Gb/s.

B (Gb/s)	0.5	1	10
(16,12) RS	3.96e+12	7.87e+12	7.83e+13
(16,12,6) Azure-LRC	7.00e+12	1.40e+13	1.39e+14
(132,128) RS	1.01e+07	1.53e+07	1.09e+08
(132,128,33) TL	2.57e+07	4.64e+07	4.20e+08
(140,128,15) Azure-LRC	3.29e+07	6.20e+07	5.85e+08
(136,128,27,34) CL	9.30e+07	1.82e+08	1.78e+09

Table 6: MTTDLs of codes (in years) for varying B (Gb/s) and $1/\lambda = 4$ years.

We configure the default parameters as follows. We set $N = 400$, $S = 16$ TB, $\varepsilon = 0.1$, and $T = 30$ minutes [34]. We also set the mean-time-to-failure $1/\lambda = 4$ years and $B = 1$ Gb/s [63]. We show the MTTDL results for varying λ (Table 5) and varying B (Table 6).

We see that $(136,128,27,34)$ CL has a lower MTTDL than $(16,12)$ RS and $(16,12,6)$ Azure-LRC with moderate stripes, but achieves a significantly higher MTTDL than other locality-based schemes for wide stripes by minimizing the cross-rack repair bandwidth for a single-node repair. For example, when $B = 1$ Gb/s and $1/\lambda = 4$ years, the MTTDL gain of $(136,128,27,34)$ CL is 10.90× of $(132,128)$ RS, 2.92× of $(132,128,33)$ TL, and 1.94× of $(140,128,15)$ Azure-LRC.

In general, combined locality achieves a higher MTTDL gain when $1/\lambda$ increases or B decreases. The former implies that multiple node failures are less probable, while the latter implies that the cross-rack bandwidth is more constrained. In either case, minimizing the cross-rack repair bandwidth for a single-node repair is critical for a high MTTDL gain.

4 Design

We design ECWide, a wide-stripe erasure-coded storage system that realizes combined locality. ECWide addresses the challenges of achieving efficient repair, encoding, and updates in wide-stripe erasure coding (§2.2), with the following goals:

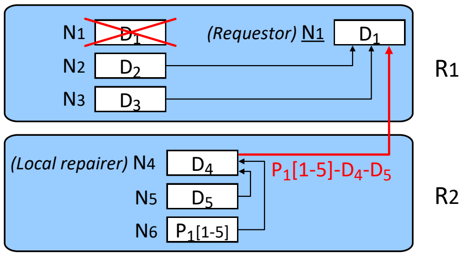


Figure 5: Repair in ECWide.

- **Minimum cross-rack repair bandwidth:** ECWide minimizes the cross-rack repair bandwidth via combined locality (§4.1).
- **Efficient encoding:** ECWide applies *multi-node encoding* that supports efficient encoding for wide stripes (§4.2).
- **Efficient parity updates:** ECWide applies *inner-rack parity updates* that allow both global and local parity chunks to be updated mostly within local racks (§4.3).

4.1 Repair

ECWide realizes combined locality for two types of repair operations: single-chunk repair and full-node repair.

Single-chunk repair. ECWide realizes two steps of combined locality in repair (§3.4). Consider a storage system that organizes data in fixed-size chunks given k, f, and γ. In Step 1, ECWide determines the parameters n and r via Equations (2) and (3). It then encodes k data chunks into $n-k$ local/global parity chunks. In Step 2, ECWide selects $(r+1)/f$ racks for each local group, and places all $r+1$ chunks of each local group into $r+1$ different nodes evenly across these racks (i.e., f chunks per rack). Since the above two steps ensure that the cross-rack repair bandwidth for a single-chunk repair is minimized as $(r+1)/f-1$ chunks (§3.4), ECWide only needs to provide the following details for the repair operation.

Figure 5 describes the repair of a lost chunk D_1 in rack R_1. Specifically, ECWide selects one node $\underline{N_1}$ (called the *requestor*) in R_1 to be responsible for reconstructing the lost chunk. It also selects one node N_4 (called the *local repairer*) in rack R_2 to perform local repair. N_4 then collects all chunks D_5 and $P_1[1\text{-}5]$ within R_2, computes an encoded chunk $P_1[1\text{-}5]-D_4-D_5$ (assuming that $P_1[1\text{-}5]$ is the XOR-sum of $D_1, D_2, \ldots, D_5$ for simplicity), and sends the encoded chunk to the requestor $\underline{N_1}$. Finally, $\underline{N_1}$ collects data chunks D_2 and D_3 within R_1, and solves for D_1 by cancelling out D_2 and D_3 from the received encoded chunk $P_1[1\text{-}5]-D_4-D_5$.

Full-node repair. A full-node repair can be viewed as multiple single-chunk repairs for multiple stripes (i.e., one lost chunk per stripe), which can be parallelized. However, each single-chunk repair involves one requestor and multiple local repairers, so multiple single-chunk repairs may choose identical nodes as requestors or local repairers, thereby overloading the chosen nodes and degrading the overall full-node repair performance. Thus, our goal is to choose as many different nodes to be requestors and local repairers as possible for effective parallelization of multiple single-chunk repairs.

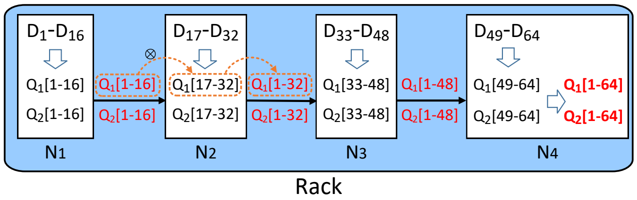


Figure 6: Multi-node encoding in ECWide.

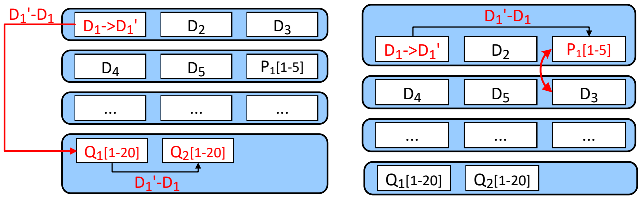
To this end, ECWide designs a least-recently-selected method to select nodes as requestors or local repairers, and implements it via a doubly-linked list and a hashmap. The doubly-linked list holds all node IDs to track which node has been recently selected or otherwise, and the hashmap holds the node ID and the node address of the list. We can then obtain the least-recently-selected node as the requestor or local repairer by simply selecting the bottom one of the list and updating the list via hashmap in O(1) time.

4.2 Encoding

Recall from §2.2 that single-node encoding for wide stripes leads to significant performance degradation for a large k. We observe that the current encoding implementation (e.g., Intel ISA-L [4] and QFS [49]) often splits data chunks of large size (e.g., 64 MiB) into smaller-size data slices and performs slice-based encoding with hardware acceleration (e.g., Intel ISA-L) or parallelism (e.g., QFS). To encode a set of k data slices that are parts of k data chunks, the CPU cache of the encoding node prefetches successive slices from each of the k data chunks. If k is large, the CPU cache may not be able to hold all prefetched slices, thereby degrading the encoding performance of the successive slices.

To overcome the limitation of single-node encoding, we consider a *multi-node encoding* scheme that aims to achieve high encoding throughput for wide-stripes. Its idea is to divide a single-node encoding operation with a large k into multiple encoding sub-operations for a small k across different nodes. It is driven by three observations: (i) the encoding performance of stripes with a small k (e.g., $k=16$) is fast (Figure 1 in §2.2); (ii) the parity chunks are linear combinations of data chunks (§2.1), so a parity chunk can be combined from multiple *partially* encoded chunks of different subsets of k data chunks; and (iii) the bandwidth among the nodes within the same rack is often abundant.

Figure 6 depicts the multi-node encoding scheme with $k=64$, assuming that two global parity chunks $Q_1[1\text{-}64]$ and $Q_2[1\text{-}64]$ are to be generated. ECWide first evenly distributes all 64 data chunks across four nodes N_1, N_2, N_3, and N_4 in the same rack. It lets each node (e.g., N_1) encode its 16 local data chunks (e.g., $D_1, D_2, \ldots, D_{16}$) into two partially encoded chunks (e.g., $Q_1[1\text{-}16]$ and $Q_2[1\text{-}16]$). The first node N_1 sends $Q_1[1\text{-}16]$ and $Q_2[1\text{-}16]$ to its next node N_2. N_2 combines the



(a) Global parity updates (b) Local parity updates

Figure 7: Inner-rack parity updates in ECWide.

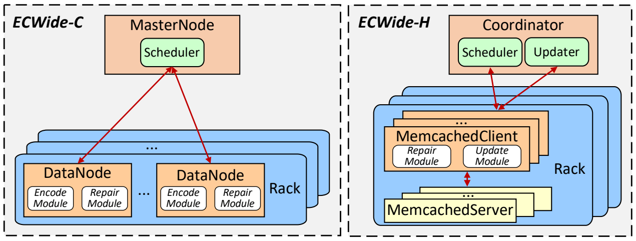


Figure 8: Architecture of ECWide.

two received partially encoded chunks with its local partially encoded chunks $Q_1[17\text{-}32]$ and $Q_2[17\text{-}32]$ to form two new partially encoded chunks $Q_1[1\text{-}32]$ and $Q_2[1\text{-}32]$, which are sent to the next node N_3. Similar operations are performed in N_3 and N_4. Finally, N_4 generates the final global parity chunks $Q_1[1\text{-}64]$ and $Q_2[1\text{-}64]$. Note that the partially encoded chunks are encoded in parallel and forwarded from N_1 to N_4 via fast inner-rack links, so as to efficiently calculating the global parity chunks of wide stripes.

ECWide needs to generate local parity chunks under combined locality, yet the local parity chunks can be more efficiently encoded from r data chunks of each local group in a single node, as r is typically much smaller than k. In addition, ECWide needs to distribute all data chunks, local parity chunks, and global parity chunks to different racks. Such a distribution incurs cross-rack data transfers; minimizing the cross-rack data transfers for the encoding of wide stripes is our future work.

4.3 Updates

To alleviate the expensive parity update overhead in wide stripes (§2.2), we present an *inner-rack parity update* scheme for wide stripes. Its idea is to limit both global and local parity updates within the same rack as much as possible, so as to mitigate cross-rack data transfers.

Figure 7(a) depicts how to perform inner-rack parity updates for two global parity chunks $Q_1[1\text{-}20]$ and $Q_2[1\text{-}20]$ (also shown in Figure 2(c)). ECWide places all the global parity chunks $Q_1[1\text{-}20]$ and $Q_2[1\text{-}20]$ in the same rack, which is always feasible without violating rack-level tolerance given that $c = f$ and the number of global parity chunks is often no more than f. In this case, when a data chunk D_1 is updated to D_1', ECWide first transfers a *delta chunk* $D_1' - D_1$ across racks for the global chunk $Q_1[1\text{-}20]$ (§2.1). It updates $Q_1[1\text{-}20]$ by adding $\alpha(D_1' - D_1)$, where α is the encoding coefficient of D_1 in $Q_1[1\text{-}20]$. ECWide updates the other global parity chunk $Q_2[1\text{-}20]$ by transferring the delta chunk only via inner-rack data transfers. Note that ECWide only incurs one cross-rack transferred chunk for updating all global parity chunks.

Figure 7(b) depicts how to perform inner-rack parity updates for the local parity chunk $P_1[1\text{-}5]$. For each stripe, ECWide first records the update frequency of data chunks of each rack and finds the most update-intensive rack for each local group. If $P_1[1\text{-}5]$ does not reside in the most update-intensive rack of its group, then ECWide swaps $P_1[1\text{-}5]$ for a random data chunk (say D_3) in the most update-intensive rack. In this way, $P_1[1\text{-}5]$ is moved to the rack that is the most update-intensive, so that the local parity updates can mostly be performed within the rack without incurring cross-rack data transfers.

If a data chunk is updated, it is important to ensure that all global and local parity chunks of the same stripe are consistently updated. ECWide may handle consistent parity updates in a state-of-the-art manner, for example, by leveraging a piggybacking method to improve the classical two-phase commits as one-phase commits [74].

5 Implementation

We implement ECWide (§4) in three major modules: a *repair* module that performs the repair operations based on combined locality, an *encode* module that performs multi-node encoding, and an *update* module that performs inner-rack parity updates. We implement two prototypes of ECWide, namely ECWide-C and ECWide-H, for cold and hot storage systems, respectively, as shown in Figure 8.

ECWide-C. ECWide-C is mainly implemented in Java with about 1,500 SLoC, while the encode module is implemented in C++ with about 300 SLoC based on Intel ISA-L [4]. It has a `MasterNode` that stores metadata and organizes the repair and encoding operations with a `Scheduler` daemon, as well as multiple `DataNodes` that store data and perform the repair and multi-node encoding operations. Note that ECWide-C does not consider the update module, assuming that updates are rare in cold storage.

For the repair operation, the `Scheduler` triggers the repair module of each `DataNode` that serves as a local repairer. Such `DataNodes` (which serve as local repairers) send the partially repaired results to the `DataNode` that serves as the requestor, which finally reconstructs the lost chunk. For the encoding operation, the `Scheduler` selects a rack and triggers the encode module of each involved `DataNode` in the rack. Those involved `DataNodes` perform multi-node encoding, and the `DataNode` that serves as the destination node generates all global parity chunks.

ECWide-H. ECWide-H builds on the Memcached in-memory key-value store (v1.4) [6] and libMemcached (v1.0.18) [5] for hot storage. It is implemented in C with about 3,000 SLoC. It follows a client-server architecture. It contains

MemcachedServers that store key-value items, as well as MemcachedClients that perform the repair and parity update operations. It also includes the Coordinator for managing metadata. The Coordinator includes a Scheduler daemon that coordinates the repair and parity update operations and an Updater daemon that analyzes the update frequency status. Note that ECWide-H does not include the encode module as in ECWide-C, since the chunk size in erasure-coded in-memory key-value stores is often small (e.g., 4 KiB [18, 73, 74]) and a single-node CPU cache is large enough to prefetch all data chunks of a wide stripe for high encoding performance (§4.2).

For the repair operation, ECWide-H performs the same way as ECWide-C, except that it uses MemcachedClients as local repairers. For the updates of global parity chunks, the Scheduler locates the rack where the global parity chunks reside, and triggers the update modules of MemcachedClients to perform the inner-rack parity updates. For the updates of local parity chunks, the Updater first triggers the swapping, in which the two involved MemcachedClients exchange the corresponding chunks. The inner-rack parity updates for the local parity chunks can be later performed. Note that some existing in-memory systems (e.g., Cocytus [74]) also deploy multiple Memcached instances in a single physical node and have a form of hierarchical topology that is suitable for topology locality.

6 Evaluation

We conduct our experiments on Amazon EC2 [1] with a number of m5.xlarge instances connected by a 10 Gb/s network. One instance represents a MasterNode for ECWide-C or a Coordinator for ECWide-H (§5), while the other instances represent the DataNodes for ECWide-C or the MemcachedClients/MemcachedServers for ECWide-H. To simulate the heterogeneous bandwidth within a rack and across racks, we partition nodes into logical racks and assign one dedicated instance as a *gateway* in each rack. The instances within the same logical rack can communicate directly via the 10 Gb/s network, while the instances in different racks communicate via the gateways. We use the Linux traffic control command tc [7] to limit the outgoing bandwidth of each gateway to make cross-rack bandwidth constrained. In our experiments, we vary the gateway bandwidth from 500 Mb/s up to 10 Gb/s.

We set the chunk size as 64 MiB for ECWide-C and 4 KiB for ECWide-H (§2.2). We plot the average results of each experiment over ten runs. We also plot the error bars for the minimum and maximum results over the ten runs. Note that the error bars may be invisible in some plots due to the small variance.

We present the experimental results of ECWide-C and ECWide-H for combined locality (CL), compared with Azure-LRC (LRC) and topology locality (TL) that represent state-of-the-art locality-based schemes. We show that CL outperforms LRC and TL for both single-chunk repair and full-node repair. We also show the efficiency of our multi-node encoding and inner-rack parity update schemes.

6.1 ECWide-C Performance

Experiment A.1 (Repair). We evaluate the repair performance of LRC, TL, and CL using ECWide-C. Here, we let $32 \leq k \leq 64$ and $2 \leq f \leq 4$, and configure different gateway bandwidth settings. For (n,k,r,z) CL, we deploy $n+1$ instances, including n instances as DataNodes and one instance as MasterNode. We select two types of LRC and two types of CL for each set of f and k with different r. We also compute the corresponding redundancy of each scheme based on Table 4. Given k, f, and r, we can compute $n = k + \lceil \frac{k}{r} \rceil + f - 1$ and $z = \lceil \frac{n}{f} \rceil$. Thus, in the following discussion, we only show the values of k, f, and r.

Figures 9(a)-9(e) show the average single-chunk repair times of LRC, TL, and CL for different values of k and f, under the gateway bandwidth of 1 Gb/s and 500 Mb/s. CL always outperforms LRC and TL under the same k, f, and the gateway bandwidth, while TL with the minimum redundancy often performs the worst. For example, in Figure 9(c), when the gateway bandwidth is 1 Gb/s, the single-chunk repair time of CL with $r = 7$ is 0.8 s, while those of LRC with $r = 7$ and TL are 3.9 s and 9.0 s, respectively; equivalently, CL reduces the single-chunk repair times of LRC and TL by 79.5% and 91.1%, respectively.

CL shows a higher gain compared to LRC under smaller gateway bandwidth. For example, in Figure 9(c), when the gateway bandwidth is 500 Mb/s, the gain of CL over LRC is 82.1%, which is higher than 79.5% when the gateway bandwidth is 1 Gb/s. The reason is that CL minimizes the cross-rack repair bandwidth, so its performance gain is more obvious when the gateway bandwidth is more constrained.

Also, the single-chunk repair time of CL increases when only r increases (see $r = 7$ and $r = 11$ in Figure 9(c)), and keeps stable when only k changes (see Figure 9(a)-9(c)). The empirical results are consistent with the theoretical results in Table 4, as the single-chunk cross-rack repair bandwidth is equal to $(r+1)/f - 1$.

Figure 9(f) shows the average full-node repair rates of LRC, TL, and CL for different values of f; we also compare CL with and without the least-recently-selected (LRS) method (§4.1). We fix $k = 64$, $r = 11$, and the gateway bandwidth as 1 Gb/s. To mimic a single node failure, we erase 64 chunks from 64 stripes (i.e., one chunk per stripe) in one node. We then repair all the erased chunks simultaneously. Note that practical storage systems often store many more chunks per node, yet each chunk of the failed node is independently associated with one stripe. Thus, we expect that using 64 chunks sufficiently provides stable performance. From the figure, we see that CL shows a higher full-node repair rate than TL and LRC. Its full-node repair rate increases with f, as the single-chunk cross-rack repair bandwidth is equal to $(r+1)/f - 1$. Also, CL

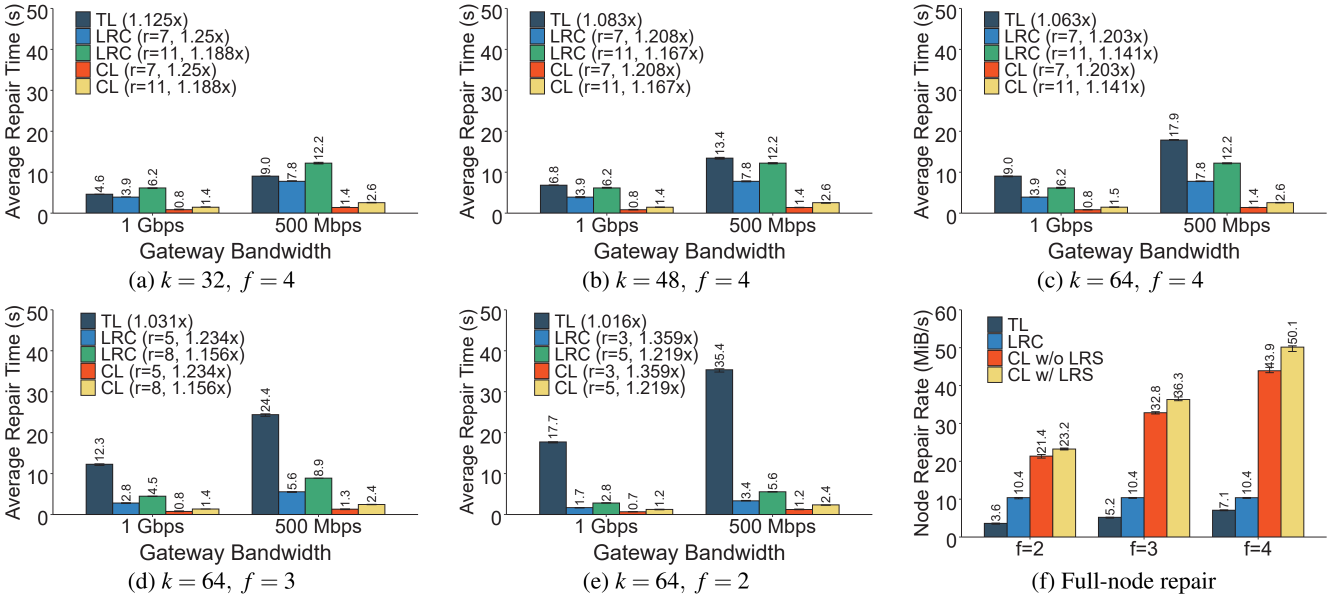


Figure 9: Experiment A.1: Average single-chunk repair time (in seconds) for different k and f under the gateway bandwidth of 1 Gb/s and 500 Mb/s (figures (a)-(e)), and average full-node repair rate for different f (figure (f)).

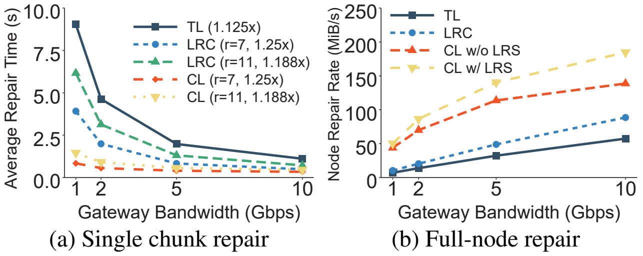


Figure 10: Experiment A.1: Single-chunk repair time and full-node repair rate for different gateway bandwidth.

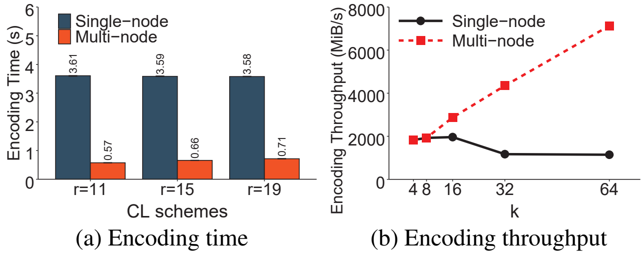


Figure 11: Experiment A.2: Encoding time and encoding throughput for single-node encoding and multi-node encoding.

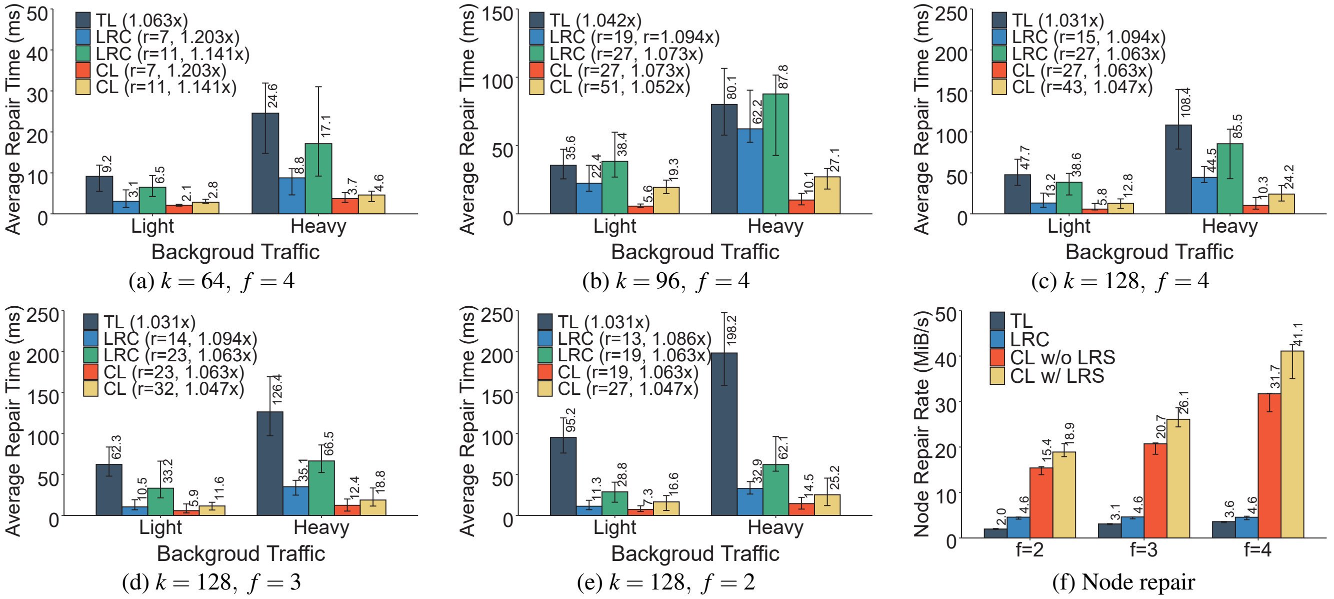
with LRS increases the full-node repair rate by 14.1% when $f = 4$ compared to CL without LRS, thereby demonstrating the efficiency of the LRS method.

Finally, Figure 10 shows how the average single-chunk repair time and the average full-node repair rate vary with the gateway bandwidth, ranging from 1 Gb/s to 10 Gb/s. Here, we fix $k = 64$ and $f = 4$. From Figure 10(a), CL still outperforms LRC and TL in single-chunk repair under all gateway bandwidth settings, although the difference becomes smaller as the gateway bandwidth increases. For example, when the gateway bandwidth is 10 Gb/s, the single-chunk repair time of CL with $r = 7$ (0.34 s) reduces those of LRC with $r = 7$ (0.49s) and TL (1.11s) by 30.6% and 69.4%, respectively. Also, from Figure 10(b), CL maintains its performance gain in full-node repair over LRC and TL, and LRS brings further improvements.

One limitation of our current implementation is that the full-node repair performance is not fully optimized. We can further improve the throughput by state-of-the-art repair parallelization techniques, such as parity declustering [30], PPR [46], and repair pipelining [41]. Nevertheless, all coding schemes are fairly evaluated under the same implementation setting. If we use parallelization techniques for all coding schemes, we expect that CL should maintain its performance gain by reducing the cross-rack repair bandwidth and I/O. We pose this issue as future work.

Experiment A.2 (Encoding). We measure the average encoding time of CL per stripe. Here, we fix $k = 64$ and $f = 4$, and let $11 \leq r \leq 19$. Figure 11(a) shows the results of single-node encoding and multi-node encoding. We see that multi-node encoding shows significantly lower encoding time than single-node encoding. For example, when $r = 11$, multi-node encoding reduces 84% of the encoding time compared to single-node encoding.

We further measure the average encoding throughput. Here, we fix $4 \leq k \leq 64$ and $f = 4$. Figure 11(b) shows the results of single-node encoding and multi-node encoding. Multi-node encoding achieves significantly high encoding throughput when k is large, since many nodes in the same rack can share their computational resources to accelerate the encoding operation. On the other hand, single-node encoding has low throughput when k is large, consistent with our findings in



(a) $k = 64,\ f = 4$

(b) $k = 96,\ f = 4$

(c) $k = 128,\ f = 4$

(d) $k = 128,\ f = 3$

(e) $k = 128,\ f = 2$

(f) Node repair

Figure 12: Experiment B.1: Average single-chunk repair time (in milliseconds) for different k and f under different types of background traffic (figures (a)-(e)), and average full-node repair rate for different f.

Figure 1. Note that Figure 1 shows higher throughput than Figure 11(b), since the former only considers the computation part of the encoding operations, while the latter also includes disk I/O for reading chunks for encoding in addition to encoding computation.

6.2 ECWide-H Performance

Experiment B.1 (Repair). We evaluate the repair performance of LRC, TL, and CL using ECWide-H. For (n,k,z) TL and (n,k,r,z) CL, we deploy $n+8z+1$ instances, including n instances as `MemcachedServers`, $8z$ instances as `MemcachedClients` (with eight instances in each of the z racks), and one instance as `Coordinator`.

We consider two deployment scenarios of ECWide-H with different loads of background traffic. To mimic background traffic, in addition to the existing `MemcachedClients` in ECWide-H, we add extra Memcached clients in the background (called *background clients*) that continuously issue read requests to `MemcachedServers`. Specifically, we consider a case of *light* background traffic where each `MemcachedServer` serves 20 background clients, and a case of *heavy* background traffic where each `MemcachedServer` serves 80 background clients.

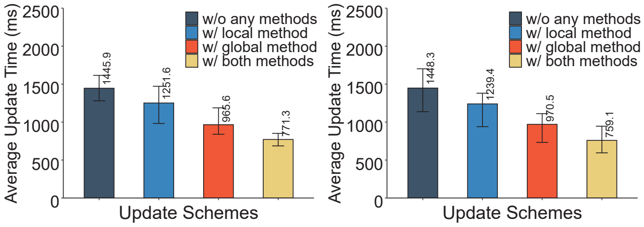
Figures 12(a)-12(e) show the average single-chunk repair times of LRC, TL, and CL for different k and f under the light and heavy background traffic loads. Here, we let $64 \leq k \leq 128$ and $2 \leq f \leq 4$. As in Experiment A.1 (§6.1), we select two types of LRC and two types of CL for each set of k and f with different r. We also compute the corresponding redundancy of each scheme based on Table 4. Similar to Experiment A.1, we see that CL still performs the best in hot storage workloads. For example, in Figure 12(c) under heavy background traffic, the single-chunk repair time of CL with $r = 27$ is 10.3 ms, while those of LRC with $r = 27$ and TL are 85.5 ms and 108.4 ms, respectively; equivalently, CL reduces the single-chunk repair times of LRC and TL by 87.9% and 90.5%, respectively. Also, CL with $r = 27$ only incurs a redundancy of 1.063×, close to the minimum redundancy of TL (1.031×).

In addition, CL under heavy background traffic shows a higher performance gain compared to the light one, similar to Experiment A.1 that compares the gateway bandwidth of 500 Mb/s to that of 1 Gb/s. The reason is that the single-chunk repair performance is more likely bottlenecked by the limited available bandwidth under heavy background traffic, in which the performance gain of CL is more prominent.

Figure 12(f) shows the average full-node repair rate for different values of f. Here, we fix $k = 64$ and $r = 7$, and focus on light background traffic. From the figure, CL shows a higher full-node repair rate than LRC and TL, and CL with LRS increases the full-node repair rate by 29.7% when $f = 4$ compared to CL without LRS. Also, the full-node repair rate of CL increases with f, consistent with the results in Figure 9(f) (see Experiment A.1 in §6.1) for the same reason.

Experiment B.2 (Updates). We evaluate the update time of a chunk with and without the inner-rack parity updates for global and local parity chunks using (136,128,27,34) CL (§4.3); without the inner-rack parity updates, we assume that each parity chunk is updated directly by the corresponding delta chunk. We use workloads generated by Yahoo! Cloud Serving Benchmark (YCSB) [21] with two read-to-update ratios, namely read-mostly (95%:5%) and update-intensive (50%:50%).

Figure 13 shows the average update times of different update schemes. Compared to without inner-rack parity updates,



(a) read:update = 95%:5%

(b) read:update = 50%:50%

Figure 13: Experiment B.2: Average update time (in milliseconds) with inner-rack parity update methods for local and global parity chunks under different read/update ratios, using (136,128,27,34) CL.

the inner-rack parity updates for global parity chunks reduce the update time by up to 33.2%, the inner-rack parity updates for local parity chunks reduce the update time by up to 14.4%, and the inner-rack parity updates for both local and global parity chunks reduce the update time by up to 47.6%.

7 Related Work

Wide-stripe erasure coding. Wide stripes have been commercially adopted (e.g., VAST [9]), but little is known about the design details, and there is no rigorous analysis on the fundamental properties of wide-stripe erasure coding in real deployment. Some studies in the literature address the wide-stripe problem from different perspectives. Li et al. [42] address the read-retry problem in hard disks using local erasure coding, where $n = 1024$ and $n - k \leq 20$. Haddock et al. [28] use general-purpose GPUs to improve encoding/decoding efficiency, where $n = 24$ and $k = 20$. Some studies consider large stripes, where both k and $n - k$ are large, for distributed storage using low-density parity-check (LDPC) codes [54] or rateless codes [44], but the minimum redundancy that they consider (e.g., 1.167× [44] and 1.5× [54]) remains higher than that in our wide-stripe problem. Our work is the first to systematically study the performance issues in wide-stripe erasure coding, including repair, encoding, and updates.

Erasure coding in distributed storage. Erasure coding has been widely studied in distributed storage (see surveys [11, 52]). As erasure coding has higher performance overhead than replication, it is often used in cold storage that treats data persistence as a first-class citizen as opposed to access performance [10, 12]. To ensure data reliability, fast repair is critical for erasure coding in cold storage. Most studies address the repair issue via either proposing new erasure codes that minimize the repair bandwidth [23, 34, 50, 58, 60, 63, 71], or designing repair-efficient techniques that mitigate the repair time [41, 46]. Erasure coding is also considered in hot storage that requires high data access performance. One notable example is erasure coding in in-memory key-value storage, in which existing studies mainly address caching [57], data management [43, 73, 74], and consistent hashing [18, 70]. Some studies focus on updates in erasure-coded storage, and mainly address performance [16, 37] and consistency [17]. Existing studies on erasure coding mainly focus on small k and m. In contrast, we focus on the application of wide stripes in both cold and hot storage.

Locality in erasure coding. Many studies exploit either parity locality or topology locality to improve the performance of erasure coding. In terms of parity locality, *locally repairable codes* [27, 33, 34, 51, 63] reduce the repair bandwidth and I/O costs by associating local parity chunks with different groups of fewer than k data chunks. *Product codes* [24, 29, 40] associate local parities with both horizontal and vertical groups of data chunks for high fault tolerance. Several studies exploit *hierarchical parity locality* to associate local parity chunks with different levels of groups of data chunks to handle multiple failures [38, 62]. In terms of topology locality, existing studies exploit rack-level locality to reduce cross-rack data transfers in repair or update operations. Some studies propose repair-optimal erasure code constructions [31,32,56] that minimize the cross-rack repair bandwidth, while the others design new techniques for efficient repair [65, 66] or updates [64]. Our work combines both parity locality and topology locality to solve the wide-stripe problem, especially on reducing the repair bandwidth for wide stripes.

8 Conclusions

Wide stripes are a new notion for erasure-coded distributed storage to achieve extreme storage savings. We propose combined locality, a novel repair mechanism that combines parity locality and topology locality to address the repair problem effectively for wide stripes. We design ECWide, a prototype system that realizes combined locality. We further design multi-node encoding and inner-rack parity updates to improve the encoding and update performance, respectively. We implement ECWide for both cold and hot storage systems, and our Amazon EC2 experiments demonstrate the efficiency of ECWide in repair, encoding, and updates.

Acknowledgement

We thank our shepherd, Cheng Huang, and the anonymous reviewers for their comments. This work was supported by National Natural Science Foundation of China (61872414), Key Laboratory of Information Storage System Ministry of Education of China, and the Research Grants Council of Hong Kong (AoE/P-404/18). The corresponding author is Yuchong Hu.

References

[1] Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`, Retrieved in Jan 2021.

[2] Amazon S3 Glacier & S3 Glacier Deep Archive. `https://aws.amazon.com/glacier/`, Retrieved in Jan 2021.

[3] HDFS erasure coding. `https://hadoop.apache.org/docs/current/hadoop-project-`

dist/hadoop-hdfs/HDFSErasureCoding.html, Retrieved in Jan 2021.

[4] Intel ISA-L. https://github.com/intel/isa-l, Retrieved in Jan 2021.

[5] libMemcached. https://libmemcached.org/libMemcached.html, Retrieved in Jan 2021.

[6] Memcached. https://memcached.org, Retrieved in Jan 2021.

[7] tc. https://linux.die.net/man/8/tc, Retrieved in Jan 2021.

[8] Tencent ultra-cold storage system optimization with Intel ISA-L - a case study. https://software.intel.com/content/www/us/en/develop/articles/tencent-ultra-cold-storage-system-optimization-with-intel-isa-l-a-case-study.html, Retrieved in Jan 2021.

[9] VastData. https://vastdata.com/providing-resilience-efficiently-part-ii/, Retrieved in Jan 2021.

[10] F. André, A.-M. Kermarrec, E. Le Merrer, N. Le Scouarnec, G. Straub, and A. Van Kempen. Archiving cold data in warehouses with clustered network coding. In *Proc. of ACM Eurosys*, page 21, 2014.

[11] S. B. Balaji, M. N. Krishnan, M. Vajha, V. Ramkumar, B. Sasidharan, and P. V. Kumar. Erasure coding for distributed storage: An overview. *CoRR*, abs/1806.04437, 2018. http://arxiv.org/abs/1806.04437.

[12] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron. Pelican: A building block for exascale cold data storage. In *Proc. of USENIX OSDI*, 2014.

[13] B. Beach. Backblaze Vaults: Zettabyte-Scale Cloud Storage Architecture. https://www.backblaze.com/blog/vault-cloud-storage-architecture/, 2017.

[14] M. Blaum, J. L. Hafner, and S. Hetzler. Partial-MDS codes and their application to RAID type of architectures. *IEEE Trans. on Information Theory*, 59(7):4510–4519, 2013.

[15] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, UC Berkeley, Aug 1995.

[16] J. C. Chan, Q. Ding, P. P. C. Lee, and H. H. Chan. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *Proc. of USENIX FAST*, pages 163–176, 2014.

[17] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure coding objects across global data centers. In *Proc. of USENIX ATC*, pages 539–551, 2017.

[18] L. Cheng, Y. Hu, and P. P. C. Lee. Coupling decentralized key-value stores with erasure coding. In *Proc. of ACM SoCC*, pages 377–389, 2019.

[19] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *Proc. of USENIX ATC*, pages 31–43, 2015.

[20] Cisco Systems. Oversubscription and density best practices. https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/storage-networking-solution/net_implementation_white_paper0900aecd800f592f.html, 2015.

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

[22] H. Dau, I. M. Duursma, H. M. Kiah, and O. Milenkovic. Repairing Reed-Solomon codes with multiple erasures. *IEEE Trans. on Information Theory*, 64(10):6567–6582, 2018.

[23] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, Sep 2010.

[24] K. S. Esmaili, L. Pamies-Juarez, and A. Datta. CORE: Cross-object redundancy for efficient data repair in storage systems. In *Proc. of IEEE BigData*, 2013.

[25] A. Fikes. Storage architecture and challenges. http://cloud.google.com/files/storage_architecture_and_challenges.pdf, 2010.

[26] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, Oct 2010.

[27] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin. On the locality of codeword symbols. *IEEE Trans. on Information theory*, 58(11):6925–6934, 2012.

[28] W. Haddock, P. V. Bangalore, M. L. Curry, and A. Skjellum. High performance erasure coding for very large stripe sizes. In *Proc. of IEEE SpringSim*, pages 1–12, 2019.

[29] J. L. Hafner. HoVer erasure codes for disk arrays. In *Proc. of IEEE/IFIP DSN*, 2006.

[30] M. Holland, G. A. Gibson, and D. P. Siewiorek. Architectures and algorithms for on-line failure recovery in

redundant disk arrays. *Distributed Parallel Databases*, 2(3):295–335, 1994.

[31] H. Hou, P. P. C. Lee, K. W. Shum, and Y. Hu. Rack-aware regenerating codes for data centers. *IEEE Trans. on Information Theory*, 65(8):4730–4745, 2019.

[32] Y. Hu, X. Li, M. Zhang, P. P. C. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal repair layering for erasure-coded data centers: From theory to practice. *ACM Trans. on Storage*, 13(4):33, 2017.

[33] C. Huang, M. Chen, and J. Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *ACM Trans. on Storage*, 9(1):3, Mar 2013.

[34] 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 ATC*, Jun 2012.

[35] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg. On fault tolerance, locality, and optimality in locally repairable codes. In *Proc. of USENIX ATC*, pages 865–877, 2018.

[36] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's key-value storage system for cloud data. In *Proc. of IEEE MSST*, pages 1–14, 2015.

[37] H. Li, Y. Zhang, Z. Zhang, S. Liu, D. Li, X. Liu, and Y. Peng. PARIX: Speculative partial writes in erasure-coded systems. In *Proc. of USENIX ATC*, 2017.

[38] J. Li and B. Li. Beehive: Erasure codes for fixing multiple failures in distributed storage systems. *IEEE Trans. on Parallel and Distributed Systems*, 28(5):1257–1270, 2016.

[39] M. Li, R. Li, and P. P. C. Lee. Relieving both storage and recovery burdens in big data clusters with R-STAIR codes. *IEEE Internet Computing*, 22(4):15–26, 2018.

[40] M. Li, J. Shu, and W. Zheng. GRID codes: Strip-based erasure codes with high fault tolerance for storage systems. *ACM Trans. on Storage*, 4(4):1–22, 2009.

[41] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, pages 567–579, 2017.

[42] Y. Li, H. Wang, X. Zhang, N. Zheng, S. Dahandeh, and T. Zhang. Facilitating magnetic recording technology scaling for data center hard disk drives through filesystem-level transparent local erasure coding. In *Proc. of USENIX FAST*, 2017.

[43] W. Litwin, R. Moussa, and T. Schwarz. LH*RS: A highly-available scalable distributed data structure. *ACM Trans. on Database Systems*, 30(3):769–811, 2005.

[44] M. Luby, R. Padovani, T. J. Richardson, L. Minder, and P. Aggarwal. Liquid cloud storage. *ACM Trans. on Storage*, 15(1):2, 2019.

[45] P. Luse and K. Greenan. Swift object storage: Adding erasure code. *SNIA Education September*, 2014.

[46] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proc. of ACM Eurosys*, page 30. ACM, 2016.

[47] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's Warm BLOB Storage System. In *Proc. of USENIX OSDI*, pages 383–398, 2014.

[48] P. Narayanan, S. Samal, and S. Nanniyur. Yahoo cloud object store-object storage at exabyte scale. `https://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at`, 2017.

[49] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. In *Proc. of ACM VLDB*, 2013.

[50] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Guyot, E. E. Gad, and Z. Bandic. Opening the chrysalis: On the real repair performance of MSR codes. In *Proc. of USENIX FAST*, 2016.

[51] D. S. Papailiopoulos and A. G. Dimakis. Locally repairable codes. *IEEE Trans. on Information Theory*, 60(10):5843–5855, Oct 2014.

[52] J. S. Plank and C. Huang. Tutorial: Erasure coding for storage applications. Slides presented at USENIX FAST 2013, Feb 2013.

[53] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. OHearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proc. of USENIX FAST*, 2009.

[54] J. S. Plank and M. G. Thomason. A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In *Proc. of IEEE/IFIP DSN*, 2004.

[55] J. S. Plank and L. Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *Proc. of IEEE NCA*, pages 173–180, 2006.

[56] N. Prakash, V. Abdrashitov, and M. Médard. The storage versus repair-bandwidth trade-off for clustered storage systems. *IEEE Trans. on Information Theory*, 64(8):5783–5805, 2018.

[57] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proc. of USENIX OSDI*, pages 401–417, 2016.

[58] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *Proc. of USENIX FAST*, 2015.

[59] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proc. of USENIX HotStorage*, 2013.

[60] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proc. of ACM SIGCOMM*, 2014.

[61] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[62] B. Sasidharan, G. K. Agarwal, and P. V. Kumar. Codes with hierarchical locality. In *Proc. of IEEE ISIT*, pages 1257–1261, 2015.

[63] 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 ACM VLDB Endowment*, pages 325–336, 2013.

[64] Z. Shen and P. P. C. Lee. Cross-rack-aware updates in erasure-coded data centers: Design and evaluation. *IEEE Trans. on Parallel and Distributed Systems*, 31(10):2315–2328, Oct 2020.

[65] Z. Shen, P. P. C. Lee, J. Shu, and W. Guo. Cross-rack-aware single failure recovery for clustered file systems. *IEEE Trans. on Dependable and Secure Computing*, 17(2):248–261, Mar/Apr 2020.

[66] Z. Shen, J. Shu, Z. Huang, and Y. Fu. ClusterSR: Cluster-aware scattered repair in erasure-coded storage. In *Proc. of IEEE IPDPS*, pages 42–51. IEEE, 2020.

[67] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proc. of ACM SYSTOR*, pages 1–7, 2014.

[68] J.-y. Sohn, B. Choi, S. W. Yoon, and J. Moon. Capacity of clustered distributed storage. *IEEE Trans. Information Theory*, 65(1):81–107, 2018.

[69] I. Tamo and A. Barg. A family of optimal locally recoverable codes. *IEEE Trans. on Information Theory*, 60(8):4661–4676, 2014.

[70] K. Taranov, G. Alonso, and T. Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *Proc. of ACM EuroSys*, page 39, 2018.

[71] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, S. Hussain, and S. Nandi. Clay codes: moulding MDS codes to yield an MSR code. In *Proc. of USENIX FAST*, page 139, 2018.

[72] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of USENIX OSDI*, pages 307–320, 2006.

[73] M. M. Yiu, H. H. Chan, and P. P. C. Lee. Erasure coding for small objects in in-memory KV storage. In *Proc. of ACM SYSTOR*, page 14, 2017.

[74] H. Zhang, M. Dong, and H. Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *Proc. of USENIX FAST*, pages 167–180, 2016.

On the Feasibility of Parser-based Log Compression in Large-Scale Cloud Systems

Junyu Wei[†], Guangyan Zhang[†,*] Yang Wang[‡], Zhiwei Liu[¶], Zhanyang Zhu[†],
Junchao Chen[†], Tingtao Sun[§], Qi Zhou[§]
[†]*Tsinghua University*, [‡]*The Ohio State University*, [¶]*China University of Geosciences*, [§]*Alibaba Cloud*

Abstract

Given the tremendous scale of today's system logs, compression is widely used to save space. While parser-based log compressor reported promising results, we observe less intriguing performance when applying it to our production logs.

Our detailed analysis shows that, first, some problems are caused by a combination of sub-optimal implementation and assumptions that do not hold on our large-scale logs. We address these issues with a more efficient implementation. Furthermore, our analysis reveals new opportunities for further improvement. In particular, numerical values account for a significant percentage of space and classic compression algorithms, which try to identify duplicate bytes, do not work well on numerical values. We propose three techniques, namely delta timestamps, correlation identification, and elastic encoding, to further compress numerical values.

Based on these techniques, we have built LogReducer. Our evaluation on 18 types of production logs and 16 types of public logs shows that LogReducer achieves the highest compression ratio in almost all cases and on large logs, its speed is comparable to the general-purpose compression algorithm that targets a high compression ratio.

1 Introduction

Most systems log internal events for various reasons, such as diagnosing system errors [9, 36, 48], profiling user behaviors [10, 11, 28], modeling system performance [2, 15], and detecting potential security problems [12, 37].

In today's datacenters, the size of such logs can grow large. In 2016, Feng. et al reported their system can generate 100GB of logs per day [13], and in 2019, this number increased to 2TB per day [30]. AliCloud, a major cloud provider and our collaborator, can generate several PBs of logs per day.

These logs usually need to be stored for a long time for multiple reasons: sometimes an anomaly is detected much later than it was logged, so the developer needs to analyze the past logs [1, 9, 21, 24]; certain analysis may require statistics over a long period of time to generate a conclusion [10, 14, 45]; for the purpose of the audition, local laws require a cloud provider to store these logs for a certain amount of time. As a result, AliCloud has decided to store its logs for 180 days. Considering it's generating several PBs of logs per day, storing these logs is a considerable overhead even for a big company.

To reduce log size, a classic solution is to compress these logs. General-purpose lossless compression methods, such as LZMA [40], gzip [6], PPMd [4], and bzip [41], can compress a file by identifying and replacing duplicate bytes. A number of recent works observe that most system logs are generated by adding variables to a string template (e.g. printf("value=%d", v)), and thus by separating them, they only need to store the variables [23, 30, 31, 33, 46]. We call these approaches *parser-based log compression* in this paper.

While these works report promising results, we observe less intriguing performance when applying this method to production logs from AliCloud: when applying Logzip [30], the latest one in this line of work, to our logs, we find it's seven times slower than LZMA, a general-purpose compression method targeting a high compression ratio, and Logzip's compression ratio is worse than LZMA on 13 out of the 18 types of the logs.

To understand whether such problems are fundamental or due to engineering issues, we perform a detailed analysis of Logzip. Our analysis shows that, first, some problems are indeed caused by a combination of sub-optimal implementation and undesirable limits: Logzip is implemented in Python and uses several notoriously slow libraries and data structures like Pandas DataFrame [5]; Logzip limits a log entry to have no more than 5 variables, which is too small for our logs; increasing the limit will further slow down Logzip, which is already seven times slower than LZMA. We address these issues by re-implementing the whole algorithm in C/C++, which significantly improves the compression speed. It further allows us to remove the limit on the number of variables to improve the compression ratio as well.

Second, our analysis reveals new opportunities for further

[*] Corresponding author: gyzh@tsinghua.edu.cn

improvement. At a high level, Logzip uses general-purpose compression methods to further compress variables: while this works well for string variables, it does not work well for numerical variables, since general-purpose compression methods target finding duplicate bytes, and there are not much duplication in numerical data in our experiments. We incorporate three techniques to further compress numerical data:

- We observe timestamps account for over 20% of the space in 8 out of 18 types (even 70% in one type) in the compressed files, mainly because AliCloud needs micro-second level timing information to accurately identify the order of events, for the purposes like performance debugging and resolving conflicts. To compress timestamps, we use the classic differential method to compute and store the delta value of two consecutive timestamps.

- We observe that numerical data are often correlated. A typical example is in an I/O log: when the user performs sequential I/Os, the offset of the next I/O is equal to the sum of the offset and length of the previous I/O. Such correlation provides an obvious opportunity to further compress numerical data. Following this idea, we have developed a novel algorithm to identify simple numerical correlation in log samples and apply the found rules during compression.

- We observe most numerical values are small and using fixed-length coding (e.g. 32 bits for an integer) will generate many 0 bits at the beginning. We propose *elastic coding*, which represents a number with an elastic number of bytes, to trim leading zeroes. Compared to general-purpose compression algorithms, elastic coding is more efficient at trimming leading zeroes; compared to fixed-length coding, elastic coding can reduce the length when the value is small but may increase the length when the value is large, which is a beneficial trade-off given our observation.

By combining all the efforts mentioned above, we have built LogReducer. We have applied LogReducer to 18 types of AliCloud logs (1.76TB in total) and 16 types of public logs (77GB in total). Compared with LZMA, LogReducer can achieve 1.19× to 5.30× compression ratio on all cases and 0.56× to 3.16× compression speed on logs over 100MB (LogReducer is comparably slower on smaller logs because of its initialization overhead). Compared with Logzip, LogReducer can achieve 1.03× to 4.01× compression ratio and 2.05× to 182.31× compression speed. Such results have confirmed that, with proper implementation and optimization, parser-based log compression is promising to compress large-scale production logs.

The contribution of this paper is three-fold. First, we study why state-of-the-art parser-based compression methods do not perform well on our production logs. Second, based on the study, we build LogReducer by improving the implementation of existing methods, applying proper techniques based on the characteristics of the logs, and introducing new techniques. Finally, we demonstrate the efficacy of LogReducer on a variety of logs. LogReducer is open source [39].

2 Background

2.1 Structure of Cloud Logs

We collect a large set of logs generated in AliCloud. They are from different applications developed by different teams, which serve for various purposes, e.g., warning and error reporting, infrastructure monitoring, user behavior tracing, and periodical summary. Table 1 shows examples of three types of logs. Samples of all 18 types can be found in [38].

The basic structure of these logs contains three parts: header, template and variable. A header includes the timestamp and the corresponding log level. The header is added by AliCloud's logging system automatically and its format is relatively static, which allows us to use a regular expression to separate the header from the remaining part. The rest of this section mainly discusses how to parse the remaining part into templates and variables.

Templates are the formalized output statements of logs. In Log F, "Write chunk %s Offset %d Length %d" and "Read chunk %s Offset %d" are two templates. Variables refer to the part which varies in each instance of the same template. In Log F, they include "3242513_B", "339911", "11" ,etc.

User behavior tracing (Log F)
[2019-08-27 15:21:24.456234] [INFO] Write chunk: 3242513_B Offset: 339911 Length: 11
[2019-08-27 15:21:24.463321] [INFO] Read chunk: 3242514_C Offset: 272633
[2019-08-27 15:21:24.464322] [INFO] Write chunk: 3242512_F Offset: 318374 Length: 7
[2019-08-27 15:21:24.474433] [INFO] Write chunk: 3242513_B Offset: 339922 Length: 55
Infrastructure monitoring (Log D)
[2018-01-12 08:53:12.188370] [10593] project:393 logstore: XDoFiqnlmZd shard:78 inflow:3376 dataInflow:18869
[2018-01-12 08:53:12.188390] [10593] project:656 logstore: lOdMafL31Pg shard:37 inflow:7506 dataInflow:42712
Warning and error reporting (Log Q)
Aug 28 03:09:02 h10c10322.et15 su[57118]: (to nobody) root on none
Aug 28 03:09:02 h10c10322.et15 su[57118]: session opened for user nobody by (uid=0)

Table 1: Examples of logs in AliCloud.

2.2 Parser-based Log Compressor

Parser-based log compression first uses a *log parser* to identify the template of each log entry and extract the corresponding variables; it then replaces the template string with a template ID to save space; it finally applies general-purpose compression methods to variables to further reduce space.

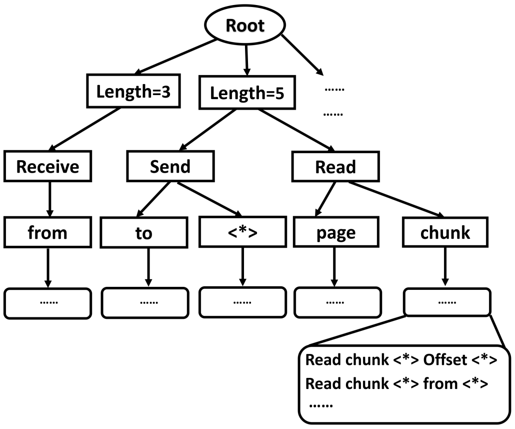


Figure 1: Parser tree architecture.

Log parser can be implemented using longest common string [8], clustering [35], and parser tree [23], among them parser tree shows better effectiveness [49]. Here we first present the concept of parser tree and then show how to build the parser tree and use it to separate templates and variables.

Parser tree. Given a list of templates and log entries, a naive approach to match the entry to a template is to compare the entry to each template and find the template which is most similar to the entry. However, when there are many templates, such one-by-one comparison is inefficient.

To improve the efficiency of template matching, several works [23, 30] use a parser tree to facilitate the matching: as shown in Figure 1, each leaf of the parser tree is a group of templates sharing the same length (i.e., the number of tokens in a log message); the first layer of internal nodes use the length of the log entry to categorize the entry; the following layers of internal nodes form multiple paths, each of them leads to a leaf node in the parser tree. Both the internal node and the template use "<*>" to represent a variable.

Assuming we have built a parser tree as shown in Figure 1, and we have a log entry "Read chunk 3242514_C Offset 272633": since its length is 5, we will first go to the internal node "Length=5"; since its first token is "Read", we will then go to the internal node "Read"; and then "chunk"; finally we will compare the log entry with each template in the leaf node and find "Read chunk <*> Offset <*>" is closest to the log entry, so we will choose this template and identify "3242514_C" and "272633" as variables.

Building the parser tree. Parser-based log compressor first builds the parser tree by parsing a sample of the log entries. For each log entry, the log parser performs four steps, and we use log entry "Read chunk 3242514_C Offset 272633" as an example to explain these steps. In the beginning, the parser tree just has one root node.

In the first step, the log parser uses predefined split characters, such as empty space or comma, to split a log entry into a list of strings called tokens. In our example, the raw log message will be divided into "Read", "chunk", "3242514_C", "Offset", "272633" accordingly.

In the second step, the log parser will check whether the internal node of the length exists (Length=5 in our case). If not, the log parser will create a new internal node. Finally, it moves to the corresponding internal node.

In the third step, the log parser traverses the tree according to the tokens in the log entry and moves to corresponding internal nodes ("Read" and "chunk" in our example). After reaching the limitation of tree depth, it reaches a leaf node, which contains a group of templates. If the corresponding internal node does not exist, the log parser will build the internal node and add the node to the prefix tree.

Definition 1. ***Similarity between log L and template T*** *(l_i is the ith token in L; t_i is the ith token in T; $\phi(a,b) = 1$ if $a = b$, otherwise $\phi(a,b) = 0$, $|\cdot|$ is the number of tokens in a log)*

$$Similarity(L,T) = \frac{\sum \phi(l_i,t_i)}{|L|} \quad (1)$$

In the fourth step, the log parser searches for the most similar template in this template group using a similarity function defined in equation 1. If the largest similarity is smaller than a threshold ε, the log parser will create a new template, which is the same as the log entry. Note that at this moment the log parser cannot tell which tokens of the log entry are variables. If the largest similarity is larger than ε, the log parser will regard this log entry as an instance of the matching template and update the template accordingly to mark different parts as variables.

For example, suppose the log parser first parses "Read chunk 3242514_C Offset 272633": since there is no template yet, the log parser will create a new template "Read chunk 3242514_C Offset 272633". Then suppose the log parser processes "Read chunk 3242514_B Offset 268832": its similarity to "Read chunk 3242514_C Offset 272633" is 0.6, so if ε is smaller than 0.6, the log parser will consider "Read chunk 3242514_B Offset 268832" as an instance of "Read chunk 3242514_C Offset 272633" and update the template into "Read chunk <*> Offset <*>"; if ε is larger than 0.6, the log parser will treat "Read chunk 3242514_B Offset 268832" as a new template.

Compressing logs. Then the compressor uses the parser tree to compress logs [30]. The procedure is similar to building a parser tree, except that in this phase, the compressor will not update the parser tree. It will first utilize the parser tree to try to match each log entry to a template. If a match is found, the log entry will be converted to the template ID and the variables; if no template is matched, the log entry will be regarded as a mismatch and will not be converted.

Afterward the compressor will group log entries according to their template IDs and store their variables in a column manner, i.e., it first stores the first variable of each log entry in the group, then stores the second variable, and so on. The column-based storage is based on the observation that variables at the same position of the same template are prone to have more redundancy, so that sliding window based algorithms such as LZ77 [40] will have more chances to trim redundancy. Finally the compressor concatenates everything and compresses it with a general-purpose compressor.

3 Restore the Promise of Parser-based Log Compression

We tested Logzip, the most recent parser-based log compression implementation, on 18 types of AliCloud's production logs. First, we find the value of the similarity threshold ε has a critical impact on the performance of Logzip. When using Logzip's default value 0.5, we find Logzip takes nearly 20 days to build the parser tree and can generate tens of thousands of templates, which impairs both the speed and the compression ratio. We tuned this value on our logs and found a value of 0.1 works well for almost all of our logs. This is due to the following reason: Logzip was mainly tested on PC logs, which usually are short and only contain a small number of variables; AliCloud's logs usually have more variables (see Table 2), and thus logs within the same templates are quite different from each other, i.e., they share only a small number of common tokens. Therefore, to extract the correct templates, we need a smaller ε. Manual tuning is always undesirable in a production environment: while we find the value 0.1 works well, for environments with more versatile logs, an automatic tuning procedure might be beneficial.

We continued testing Logzip with $\varepsilon = 0.1$ and found the result is still not ideal in terms of both compression ratio and compression speed: compared with LZMA, the general-purpose compression algorithm that can achieve the highest compression ratio on our logs, Logzip is seven times slower, and on 13 out of the 18 types of logs, Logzip's compression ratio is lower (§6.1). Our detailed analysis revealed a correlated problem between compression ratio and speed:

Logzip implementation assumes that a log entry usually has no more than five variables: for log entries with more than five variables, Logzip will regard content after the first variable as a large variable, and feed it to the general-purpose compressor. However, as shown in Table 2, this assumption does not hold on most of our logs (many have over 10 variables and one has 176 variables). As a result, Logzip loses its effectiveness on our logs, which can explain its poor compression ratio.

We tried to increase this limit and found it further exacerbates the speed problem of Logzip: while Logzip is already seven times slower than LZMA with the limit of five variables, increasing the limit to 256 will make Logzip unbearably slow, which might be the reason Logzip sets a small limit. We profiled Logzip to understand its bottleneck and found it has used several notoriously slow libraries or data structures including Pandas DataFrame, Python array append, etc. To address this problem, we re-implement the whole algorithm in C/C++ and dramatically improve the speed. The increased speed allows us to remove the limit of five variables as well. We further improve the speed with the following techniques:

Cutting the Parser Tree. We have observed that the total number of templates in our production logs is usually small: as shown in Table 2, 15 types of logs have less than 50 templates. If we group them based on length, the number of templates in one group is even smaller.

The reason behind this observation is that these cloud logs are generated by developers in the operation engineering group of AliCloud, and thus their patterns are relatively static compared to logs generated by cloud users.

Based on this observation, we cut the parser tree into only one layer in the compression phase: we only take length into consideration and we store templates with the same length together and search them one by one. This optimization has improved compression speed and avoided the tuning of the depth of the parser tree.

Batch processing. If we need to compress a large number of small log files, and we start one compressor process to compress each log, we observe the overhead to start and stop processes could slow down the whole compression significantly. Therefore, we allow our compressor to take a batch of log files as inputs and compress them together.

With all the efforts mentioned above, we have restored the promise of parser-based compression: as shown in §6.2, compared with LZMA, our implementation (i.e., LogReducer-B) can achieve 1.16× - 3.73× compression ratio and 0.51× - 2.01× compression speed.

4 Further Compressing Numerical Variables

We have done a detailed analysis on the compressed files generated by the previous step and found that in 10 of the 18 types of logs, numerical variables account for over 50% space after compression; for other types of logs the rate is at least 20%; in three cases, the rate is over 80% (Table 3). This is because 1) our logs have a large number of numerical variables and 2) general-purpose compression methods, which try to identify redundant bytes, do not work well with numerical variables.

4.1 Compressing Timestamps

Our analysis shows that timestamps are the first dominant numerical data in our logs. As shown in Table 3, in eight types of logs, timestamps account for more than 20% of space. In one case, this rate can reach close to 70%.

This is because AliCloud needs precise timing information to order events, for purposes like debugging and auditing. In its environment, system logs can be generated at a high

Log type	Log A	Log B	Log C	Log D	Log E	Log F	Log G	Log H	Log I
# of templates	42	29	3	6	74	7	202	39	48
Avg. # of variables	14	5	10	13	22	12	7	57	176
Log type	**Log J**	**Log K**	**Log L**	**Log M**	**Log N**	**Log O**	**Log P**	**Log Q**	**Log R**
# of templates	29	10	4	16	49	1	20	43	130
Avg. # of variables	3	46	7	9	4	13	22	2	5

Table 2: Template information on 18 types of logs in AliCloud.

Log type	Log A	Log B	Log C	Log D	Log E	Log F	Log G	Log H	Log I
Number Rate(%)	46.63	68.51	52.19	82.86	51.69	88.42	33.92	45.51	31.65
Time Rate(%)	36.75	38.97	15.28	15.49	10.42	10.07	22.88	31.23	4.22
Log type	**Log J**	**Log K**	**Log L**	**Log M**	**Log N**	**Log O**	**Log P**	**Log Q**	**Log R**
Number Rate(%)	39.27	69.85	24.89	53.54	53.40	78.47	27.30	29.36	84.96
Time Rate(%)	26.96	7.18	9.32	21.53	25.63	14.79	15.77	14.90	68.27

Table 3: Space consumption of numerical variables and timestamps in compressed file.

speed, up to one million entries per second, which motivates AliCloud to record timestamps at micro-second level. As a result, first, it takes more bits to store the microsecond level timestamps than millisecond or second level timestamps, and second, there is not much redundancy in the timestamps.

To compress timestamps, we use the classical differential method, which records the delta value between two consecutive timestamps. This method can significantly reduce the size of timestamps when the target system generates logs frequently, namely the delta value will be small. By using this method, we can reduce the space overhead and pass a much smaller number to the general-purpose compressor, which can improve both compression ratio and compression speed.

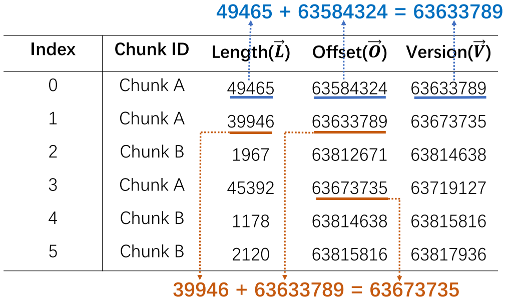
4.2 Correlation Identification and Utilization

We observe numerical variables sometimes are correlated. For example, in an I/O trace, if the user performs sequential I/Os, the offset of the next I/O will be equal to the sum of the offset and length of the previous I/O.

Such correlation provides an obvious opportunity to compress numerical data. If most values of certain variables follow certain kind of correlation, we only need to store how values deviate from the correlation in a *residue vector*; since most values of the residue vector will be zeroes, they will be effectively compressed by a general-purpose compressor.

For example, for logs of type Log F with templates “Write Chunk <*> Length <*> Offset <*> Version <*>”, we can extract four variables from the template and three of them are numerical variables, namely $\vec{L}$ (Length), $\vec{O}$ (Offset), $\vec{V}$ (Version). In our logs, we find three types of correlations in these variables. Note that the values of each variable form a vector since there are multiple log entries.

- Inter-variable correlation: Version is often equal to the sum of Offset and Length, namely $\vec{V} = \vec{L} + \vec{O}$, and its residue vector is $\vec{V} - \vec{L} - \vec{O}$.


49465 + 63584324 = 63633789
39946 + 63633789 = 63673735

Index	Chunk ID	Length($\vec{L}$)	Offset($\vec{O}$)	Version($\vec{V}$)
0	Chunk A	49465	63584324	63633789
1	Chunk A	39946	63633789	63673735
2	Chunk B	1967	63812671	63814638
3	Chunk A	45392	63673735	63719127
4	Chunk B	1178	63814638	63815816
5	Chunk B	2120	63815816	63817936

Figure 2: Numerical correlations observed on Log F.

- Intra-variable correlation: Lengths of the same Chunk ID are often close. We can compute its residue vector as $\vec{L}[i] - \vec{L}[i-1] = \vec{\Delta L}$.
- Mixed correlation: if the user is performing sequential I/Os to a chunk, then its lengths and offsets have the following correlation: $\vec{O}[i] = \vec{O}[i-1] + \vec{L}[i-1]$. Its residue vector is $\vec{O}[i] - \vec{O}[i-1] - \vec{L}[i-1] = \vec{\Delta O} - \vec{L}$.

Correlation identification. We propose a novel method to identify such correlation. The goal of correlation identification process is to find the relationship across and within different variables so that we can represent some variables with residue vectors, which can be compressed more effectively. To achieve this goal, we first enumerate different combinations of variables, the IDs to group different entries (e.g. ChunkID in Figure 2), and the aforementioned correlation rules, and compute the corresponding residue vectors. Then we select vectors from the original vectors and the residue vectors with the goals of 1) maximizing compression ratio and 2) being able to recover all original vectors.

The whole identification process is illustrated in algorithm 1, which maintains three sets: the target set ψ; the recover set $\mathbb{R}$ including all original vectors that can be recovered from the current ψ; the total candidate set $\mathbb{T}$ including all candidate vectors. One of its key data structure is a map from the candidate vectors to original vectors: $map(\vec{C})$ will return all original vectors that $\vec{C}$ is built from (e.g., $map(\vec{A}-\vec{B}) = \vec{A}$ and $\vec{B}$).

Algorithm 1 Correlation identification algorithm

```
 1: Recoverable set $\mathbb{R} = \emptyset$
 2: Final vector set $\Psi = \emptyset$
 3: Initialize candidate set $\mathbb{T}$
 4: repeat
 5:     $\mathbb{C} = \{\vec{C} \in \mathbb{T} : |map(\vec{C}) - \mathbb{R}| = 1\}$
 6:     $\vec{C}_{min}$ = vector with the smallest entropy in $\mathbb{C}$.
 7:     $\Psi \leftarrow \Psi \cup \vec{C}_{min}$
 8:     $\mathbb{R} \leftarrow \mathbb{R} \cup map(\vec{C}_{min})$
 9: until $\mathbb{R}$ contains all original vectors
10: Output $\Psi$
```

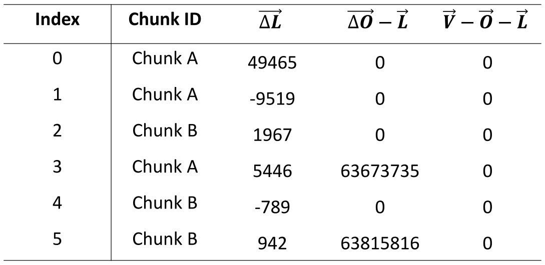
The algorithm works in iterations: in each iteration, it first tries to find all candidate vectors $\vec{C}$ that can recover one more variable compared to the current recoverable set $\mathbb{R}$ (line 5); then among them, it chooses the one with the highest compression ratio (line 6). Here we predict the compression ratio of a candidate using its Shannon Entropy [26], defined in Definition 2; finally it updates Ψ and $\mathbb{R}$ accordingly (lines 7 and 8); it repeats this until Ψ can recover all original vectors (line 9). The cost of enumeration is acceptable, since it is performed on samples of logs.

Definition 2. ***Entropy for a variable vector.*** *S_A denotes the set of all values appearing in $\vec{A}$ and #(s) denotes the number of times the value s appears in $\vec{A}$.*

$$E(\vec{A}) = -\sum_{s \in S_A} \frac{\#(s)}{|A|} \log \frac{\#(s)}{|A|} \tag{2}$$

In Figure 2, Ψ will finally contain three residue vectors, namely:$\{\Delta\vec{L}, \Delta\vec{O}-\vec{L}, \vec{V}-\vec{L}-\vec{O}\}$. These three residue vectors are enough to recover all original variable vectors $\vec{L}, \vec{O}, \vec{V}$, and will have a higher compression ratio than the original variable vectors.

Correlation utilization. The output of the training phase for numerical correlations is the target vector set Ψ. In the compression phase, we calculate each residue vector in set Ψ and discard original vectors that do not appear in Ψ. If we apply three correlations in Ψ to our example in Figure 2, the result is shown in Figure 3. As one can see, for variables that perfectly match certain rules, their residue vectors contain many zeroes; even for those that do not perfectly match the rules, their values are smaller, which facilitates the elastic encoder discussed in the following section.



Index	Chunk ID	$\Delta\vec{L}$	$\Delta\vec{O}-\vec{L}$	$\vec{V}-\vec{O}-\vec{L}$
0	Chunk A	49465	0	0
1	Chunk A	-9519	0	0
2	Chunk B	1967	0	0
3	Chunk A	5446	63673735	0
4	Chunk B	-789	0	0
5	Chunk B	942	63815816	0

Figure 3: Processing result of logs in Figure 2.

4.3 Elastic Encoder

The simplest way to represent numerical variables is to use fixed number of bytes (e.g. 4 bytes to represent an integer, 8 bytes to represent a long value, etc). However, if most numbers are small, these bytes will contain many leading zeroes (for positive numbers) or ones (for negative numbers). General-purpose compression may be able to find such consecutive zeroes or ones, but since it needs to search for such zeroes/ones and store additional metadata to record the length of zeroes/ones, we design a dedicated encoding algorithm to trim leading zeroes.

To efficiently exploit such opportunity, we apply an elastic encoding method to store numbers according to their size. We cut the 32-bit integer into 7-bit segments and add one bit to each segment, indicating whether the segment is the last segment (1 means it is the last). Then we discard the prefix of segments containing only zeroes. Here we choose the number 7 because, after adding one bit, each segment takes a byte, which is easy to handle.

For a negative number represented by two-complement encoding, it is not trivial to just change all ones to zeroes, since the leading ones include the first bit which indicates this number is a negative number. To overcome this problem, we adopt a shifting operator [43] to move the first bit to the last position and reverse all other bits if the original number is negative. By adopting this method, we will process negative numbers in the same way as processing positive ones.

By using elastic encoding, we will trim the leading zeros/ones at the cost of adding one-bit metadata for every remaining 7 bits. Therefore, the smaller the number is, the more redundancy we can trim. More precisely, for an integer between $[-2^{7n}, -2^{7(n-1)}) \cup (2^{7(n-1)} - 1, 2^{7n} - 1](0 < n < 6)$, elastic encoding can save $(32 - 8n)$ bits compared with using fixed 32 bits. In our logs, we find this method can save 24 bits (i.e., $n = 1$) for more than 60% of the numbers.

Note that both delta timestamps and applying correlation contribute to the effectiveness of elastic encoding since these techniques tend to make numbers smaller.

5 Architecture and Implementation

Based on the observations and ideas mentioned previously, we have built LogReducer, a parser-based log compressor,

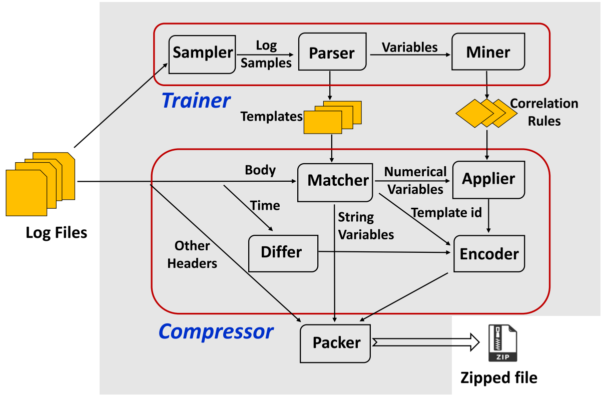


Figure 4: LogReducer architecture.

with about 3,000 lines of C/C++ code. Figure 4 shows its architecture. LogReducer contains two phases, training phase and compression phase.

Training. Training phase is done over sampled data. It uses a parser to extract templates (§2) and a correlation miner to find possible correlations (§4.2). At the end of this phase, LogReducer may find a list of templates and correlation rules.

Just like any other methods relying on sampling, we expect such samples to capture the properties of real logs as much as possible. Traditional parser-based compression methods like Logzip only need to extract templates during the training phase, and since the template of a log entry only depends on the entry itself, these methods can use random sampling. LogReducer, however, tries to identify data correlation across adjacent log entries, and random sampling will lose such relationship. To address this challenge, we first randomly pick several starting points and then choose a contiguous sequence of logs from each starting point. This method shows good performance on both extracting templates and identifying correlations across adjacent log entries.

Compression. In the compression phase, for each log entry, LogReducer will first extract its header. LogReducer will further extract the timestamps from the headers and compute the delta values of consecutive timestamps (§4.1). Then LogReducer will try to match the log entry to templates using the parser (§3) and apply founded correlations to numerical variables (§4.2). Then LogReducer will encode all numerical data, including timestamps, numerical variables, and template IDs, using elastic encoder (§4.3). Finally, LogReducer will pack all data using LZMA since we find it can almost always achieve the highest compression ratio on our logs.

In order to illustrate the whole process of compression, we exhibit a complete compression case. Suppose we have four input log entries of Log F shown in Table 1 (§2) and the templates and correlations founded in the training phase.

LogReducer first extracts their headers and matches their bodies to templates. The results are shown in Table 4, in which each log entry is divided into three parts, namely log header, template ID, and corresponding variables. Here the second log entry belongs to template:"Read chunk <*> Offset:<*>", whose template ID is 2 and the other three log entries belong to template: "Write chunk <*> Offset:<*> Length:<*>", whose template ID is 1. As a result, template 2 has two variables and template 1 has three variables.

Headers	Template ID	V1	V2	V3
[2019-08-27 15:21:24.456234] [INFO]	1	3242513_B	339911	11
[2019-08-27 15:21:24.463321] [INFO]	2	3242514_C	272633	-
[2019-08-27 15:21:24.464322] [INFO]	1	3242512_F	318374	7
[2019-08-27 15:21:24.474433] [INFO]	1	3242513_B	339922	55

Table 4: Extracting headers and matching templates.

Then LogReducer will compute the difference of adjacent timestamps and utilize correlations over numerical variables. Table 5 shows the result of these steps: all timestamps become much smaller except the first one; since LogReducer identifies the sequential access pattern for 3242513_B, it does not need to store the offset of the second access and we calculate the delta result for write length of the same chunk (i.e., log entry 4). Finally LogReducer encodes all numerical results using elastic encoder, organizes all variables in a column manner, and packs them with LZMA.

Time	Other Header	Template ID	V1	V2	V3
2019 08 27 15 21 24 456234	[INFO]	1	3242513_B	339911	11
0 0 0 0 0 0 7087	[INFO]	2	3242514_C	272633	-
0 0 0 0 0 0 1001	[INFO]	1	3242512_F	318374	7
0 0 0 0 0 0 10111	[INFO]	1	3242513_B	0	44

Table 5: Computing the delta values of timestamps and applying correlation.

Others. In cloud environments, logs usually contain a large amount of information. To make such information easy to be understood by humans, logs often need to be truncated into several lines. Such multi-line log entries do not exist in the PC logs where Logzip was tested upon. Logzip simply treats them as a mismatch, which obviously reduces its effectiveness.

We calculate the rate of multi-line logs in our production logs and find in Log H and Log R, the rate of multi-line logs have reached up to 5% of the whole size. To support multi-line log entries, we do not split log entries based on a new line symbol; instead, we split log entries based on log headers, as we discussed in §2.1. Doing so achieves both higher compression ratio and higher compression speed.

Besides, to improve the generality of LogReducer beyond AliCloud logs, we implement a head-format adaptor: based on the assumption that the number of tokens in the head is static for the same type of logs, this adaptor tries to treat the first n tokens as the head (it tries $n = 1$ to 10 in our experiments) to see which n value can achieve the best compression ratio.

6 Evaluation

Our evaluation tries to answer three questions:

- What is the overall performance of LogReducer in terms of compression ratio and compression speed on AliCloud logs? (§6.1)
- What is the effect of each individual technique of LogReducer? (§6.2)
- How does LogReducer perform on logs beyond AliCloud logs? (§6.3)

To answer these questions, we measure the performance of LogReducer on 18 types of production logs from AliCloud with a total size of about 1.76TB (Table 6). We measure both the compression ratio (i.e., $\frac{Original\ size}{Compressed\ size}$) and the compression speed (MB/s).

For comparison, we also measure the performance of two general-purpose compression algorithms (gzip and LZMA) and two log-specific compression algorithms (LogArchive [3] and Logzip [30]). gzip is a classical compression tool. It targets high compression speed instead of a high compression ratio. We use "tar" [7] command to compress log dataset with gzip. LZMA is a well-studied general-purpose compression method based on LZ77 [40] algorithm. It has a high compression ratio but a relatively low compression speed. We use 7z [50] to compress the log data with LZMA. LogArchive is a bucket-based log compression method. We use its open-source code to compress our logs data [17]. Logzip is the latest implementation of parser-based compressor. We use its open-source code [19]. Note that as discussed in §3, we change the ε value of Logzip from 0.5 to 0.1.

Testbed. We perform all experiments using 4 Linux servers, each with 2× Intel Xeon E5-2682 2.50GHz CPUs (with 16 cores), 188GB RAM, and Red Hat 4.8.5 with Linux kernel 3.10.0. For each method, we use 4 threads to compress the log data in parallel and sum their total time.

6.1 Overall Performance

Compression ratio. As shown in Figure 5(a), LogReducer has the highest compression ratio on all logs. It can achieve 1.54× to 6.78× compression ratio compared to gzip, 1.19× to 4.80× compared to LZMA, 1.11× to 3.60× compared to LogArchive, and 1.45× to 4.01× compared to Logzip.

In our experiments, Logzip failed on Log I; LogArchive failed on Log I and Log J. Both of these two logs have much longer log entries than others, which causes buffer overflow in Logzip and LogArchive. We use LZMA to compress failed logs, since it is the default setting of Logzip and LogArchive.

LogReducer can compress all 1.76TB log dataset into 34.25GB, which takes only 1.90% space after compression. gzip, LZMA, LogArchive, and Logzip can compress all 1.76TB into 152.03GB, 107.22GB, 91.54GB, and 89.86GB respectively. As a result, their space consumption is 4.44×, 3.13×, 2.67× and 2.62× as much as LogReducer respectively.

We further compute the improvement of LogReducer over the best of the other four algorithms on all 18 logs. LogReducer has the highest improvement on Log F and lowest improvement on Log L due to the following reasons: Log F has several typical correlations we discussed in §4.2 and LogReducer can identify them and trim redundancy effectively, while other works cannot utilize such correlation. Log L has a low percentage of numerical values (only 24.89%) and timestamps (only 9.32%), which means the new techniques introduced by LogReducer are not very effective.

Compression speed. As shown in Figure 5(b), LogReducer is 4.01×-182.31× as fast as Logzip and 4.49×-11.65× as fast as LogArchive. LogReducer is comparable to LZMA in compression speed (0.56×-3.16×): it is slower than LZMA on 8 out of 18 logs; in some special cases (Log K, Log F, Log O) LogReducer is 2×-3× as fast as LZMA. LogReducer is slower than gzip, as gzip is optimized for speed. We do not show the speed of gzip in Figure 5(b) since its high value will make other bars hard to distinguish.

To compress all 1.76TB logs, LogReducer takes 58.19 hours; Logzip takes nearly 27 days; LogArchive takes nearly 23 days; LZMA takes 91.54 hours; gzip takes 25.35 hours. In other words, LogReducer is 11.22×, 9.43×, and 1.57× as fast as Logzip, LogArchive, and LZMA respectively; it is about 60% slower than gzip.

6.2 Effects of Individual Techniques

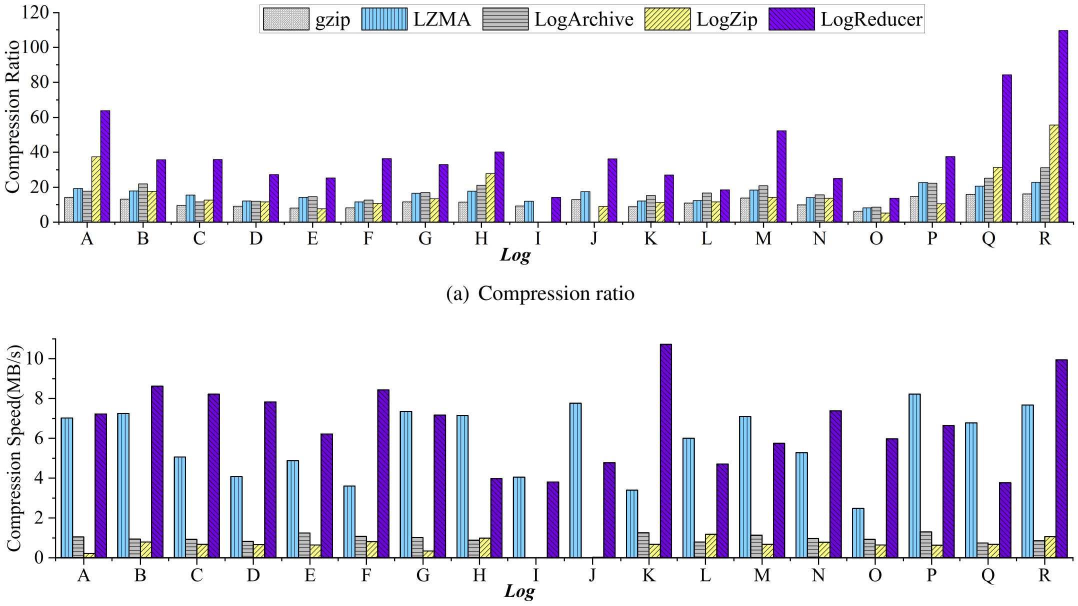
This section measures the effects of individual techniques presented in §3 and §4.

We use an efficient re-implementation of parser-based compressor as our baseline (LogReducer-NB), which includes the C/C++ implementation, removing the limit on the number of variables, and cutting the parser tree (§3). We add batch processing on LogReducer-NB to get LogReducer-B (§3), add delta timestamps on LogReducer-B to get LogReducer-D (§4.1), add elastic encoding approach on LogReducer-D to get LogReducer-ED (§4.3) and finally add numerical correlation utilization (§4.2) on LogReducer-ED to get the full version of LogReducer. The result is shown in Table 7.

As one can see, the efficient re-implementation (NB version) significantly improves the compression ratio and compression speed over Logzip in almost every type of logs. This has confirmed one of our key observations: an efficient implementation is critical to realize the full potential of parser-based log compression.

Log type	Log A	Log B	Log C	Log D	Log E	Log F	Log G	Log H	Log I
Total Size(GB)	18.67	16.05	45.82	65.74	34.98	443.30	148.88	0.19	14.20
Total Line(10^6)	74.74	72.60	231.43	406.98	77.56	1425.37	579.94	1.08	8.65
Time Span(H)	476	75	87	20	6	8	1563	32	8977
Log type	**Log J**	**Log K**	**Log L**	**Log M**	**Log N**	**Log O**	**Log P**	**Log Q**	**Log R**
Total Size(GB)	18.67	16.05	45.82	65.74	34.98	443.30	148.88	0.19	14.20
Total Line(10^6)	74.74	72.60	231.43	406.98	77.56	1425.37	579.94	1.08	8.65
Time Span(H)	85	3335	238	30	174	1512	62	165	722

Table 6: Log dataset description.



(a) Compression ratio

(b) Compression speed

Figure 5: Performance on AliCloud logs

The B version (batch processing) is over 1.5× as fast as the NB version on 10 logs. In particular, it is 1.99× and 1.82× as fast as the NB version on Log D and Log C. These two logs have many files and thus LogReducer can save much time by batch processing. Batch processing has no impact on compression ratio as it does not change the logic of the compression algorithm.

The compression ratio of the D version (delta timestamps) is over 1.1× as high as the B version on 3 logs and over 1.05× as high on 7 logs. In particular, its compression ratio is 1.24× as high as the B version on Log R. Delta timestamps can bring significant improvement to logs which have a large percentage of timestamp values: Log R, Log A and Log H have the highest, third highest and fourth highest timestamp percentage among all 18 logs (see Table 3) and thus can benefit from delta timestamps. Log B, which has the second highest timestamp percentage, is relatively sparser and does not benefit much from delta timestamps. Delta timestamps improves compression speed as well by feeding a smaller intermediate result to the general-purpose compressor: it is over 1.05× as fast as the B version on 6 logs.

The compression ratio of the ED version (elastic encoding) is over 1.05× as high as the D version on 12 logs. It mainly improves compression ratio on logs with a large percentage of small numbers, such as Log D, Log R, and Log M. The ED version is over 1.5× as fast as the D version on 5 logs and 1.2× as fast on 11 logs, since elastic encoding provides a dedicated and thus more efficient way to trim leading zeroes or ones compared with general-purpose methods.

The compression ratio of the LR version (correlation identification and utilization) is over 1.05× as high as the ED version on 4 logs. In particular, it is 2.07× as high as the ED version on Log F and 1.13× as high on Log O, because correlations are common in these two logs. It will incur overhead, its speed is 0.7× to 1.05× compared with ED version.

6.3 Performance on Public Logs

To examine the generality of LogReducer beyond AliCloud logs, we evaluate LogReducer on 16 types of public logs [18]

	Compression Ratio						**Compression Speed (MB/s)**						
	LZMA	**Logzip**	**B & NB**	**D**	**ED**	**LR**	**LZMA**	**Logzip**	**NB**	**B**	**D**	**ED**	**LR**
Log A	19.30	37.34	53.96	61.94	63.79	63.86	7.03	0.22	3.99	6.42	6.31	7.58	7.23
Log B	17.91	17.64	32.66	33.55	35.63	35.67	7.25	0.79	3.80	6.75	7.21	9.52	8.63
Log C	15.48	12.61	30.36	32.30	34.80	35.81	5.06	0.68	3.47	6.31	6.17	9.50	8.22
Log D	12.16	11.57	23.08	24.50	26.56	27.26	4.08	0.66	2.84	5.64	5.29	9.80	7.83
Log E	14.19	7.73	22.99	23.35	24.73	25.22	4.89	0.64	4.33	5.34	5.42	6.93	6.22
Log F	11.58	10.69	16.32	16.47	17.62	36.42	3.60	0.81	3.32	4.33	4.33	8.00	8.44
Log G	16.58	13.42	30.23	31.76	33.00	32.99	7.35	0.34	4.07	5.89	6.52	7.27	7.17
Log H	17.73	27.73	34.85	38.58	40.05	40.08	7.15	0.99	3.71	3.64	3.83	3.96	3.98
Log I	11.95	/	13.88	13.88	14.03	14.26	4.05	/	5.26	3.81	3.57	3.85	3.81
Log J	17.46	9.04	31.16	33.25	34.94	36.22	7.76	0.03	2.72	4.37	4.60	4.82	4.78
Log K	12.14	11.20	23.88	24.51	25.74	26.97	3.39	0.67	4.53	6.82	6.24	10.76	10.72
Log L	12.38	11.62	17.75	17.96	18.43	18.48	6.01	1.17	2.55	4.74	4.80	5.47	4.71
Log M	18.42	14.20	37.56	39.14	43.56	43.99	7.10	0.67	4.90	5.22	6.55	7.21	5.75
Log N	14.11	13.64	22.43	22.63	23.71	25.01	5.28	0.77	3.56	5.68	5.66	7.61	7.38
Log O	8.25	5.23	11.35	11.28	12.05	13.67	2.48	0.64	2.52	3.42	3.44	7.15	5.98
Log P	22.73	10.61	34.90	35.98	36.92	37.58	8.22	0.63	5.75	5.52	7.14	9.32	6.64
Log Q	20.55	31.27	76.72	79.05	83.09	84.25	6.78	0.68	2.41	3.67	3.72	3.76	3.77
Log R	22.82	55.63	80.73	100.44	109.21	109.51	7.67	1.07	4.94	8.23	7.85	10.87	9.95

Table 7: Effects of individual techniques on compression ratio and compression speed. X is short for LogReducer-X (X∈{B, NB, D, ED}). LR stands for the full version of LogReducer. "/": Logzip failed on Log I.

from diverse sources [25, 49]. As shown in Figure 6, the compression ratio of LogReducer is 1.03×–3.15× compared with Logzip, 1.19×–5.14× compared with LogArchive, 1.23×–5.30× compared with LZMA, and 1.79×–20.27× compared with gzip. We further investigate the logs on which LogReducer has less improvement: some of them have too many templates (e.g. Android, Thunderbird), which causes all parse-based methods, including LogReducer, to have many mismatches; some of them have only a few variables and even fewer numerical variables (e.g., Thunderbird, Proxifer), which causes LogReducer's optimizations to be less effective; in addition, LogZip has a specific optimization for HDFS log, which improves the compression ratio of LogZip.

In terms of compression speed, LogReducer is 2.05×–101.12× as fast as Logzip and 1.79×–9.95× as fast as LogArchive. LogReducer is slower than LZMA by up to 5.88× and than gzip by up to 36.16× due to two reasons. First, since over half of the logs are smaller than 100MB, the initialization overhead of LogReducer (e.g. space allocation) becomes significant, taking over 40% of the time. Second, some cases have too many templates (e.g. Android, Thunderbird), which causes a low matching rate and a waste of time.

Such results have confirmed the assumptions of LogReducer: LogReducer is mainly designed for large-scale logs with a small number of templates and many variables. When such assumptions hold, LogReducer can perform significantly better than existing methods; when such assumptions do not hold, LogReducer is less effective but can still achieve the highest compression ratio.

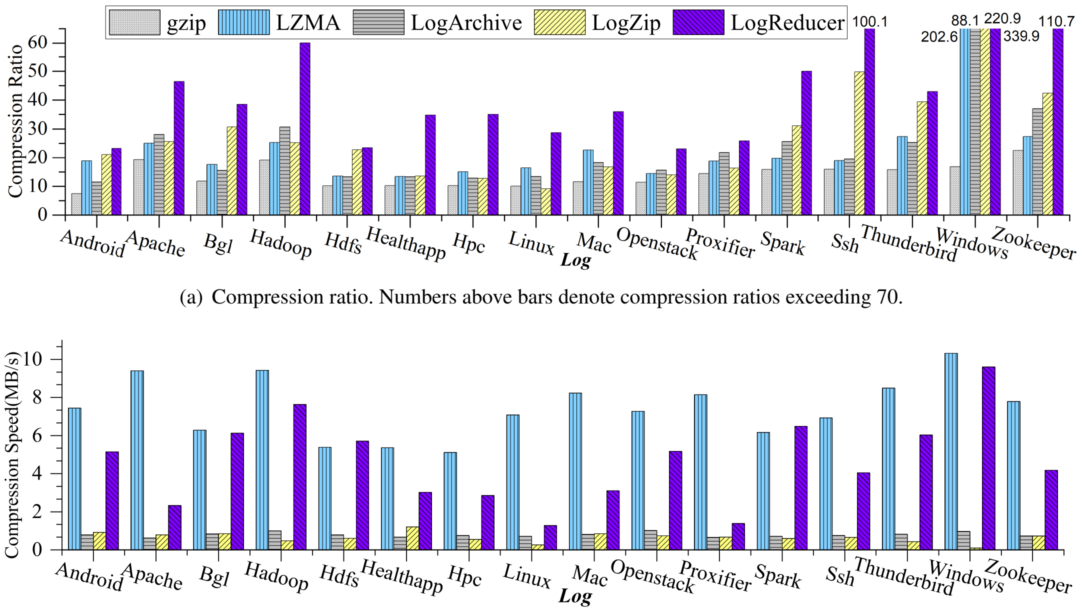
7 Related Work

Log parser. Log parser focuses on the extraction process of log templates, which can be divided into three types: cluster-based methods (LKE [14], LogSig [44], SHISO [34], LenMa [42], LogMine [20]), frequent-pattern-based methods (SLCT [46], LFA [35]), and heuristic-structure-based methods (IPLoM [32], AEL [27], Drain [23]).

Cluster-based methods divide the logs into clusters and extract templates for each cluster. Pattern-based methods try to extract frequent patterns from log entries and regard them as constant templates. Heuristic methods will extract log structure based on observations of log entries. Zhu et al. [49] compare these methods and find that Drain performs better than others. As a result, both Logzip and our implementation are based on Drain.

Number encoding methods. LevelDB [16] has used variant encoding to represent numbers based on their size. Thrift [43] has used Zigzag encoding to get more leading zero to enable efficient data serialization when communicating between processes. Compared with them, LogReducer further uses elastic encoding to reduce the space overhead of storing numerical variables.

General-purpose compression approaches. These methods can be categorized into three kinds: statistic-based, predict-based, and dictionary-based. Statistic-based compression methods (e.g., Huffman coding [40], Arithmetic coding [47]) first collect statistic information about input logs and then design variant length coding for each tokens. Predict-



(a) Compression ratio. Numbers above bars denote compression ratios exceeding 70.

(b) Compression speed

Figure 6: Performance on public dataset.

based compression methods (e.g., PPMd [4]) predict the next token based on current context during reading the input stream, and assigns a shorter encoding if prediction is successful. Dictionary-based compression methods (e.g., LZMA, gzip) search for similar tokens in a sliding window and store them in a dictionary when processing the input stream.

Statistic-based methods need to read the input log file twice. As a result, when the input log file is large, they are not efficient. With prediction-based methods, the appearance of variables will decrease the prediction accuracy. Dictionary-based methods may lose the chance to trim redundancy within a long distance, and do not take the delta of timestamps and correlation of variables into consideration, since they are not related to redundancy literally. Our methods utilize general-purpose compression approaches and improve their effectiveness on log data.

Log-specific compression approaches. These methods can be divided into two categories: parser-based and non parser-based. CLC [22], LogArchive [3], Cowic [29] and MLC [13] process variables and templates together. CLC tries to find the frequent patterns shown in log files and processes these patterns directly. LogArchive uses similarity function and sliding windows to divide log entries into different buckets and compresses buckets together to improve compression ratio. Cowic does not focus on the compression ratio. Instead, it tries to decrease the decompression latency by only decompressing needed logs rather than the whole files. MLC uses block-level duplication methods to find redundancy concealing between log entries and divide them into groups according to their similarities and compress them using delta encoding.

Logzip [30] extracts templates and processes templates and variables separately. It uses a parser to get several templates on a small sample and extracts all templates in original log files by iterative matching. Finally, it compresses template IDs and variables using general-purpose compression methods separately. However, Logzip does not perform well on our logs due to sub-optimal implementation.

8 Conclusion

This work examines the latest parser-based log compression approach on production logs. It observes that, first, an efficient implementation is critical to realize the full potential of this approach; and second, there are more opportunities to further compress logs. Based on these ideas, we have built LogReducer, which shows promising compression ratio and compression speed.

Acknowledgment

We thank all reviewers for their insightful comments, and especially our shepherd, Dalit Naor, for her guidance during our camera-ready preparation. This work was supported by the National key R&D Program of China under Grant 2018YFB0203902, and the National Natural Science Foundation of China under Grants 61672315 and 62025203.

References

[1] Boyuan Chen and Zhen Ming Jiang. Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 39th International Conference on Software Engineering*, pages 71–81. IEEE, 2017.

[2] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The mystery machine: End-to-end performance analysis of large-scale Internet services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pages 217–231. USENIX Association, 2014.

[3] Robert Christensen and Feifei Li. Adaptive log compression for massive log data. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1283–1284. ACM, 2013.

[4] John Cleary and Ian Witten. Data compression using adaptive coding and partial string matching. *IEEE transactions on Communications*, 32(4):396–402, 1984.

[5] Python date engineer group. Python data analysis library Pandas. https://pandas.pydata.org/, 2015.

[6] Peter Deutsch. DEFLATE compressed data format specification version 1.3. https://tools.ietf.org/html/rfc1951, 1996.

[7] GNU developer group. Homepage and documentation of Tar. https://www.gnu.org/software/tar/, 2019.

[8] Min Du and Feifei Li. Spell: Streaming parsing of system event logs. In *Proceedings of the 16th International Conference on Data Mining*, pages 859–864. IEEE, 2016.

[9] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298. ACM, 2017.

[10] Susan Dumais, Robin Jeffries, Daniel M Russell, Diane Tang, and Jaime Teevan. Understanding user behavior through log data and analysis. In *Ways of Knowing in HCI*, pages 349–372. Springer, 2014.

[11] Yaochung Fan, Yuchi Chen, Kuanchieh Tung, Kuochen Wu, and Arbee L P Chen. A framework for enabling user preference profiling through Wi-Fi logs. *IEEE Transactions on Knowledge and Data Engineering*, 28(3):592–603, 2016.

[12] Bettina Fazzinga, Sergio Flesca, Filippo Furfaro, and Luigi Pontieri. Online and offline classification of traces of event logs on the basis of security risks. *Journal of Intelligent Information Systems*, 50(1):195–230, 2018.

[13] Bo Feng, Chentao Wu, and Jie Li. MLC: an efficient multi-level log compression method for cloud backup systems. In *Proceedings of 2016 IEEE Trustcom/BigDataSE/ISPA*, pages 1358–1365. IEEE, 2016.

[14] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 9th IEEE international conference on data mining*, pages 149–158. IEEE, 2009.

[15] Mona Ghassemian, Philipp Hofmann, Christian Prehofer, Vasilis Friderikos, and Hamid Aghvami. Performance analysis of Internet gateway discovery protocols in ad hoc networks. In *Proceedings of 2004 IEEE Wireless Communications and Networking Conference*, volume 1, pages 120–125. IEEE, 2004.

[16] Sanjay Ghemawat and Jeff Dean. LevelDB. https://github.com/google/leveldb, 2011.

[17] LogArchive group. Open source code of LogArchive. https://github.com/robertchristensen/log_archive_v0, 2019.

[18] Loghub group. Download link of public log dataset. https://zenodo.org/record/1596245#.XMMZ1dv7S-Y, 2019.

[19] Logzip group. Open source code of Logzip. https://github.com/logpai/logzip, 2019.

[20] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. LogMine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1573–1582. ACM, 2016.

[21] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. Studying and detecting log-related issues. *Empirical Software Engineering*, 23(6):3248–3280, 2018.

[22] Kimmo Hätönen, Jean François Boulicaut, Mika Klemettinen, Markus Miettinen, and Cyrille Masson. Comprehensive log compression with frequent patterns. In *Proceedings of 2003 International Conference on Data Warehousing and Knowledge Discovery*, pages 360–370. Springer, 2003.

[23] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *Proceedings of 2017 IEEE International Conference on Web Services*, pages 33–40. IEEE, 2017.

[24] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience report: System log analysis for anomaly detection. In *Proceedings of the 27th International Symposium on Software Reliability Engineering*, pages 207–218. IEEE, 2016.

[25] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. Loghub: A large collection of system log datasets towards automated log analytics. *arXiv preprint arXiv:2008.06448*, 2020.

[26] Edwin T Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.

[27] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *Proceedings of the 8th International Conference on Quality Software*, pages 181–186. IEEE, 2008.

[28] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. The unified logging infrastructure for data analytics at Twitter. *Proceedings of the VLDB Endowment*, 5(12):1771–1780, 2012.

[29] Hao Lin, Jingyu Zhou, Bin Yao, Minyi Guo, and Jie Li. Cowic: A column-wise independent compression for log stream analysis. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 21–30. IEEE, 2015.

[30] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R Lyu. Logzip: extracting hidden structures via iterative clustering for log compression. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pages 863–873. IEEE, 2019.

[31] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering*, 24(11):1921–1936, 2011.

[32] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1255–1264, 2009.

[33] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. A search-based approach for accurate identification of log message formats. In *Proceedings of the 26th Conference on Program Comprehension*, pages 167–177. ACM, 2018.

[34] Masayoshi Mizutani. Incremental mining of system log format. In *Proceedings of 2013 IEEE International Conference on Services Computing*, pages 595–602. IEEE, 2013.

[35] Meiyappan Nagappan and Mladen A Vouk. Abstracting log lines to log event types for mining software system logs. In *Proceedings of the 7th Working Conference on Mining Software Repositories*, pages 114–117. IEEE, 2010.

[36] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 26–26. USENIX Association, 2012.

[37] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. Detection of early-stage enterprise infection by mining large-scale log data. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 45–56. IEEE, 2015.

[38] LogReducer research group. Open sample of large-scale cloud logs. https://github.com/THUBear-wjy/openSample, 2020.

[39] LogReducer research group. Open source code of LogReducer. https://github.com/THUBear-wjy/LogReducer, 2020.

[40] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

[41] Julian Seward. The bzip2 home page. http://www.bzip.org, 1997.

[42] Keiichi Shima. Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213*, 2016.

[43] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.

[44] Liang Tang, Tao Li, and Chang-Shing Perng. LogSig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 785–794. ACM, 2011.

[45] Sarah K Tyler and Jaime Teevan. Large scale query log analysis of re-finding. In *Proceedings of the 3rd ACM international conference on Web search and data mining*, pages 191–200, 2010.

[46] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management*, pages 119–126. IEEE, 2003.

[47] Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[48] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural support for programming languages and operating systems*, pages 143–154. ACM, 2010.

[49] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. Tools and benchmarks for automated log parsing. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, pages 121–130. IEEE, 2019.

[50] 7 zip developer group. 7-zip file achiever home page. https://www.7-zip.org/, 2019.

CNSBench: A Cloud Native Storage Benchmark

Alex Merenstein[1], Vasily Tarasov[2], Ali Anwar[2], Deepavali Bhagwat[2], Julie Lee[1],
Lukas Rupprecht[2], Dimitris Skourtis[2], Yang Yang[1], and Erez Zadok[1]
[1]*Stony Brook University* *and* [2]*IBM Research–Almaden*

Abstract

Modern hybrid cloud infrastructures require software to be easily portable between heterogeneous clusters. Application containerization is a proven technology to provide this portability for the *functionalities* of an application. However, to ensure *performance* portability, dependable verification of a cluster's performance under realistic workloads is required. Such verification is usually achieved through benchmarking the target environment and its storage in particular, as I/O is often the slowest component in an application. Alas, existing storage benchmarks are not suitable to generate cloud native workloads as they do not generate any storage control operations (*e.g.*, volume or snapshot creation), cannot easily orchestrate a high number of simultaneously running distinct workloads, and are limited in their ability to dynamically change workload characteristics during a run.

In this paper, we present the design and prototype for the first-ever Cloud Native Storage Benchmark—CNSBench. CNSBench treats control operations as first-class citizens and allows to easily combine traditional storage benchmark workloads with user-defined control operation workloads. As CNSBench is a cloud native application itself, it natively supports orchestration of different control and I/O workload combinations at scale. We built a prototype of CNSBench for Kubernetes, leveraging several existing containerized storage benchmarks for data and metadata I/O generation. We demonstrate CNSBench's usefulness with case studies of Ceph and OpenEBS, two popular storage providers for Kubernetes, uncovering and analyzing previously unknown performance characteristics.

1 Introduction

The past two decades have witnessed an unprecedented growth of cloud computing [54]. By 2020, many businesses have opted to run a significant portion of their workloads in public clouds [43] while the number of cloud providers has multiplied, creating a broad and diverse marketplace [1,17,18,25]. At the same time, it became evident that, in the foreseeable future, large enterprises will continue (i) running certain workloads on-premises (*e.g.*, due to security concerns), and (ii) employing multiple cloud vendors (*e.g.*, to increase cost-effectiveness or to avoid vendor lock-in). These *hybrid multicloud* deployments [41] offer the much needed flexibility to large organizations.

One of the main challenges in operating in a hybrid multicloud is workload portability—allowing applications to easily move between public and private clouds, and on-premises data centers [52]. Software containerization [10] and the larger cloud native [7] ecosystem is considered to be the enabler for providing seamless application portability [44]. For example, a container image [40] includes all user-space dependencies of an application, allowing it to be deployed on any container-enabled host while container orchestration frameworks such as Kubernetes [22] provide the necessary capabilities to manage applications across different cloud environments. Kubernetes's declarative nature [23] lets users abstract application and service requirements from the underlying site-specific resources. This allows users to move applications across different Kubernetes deployments—and therefore across clouds—without having to consider the underlying infrastructure.

An essential step for reliably moving an application from one location to another is validating its performance on the destination infrastructure. One way to perform such validation is to replicate the application on the target site and run an application-level benchmark. Though reliable, such an approach requires a custom benchmark for every application. To avoid this extra effort, organizations typically resort to using component-specific benchmarks. For instance, for storage, an administrator might run a precursory I/O benchmark on the projected storage volumes.

A fundamental requirement for such a benchmark is the ability to generate realistic workloads, so that the experimental results reflect an application's actual post-move performance. However, existing storage benchmarks are inadequate to generate workloads characteristic of modern *cloud native* environments due to three main shortcomings.

First, cloud native storage workloads include a *high number of control operations*, such as volume creation, snapshotting, etc. These operations have become much more frequent in cloud native environments as users, not admins [13,24], directly control storage for their applications. As large clusters have many users and frequent deployment cycles, the number of control operations is high [4,51,55].

Second, a typical containerized cluster hosts a *high number of diverse, simultaneously running workloads*. Although this workload property, to some extent, was present before in VM-based environments, containerization drives it to new levels. This is partly due to higher container density per node, fueled by the cost effectiveness of co-locating multiple tenants in a shared infrastructure and the growing popularity of microservice architectures [57,62]. To mimic such workloads, one needs to concurrently run a large number of distinct storage benchmarks across containers and coordinate their progress, which currently involves a manual and laborious process that becomes impractical in large-scale cloud native environments.

Third, applications in cloud native environments are *highly* *dynamic*. They frequently start, stop, scale, failover, update, rollback, and more. This leads to various changes in workload behavior over short time periods as the permutation of workloads running on each host change. Although existing benchmarks allow one to configure separate runs of a benchmark to generate different phases of workloads [47, 48], such benchmarks do not provide a versatile way to express dynamicity within a *single* run.

In this paper we present *CNSBench*—the first open-source Cloud Native Storage Benchmark capable of (i) generating realistic control operations; (ii) orchestrating a large variety of storage workloads; and (iii) dynamically morphing the workloads as part of a benchmark run.

CNSBench incorporates a library of existing data and metadata benchmarks (*e.g.*, fio [15], Filebench [60], YCSB [46]) and allows users to extend the library with new containerized I/O generators. To create realistic control operation patterns, a user can configure CNSBench to generate different control operations following variable (over time) operation rates. CNSBench can therefore be seen as both (i) a framework used for coordinating the execution of large number of containerized I/O benchmarks and (ii) a benchmark that generates control operations. Crucially, CNSBench bridges these two roles by generating the control operations to act on the storage used by the applications, thereby enabling the realistic benchmarking of cloud native storage.

As an example, consider an administrator evaluating storage provider performance under a load that includes frequent snapshotting. Conducting an evaluation manually requires the administrator to create multiple storage volumes, run a complex workload that will use that volumes (*e.g.*, a MongoDB database with queries generated by YCSB), and then take snapshots of the volumes while the workload runs. The same evaluation with CNSBench requires just that the administrator specify which workload to run, which storage provider to use, and the rate with which snapshots should be taken. CNSBench handles instantiating each component of the workload (*i.e.*, the storage volume, the MongoDB database, and the YCSB client) and then executing the control operations to snapshot the volume as the workload runs.

While developing CNSBench, we have also been building out a library of pre-defined workloads. The previous example uses one such workload, which consists of YCSB running against a MongoDB instance. If the administrator instead wanted to instantiate a workload not found in our library, it is easy to package an existing application into a workload that can be used by CNSBench. In that case, we would also encourage the administrator to contribute their new workload back to our library so that it could be used by a broader community.

To demonstrate CNSBench's versatility, we conducted a study comparing cloud native storage providers. We pose three questions in our evaluation: (A) How fast are different cloud storage solutions under common control operations? (B) How do control operations impact the performance of user applications? (C) How do different workloads perform when run alongside other workloads? We use Ceph [5] and OpenEBS [28] in our case study as sample storage providers. Our results show that control operations can vary significantly between storage providers (*e.g.*, up to 8.5× higher Pod creation rates) and that they can slow down I/O workloads by up to 38%.

In summary, this paper makes the following contributions:

1. We identify the need and unique requirements for cloud native storage benchmarking.
2. We present the design and implementation of CNSBench, a benchmark that fulfills the above requirements and allows users to conveniently benchmark cloud native storage solutions with realistic workloads at scale.
3. We use CNSBench to study the performance of two storage solutions for Kubernetes (Ceph and OpenEBS) under previously not studied workloads.

CNSBench is open-source and available for download from *https://github.com/CNSBench*.

2 Kubernetes Background

We implemented our benchmark for Kubernetes and so in the following sections we use many Kubernetes concepts to contextualize CNSBench's design and use cases. Therefore, we begin with a brief background on how Kubernetes operates.

Overview. A basic Kubernetes cluster is shown in Figure 1. It consists of control plane nodes, worker nodes, and a storage provider (among other components). Worker and control plane nodes run *Pods*, the smallest unit of workload in Kubernetes that consist of one or more *containers*. User workloads run on the worker nodes, whereas core Kubernetes components run on the control plane nodes. Core components include (1) the API server, which manages the state of the Kubernetes cluster and exposes HTTP endpoints for accessing the state, and (2) the scheduler, which assigns Pods to nodes. Typically, a Kubernetes cluster has multiple worker nodes and may also have multiple control plane nodes for high availability.

The storage provider is responsible for provisioning persistent storage in the form of *volumes* as required by individual Pods. There are many architectures, but the "hyperconverged" model is common in cloud environments. In this model, the storage provider aggregates the storage attached to each worker node into a single storage pool.

The state of a Kubernetes cluster, such as what workloads are running on what hosts, is tracked using different kinds of *resources*. A resource consists of a *desired state* (also referred to as its specification) and a *current state*. It is the job of that resource's *controllers* to reconcile a resource's current and desired states, for example, starting a Pod on node X if its desired state is "running on Node X". Pods and Nodes are examples of resources.

Persistent Storage. Persistent storage in Kubernetes is represented by resources called *Persistent Volumes* (PVs). Access to a PV is requested by attaching the Pod to a resource called a *Persistent Volume Claim* (PVC). Figure 1 depicts this process: ❶ A Pod that requires storage creates a PVC, specifying how much storage space it requires and which storage

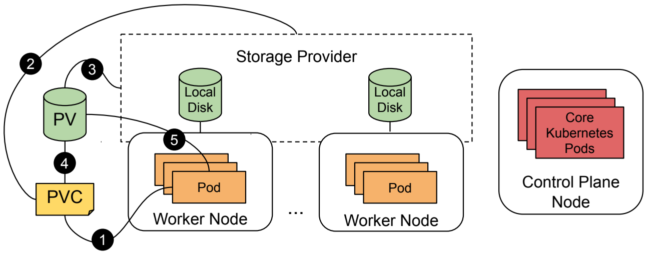


Figure 1: Basic topology of a Kubernetes cluster, with a single control plane node, multiple worker nodes, and a storage provider which aggregates local storage attached to each worker node. Also shows the operations and resources involved in providing a Pod with storage.

provider the PV should be provisioned from. ❷ If there is an existing PV that will satisfy the storage request then it is used. Otherwise, ❸ a new PV is provisioned from the storage provider specified in the PVC. A PVC specifies what storage provider to use by referring to a particular *Storage Class*. This class is a Kubernetes resource that combines a storage provider with a set of configuration options. Examples of common configuration options are what file system to format the PV with and whether the PV should be replicated across different nodes.

Once the PV has been provisioned, ❹ it is bound to the PVC, and ❺ the volume is mounted into the Pod's file system.

Kubernetes typically communicates with the storage provider using the *Container Storage Interface* (CSI) specification [9], which defines a standard set of functions for actions such as provisioning a volume and attaching a volume to a Pod. Before CSI, Kubernetes had to be modified to add support for individual storage providers. By standardizing this interface, a new storage provider needs only to write a CSI driver according to a well-defined API, to be used in any container orchetrator supporting CSI (*e.g.*, Kubernetes, Mesos, Cloud Foundry).

Although Kubernetes has good support for provisioning and attaching file and block storage to pods via PVs and PVCs, no such support exists for object storage. Therefore, CNSBench currently supports benchmarking only file and block storage.

3 Need for Cloud Native Storage Benchmarking

In this section we begin with describing the properties of cloud native workloads, which current storage benchmarks cannot recreate. We then present the design requirements for a cloud native storage benchmark.

3.1 New Workload Properties

The rise of containerized cloud native applications has created a shift in workload patterns, which makes today's environments different from previous generations. This is particularly true for storage workloads due to three main reasons: (i) the increased frequency of control operations; (ii) the high diversity of individual workloads; and (iii) the dynamicity of these workloads.

Control Operations. Previously infrequent, control operations became significantly more common in self-service cloud native environments. As an example, consider the frequent creations and deletions of containers in a cloud native environment. In many cases, these containers require persistent storage in the form of a storage volume and hence, several control operations need to be executed: the volume needs to be created, prepared for use (*e.g.*, formatted with a file system), attached to the host where the container will run (*e.g.*, via iSCSI), and finally mounted in the container's file system namespace. Even if a container only needs to access a volume that already exists, there are still at least two operations that must be executed to attach the volume to the node where the container will run and mount the volume into the container.

To get a better idea of how many control operations can be executed in a cloud native environment, consider these statistics from one container cluster vendor: in 2019 they observed that over half of the containers running on their platform had a lifetime of no more than five minutes [37]. In addition, they found that each of their hosts were running a median of 30 containers. Given these numbers, a modestly sized cluster of 20 nodes would have a new container being created every second on average. We are not aware of any public datasets that provide insight into what ratio of these containers require storage volumes. However, anecdotal evidence and recent development efforts [14] indicate that many containers do in fact attach to storage volumes.

In addition to being abundant, control operations, depending on the underlying storage technology, can also be data intensive. This makes them slow and increases their impact on the I/O path of running applications. For example, volume creation often requires (i) time-consuming file system formatting; (ii) snapshot creation or deletion, which, depending on storage design, may consume a significant amount of I/O traffic; (iii) volume resizing, which may require data migration and updates to many metadata structures; and (iv) volume reattachment, which causes cache flushes and warmups.

Now that data-intensive control operations are more common, there is a new importance to understanding their performance characteristics. In particular, there are two categories of performance characteristic that are important to understand: (1) How long does it take a storage provider to execute a particular control operation? This is important because in many cases, control operations sit on the critical path of the container startup. (2) What impact does the execution have on I/O workloads? This impact can be significant either due to the increased load on the storage provider or the particular design of the storage provider. For example, some storage providers freeze I/O operations during a volume snapshot, which can lead to a spike in latency for I/O operations [33].

Existing storage benchmarks and traces focus solely on data and metadata operations, turning a blind eye to control operations.

Diversity and Specialization. The lightweight nature of containers allows many different workloads to share a single server or a cluster [37]. Workload diversity is fueled by a variety of factors. First, projects such as Docker [11] and Kubernetes [22] have made containerization and cloud native computing more accessible to a wide range of users and organizations, which is apparent in the diversity of applications present in public repositories. For example, on Docker Hub [12] there are container images for fields such as bioinformatics, data science, high-performance computing, and machine learning—in addition to the more traditional cloud applications such as web servers and databases. Additionally, the popularity of microservice architectures has caused traditionally monolithic applications to be split up into many small, specialized components [62]. Finally, the increasingly popular serverless architecture [3], where functions run in dynamically created containers, takes workload specialization even further through an even finer-grained split of application components, each with their own workload characteristics.

The result of these factors is that the workloads running in a typical shared cluster (and on each of its individual hosts) have a highly diverse set of characteristics in terms of runtime, I/O patterns, and resource usage. Understanding system performance in such an environment requires benchmarks that recreate the properties of cloud native workloads. Currently, such benchmarks do not exist. Hence, realistic workload generation is possible only by manual selection, creation, and deployment of several appropriate containers (*e.g.*, running multiple individual storage benchmarks that each mimic the characteristics of a single workload). As more applications of all kinds adopt containerization and are broken into sets of specialized microservices, the number of containers that must be selected to make up a realistic workload continues to increase. Making this selection manually has become infeasible in today's cloud native environments.

Elasticity and Dynamicity. Cloud native applications are usually designed to be elastic and agile. They automatically scale to meet user demands, gracefully handle failed components, and are frequently updated. Although some degree of elasticity and dynamicity has always been a trait of cloud applications, the cloud native approach takes it to another level.

In one example, when a company adopted cloud native practices for building and operating their applications, their deployment rate increased from rolling out a new version 2–3 times per week to over 150 times in a single day [16]. Other examples include companies utilizing cloud native architectures to achieve rapid scalability in order to meet spikes in demand, for example in response to breaking news [27] or the opening of markets [2].

Currently, benchmarks lack the capability to easily evaluate application performance under these highly dynamic conditions. In some cases, benchmark users resort to creating these conditions manually to evaluate how applications will respond—for example manually scaling the number of database instances [46]. However, the high degree of dynamicity and diversity found in cloud native environments makes recreating these conditions manually nearly impossible.

3.2 Design Requirements

The fundamental functionality gap in current storage benchmarks is their inability to generate control-operation workloads representative of cloud native environments. At the same time, the I/O workload (data and metadata, not control operations) remains an important component of cloud native workloads, and is more diverse and dynamic than before. Therefore, the primary goal for a cloud native storage benchmark is to enable combining control-operation workloads and I/O workloads—to better evaluate application and cluster performance. This goal led us to define the following five core requirements:

1. I/O workloads should be specified and created independently from control workloads, to allow benchmarking (i) an I/O workload's performance under different control workloads and (ii) a control workload's performance with different I/O workloads.
2. It should be possible to orchestrate I/O and control workloads to emulate a dynamic environment that is representative of clouds today. In addition, it should be possible to generate control workloads that serve as microbenchmarks for evaluating the performance of individual control operations.
3. I/O workloads should be generated by running existing tools or applications, either synthetic workload generators like Filebench or real applications such as a web server with a traffic generator.
4. It should be possible for users to quickly configure and run benchmarks, without sacrificing the customizability offered to more advanced users.
5. The benchmark should be able to aggregate unstructured output from diverse benchmarks in a single, convenient location for further analyses.

A benchmark which meets these requirements will allow a user to understand the performance characteristics of their application and their cluster under realistic cloud native conditions.

4 CNSBench Design and Implementation

To address the current gap in benchmarking capabilities in cloud native storage, we have implemented the *Cloud Native Storage Benchmark*—CNSBench. Next, we describe CNSBench's design and implementation. We first overview its architecture and then describe the new Kubernetes *Benchmark* custom resource and its corresponding controller in more detail.

Overview. In Kubernetes, a user creates Pods (one of Kubernetes' core resources) by specifying the Pod's configuration in a YAML file and passing that file to the `kubectl` command line utility. Similarly, we want CNSBench users to launch new instances by specifying CNSBench's configuration in a YAML file and passing that file to `kubectl`. To achieve that, our CNSBench implementation follows the *operator design pattern*, which is a standard mechanism for introducing new functionality into a Kubernetes cluster [31]. In this pattern, a developer defines an *Operator* that comprises a custom resource and a controller for that resource. For our implementation of CNSBench, we defined a custom *Benchmark* resource and implemented a corresponding *Benchmark Controller*. Together, these two components form the *CNSBench Operator*. The Benchmark resource specifies the I/O and control workloads, which the controller is then responsible for running.

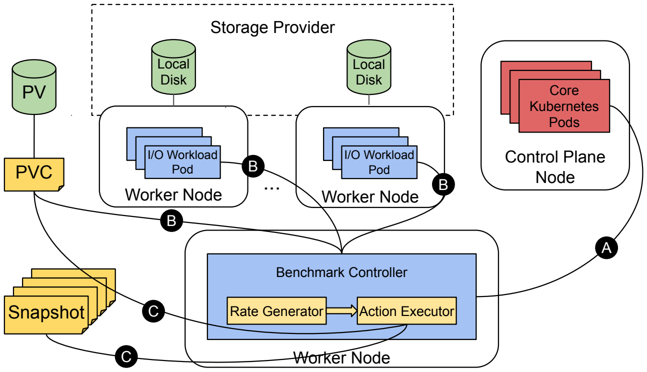


Figure 2: CNSBench overview with its components in blue

Figure 2 shows the Kubernetes cluster depicted in Figure 1 with added CNSBench components shown in blue. The overall control flow is as follow: Ⓐ The Benchmark controller watches the API server for the creation of new Benchmark resources. Ⓑ When a new Benchmark resource is created, the controller creates the resources described in the Benchmark's I/O workload: the *I/O Workload Pods* for running the workloads and the Persistent Volume Claims (PVCs) for the Persistent Volumes (PVs) against which the workloads are run. Ⓒ For running the control operation workload, the Benchmark includes a *Rate Generator*, which triggers an *Action Executor* in user-specified intervals to invoke the desired control operations (*actions*).

4.1 Benchmark Custom Resource

The Benchmark custom resource lets users specify three main benchmark properties: (1) the control operation workload; (2) the I/O workloads to run; and (3) where the output should be sent for collection.

Control operation workload. One of CNSBench's primary requirements is the ability to create realistic control workloads. However, microbenchmarks that purposefully stress only one component or operation of a system are also valuable (*e.g.*, for an in-depth analysis and point optimization of system performance). Useful insights can be derived, for instance, from a benchmark that executes some control operation at a regular interval. Our control workload specification satisfies both use cases, by making it easy to create simple control workloads without sacrificing the ability to define realistic ones.

In CNSBench, control workloads are specified using a combination of *actions* and *rates*. Actions execute operations, for instance *create resource* (*e.g.*, *create Pod or Volume*), *delete resource* (*e.g.*, *delete snapshot*), *snapshot volume*, and *scale resource* (*e.g.*, *scale database deployment*). Rates trigger associated actions at some interval. For our evaluations we used a simple rate which runs actions every T seconds, but more sophisticated rates could be implemented to enable the creation of more realistic control workloads. For example, given a set of cluster traces that logged when different operations were executed, a rate could be implemented that reads those traces and generates a control workload mimicking their specific operating conditions. Actions and rates are deliberately decoupled, so that these more sophisticated rates can be developed independently from CNSBench and then plugged in later.

I/O workload. Often, a benchmark's goal is to understand how a particular workload or set of workloads will perform under various conditions. The role of CNSBench's I/O workload component is to either instantiate those workloads or to instantiate a synthetic workload with the same I/O characteristics of a real workload. Specifying these I/O workloads requires defining all of the different resources (*e.g.*, Pods and PVCs) that must be created in order to run the I/O workload. This can be difficult and make benchmark specifications long and complex.

To ease the burden on users and to help them focus on the overall benchmark specification, rather than the specific details of the I/O workload, CNSBench separates the I/O workload specification from the rest of the benchmark specification. The I/O workload specification is defined using a *ConfigMap*—a core Kubernetes resource for storing configuration files and other free-form text. These files contain the specifications for the Pods that will run the I/O workloads, as well as specifications for supporting resources such as PVCs. In addition, they use metadata annotations to specify information such as what output files should be collected and what parsers should be used to process them. Since the specification uses a core Kubernetes resource, it can be accessed using standard Kubernetes tools from anywhere in the cluster.

Users specify which I/O workloads to run in a Benchmark custom resource using a *create resource* action that references (by name) the I/O workload to create. To enable reuse across various use cases and benchmarks, fields in an I/O workload specification can be parameterized and given a value when the workload is instantiated by a specific benchmark.

We are building out an open source Workload Library, available at *https://github.com/CNSBench/workload-library*, which offers pre-packaged I/O workloads including fio [15], Filebench [60], pgbench [32], YCSB [46], and RocksDB's db_bench [48].Ideally, most users will be able to find a suitable I/O workload in the library and hence, do not need to define their own. We hope that community members will contribute the I/O workloads that they develop to this library as well.

Control and data operations. In some cases control and I/O operations can be intertwined. For example, an increase in I/O operations can cause a workload to scale out, which in turn can execute more control operations. Reproducing such events with CNSBench would require a feedback mechanism that conveys to CNSBench information about the I/O operations executed by the I/O workloads. CNSBench's design and implementation do not preclude such mechanism but we leave its implementation to future work.

Benchmark output. Many of the results of a CNSBench benchmark will be generated by the I/O workload Pods. Collecting this output presents three challenges. First, Kubernetes currently lacks the ability to extract files from Pods in a clean and generic manner [19]. Second, the output produced by some tools can be large, especially for long-running processes that produce output throughout the run. Third, in our experience, many I/O

```
1  kind: Benchmark
2  metadata:
3    name: fio-benchmark
4  spec:
5    actions:
6      - name: fio Ⓓ
7        createObjSpec:
8          workload: fio Ⓐ
9          count: 3
10         vars:
11           storageClass: obs-r1 Ⓒ
12       outputs:
13         outputName: es
14     - name: snapshots
15       rateName: minuteRate
16       snapshotSpec:
17         actionName: fio Ⓓ
18         snapshotClass: obs-csi
19   rates:
20     - name: minuteRate
21       constantRateSpec:
22         interval: 60s
23   outputs:
24     - name: es
25       httpPostSpec:
26         url: http://es:9200/fio/_doc/
```

Listing 1: Sample Benchmark Custom Resource Specification

```
1  kind: ConfigMap
2  metadata:
3    name: fio Ⓐ
4  spec:
5    data:
6      pod.yaml: | Ⓑ
7        ...
8      pvc.yaml: | Ⓑ
9        ...
10       storageClass: {{storageClass}} Ⓒ
11       ...
```

Listing 2: Sample I/O workload specification

workloads produce output as unstructured text. This can make it difficult to analyze the results using tools such as Kibana [20], especially if the benchmark consists of multiple I/O workloads that all report results in a different unstructured output formats.

To address these issues, we allow I/O workload authors to specify which files should be collected from the workload Pods and to provide a parser script to process the output. Parsing the output allows large files to be reduced to a more succinct size and to output results in a standard fashion. The output files are collected and parsed using a helper container, described in more detail in Section 4.2. Parsers for common I/O benchmarking tools can be included in the Workload Library, either packaged with the tool's workload specification or as a standalone entry. For instance, we include parsers for fio and YCSB in the Workload Library.

The user specifies where the final, parsed results should be sent to in the *output* section of the Benchmark custom resource. Results do not all need to be sent to the same output. For instance, a benchmark with both fio and YCSB I/O workloads could send the fio results to one location and the YCSB results to another. The benchmark metadata, including the Benchmark resource specification and the start and end times, can be sent to an output as well.Currently CNSBench supports sending the results to a collection server via an HTTP POST request to a user-specified URL. Support for additional kinds of output, such as simply writing the output to a file, can be easily added.

In addition to workload output, it is also important to collect metrics such as Pod or Node resource utilization during a benchmark run. We defer the collection of these metrics to any of the many tools that are commonly used to collect such metrics in a Kubernetes cluster [38].

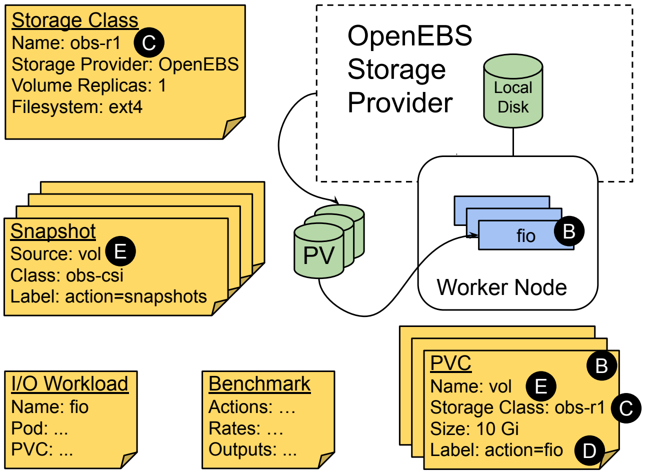


Figure 3: Subset of a Kubernetes cluster with a single worker node and a PV. Shows the CNSBench resources that are involved (the I/O Workload and Benchmark), as well as the core Kubernetes resources created by the CNSBench controller according to the Benchmark specification (the Snapshots, PVCs, PV, and workload Pods).

Example. An example Benchmark custom resource is shown in Listing 1 and an example of an I/O workload specification is shown in Listing 2. Due to space constraints, many of the details of the I/O workload specification are omitted. Figure 3 shows the Kubernetes resources that are created as a result of this Benchmark specification.

Lines 6–13 of Listing 1 specify the benchmark's I/O workload. Line 8 references the name of the I/O workload that should be run, labeled Ⓐ in both listings. Lines 6–11 of Listing 2 specify the resources that make up the I/O workload. These correspond to the Pods and PVCs in Figure 3 labeled Ⓑ.

I/O workload specifications can be parameterized to enable their reuse across different use cases and benchmarks. An example of this is on line 10 of Listing 2, where the PVC's Storage Class field is parameterized. Label Ⓒ in the two listings and in Figure 3 shows how this parameter is set in the Benchmark custom resource specification (line 11 in Listing 1), and then how that value is used in the workload's PVCs.

Lines 14-18 of Listing 1 specify a *snapshot volume* action. In Kubernetes, volume snapshots are created using a *Snapshot* resource which references a PVC to use as the source of the snapshot. The user indicates which action's PVCs should be snapshotted by referencing the target action by name (line 17 of Listing 1). Since all resources created by an action are labeled with that action's name, the controller can map an action name to a set of PVCs (label Ⓓ). These PVCs are then used as the source in the Snapshot resource (label Ⓔ). Additional examples can be found at *https://github.com/CNSBench/CNSBench*.

4.2 Benchmark Controller

The Benchmark Controller watches for newly created Benchmark objects and runs their specified actions. The controller has three main responsibilities: (1) triggering control operations; (2) synchronizing the individual benchmark workloads; and (3) collecting the output of the individual workloads.

Triggering control operations. When a new Benchmark resource is created, the Controller starts two *goroutines* (Go's equivalent of a thread) for each of the specified rates: one is responsible for generating the rate, and the other is responsible for running all of the actions using that rate. The rate goroutine uses a shared channel to tell the executor goroutine when it is time to run an action. As described in Section 4.1, decoupling the rates from the actions simplifies adding new kinds of rates or actions later.

Actions not tied to any rate are run by the controller as soon as the Benchmark resource is created. This is often how I/O workloads are instantiated, since they often use a long running process that generates I/O throughout the benchmark's duration.

Synchronizing workloads. In many cases, I/O workloads require an initialization step such as loading data into a database or creating a working set of files. When there are multiple I/O workloads being run, some workloads can finish their initialization step faster than others and begin running their main workload earlier. This can cause misleading and inconsistent results. If the purpose of the benchmark is to evaluate a storage provider's performance under the concurrent load of ten read-heavy I/O workloads, then all ten should start at the same time.

To synchronize the I/O workloads, CNSBench leverages Kubernetes' *initialization containers* feature. Pods have a list of initialization containers which are executed in order, each one running to completion before the next one starts. The Pod's main containers do not run until all of the initialization containers have completed. CNSBench assumes that a workload's initialization step has been put into an initialization container, which is the responsibility of the I/O workload's author. Although this is usually a straightforward task, it is an example of why separating the I/O workload specifications from the rest of the Benchmark specifications is useful: it allows users to select existing workloads from the Workload Library and not worry about how their workload's initialization is implemented.

When the Benchmark controller instantiates the I/O workloads, it adds an additional *synchronization container* at the end of the list of initialization containers. This container runs a script that queries the Kubernetes API server for the status of each instance of the I/O workload and checks to see if all of their initialization containers have completed (all except for the other synchronization containers). Once all of the non-synchronization initialization containers have completed, the script exits and the synchronization containers stop successfully, allowing Kubernetes to run each Pods' main container. Since all instances of the I/O workload have this synchronization container added, all instances begin running their main containers simultaneously. Many workloads support running for a set amount of time, so synchronizing the finish of each workload is generally not an issue.

Output and metrics collection. As described in Section 4.1, I/O workload authors can specify which files to extract from a workload's Pods and provide a script to parse those files. Extracting these files from the workload Pods is difficult since there is no standard interface for doing so [19]. The approach used by the official Kubernetes command-line client `kubectl` involves running the `tar` utility inside the target container, and does not work after the container has finished running [21].

To work around these difficulties, the controller modifies the workload Pod to add both a helper container responsible for running the parser script, and also a volume mounted by both the helper and workload containers. The I/O workload author must ensure that the workload's output is written to this volume, which will be mounted at `/output`. Similar to how the synchronization container works, the helper container queries the Kubernetes API server to find out when the workload container has finished; thereafter, the output is ready to be parsed.

5 Evaluation

To demonstrate both the need for and the utility of CNSBench, we ran several benchmarks to look at different aspects of cloud native storage performance. We examine the performance of individual control operations, the impact that control operations have on I/O workloads, and the impact that different combinations of I/O workloads can have on overall performance.

5.1 Methodology

To evaluate our benchmark, we instantiated an 11-node Kubernetes v1.18.6cluster in an on-premises OpenStack environment: one control plane node and 10 workers. Each worker node is a virtual machine with 4 vCPUs, 8GB of RAM, and 384GB of locally attached storage. The control plane node is a VM with 4 vCPUs, 12GB of RAM, and 100GB of local storage. The VM hosts were located in multiple racks, with racks connected via a 10Gbps network and individual hosts connected to the top of rack switch via 1Gbps links.

We used two storage providers: OpenEBS and Ceph. Our requirements for the storage systems were that they be open-source, free, and not based on cloud-as-a-service model—so we could install and test them locally, and to enable more repeatable results. Additionally, they had to have a CSI driver. These requirements eliminated many existing storage systems. Out of the remaining options, we selected Ceph and OpenEBS due to their popularity.

OpenEBS [28] is a new storage provider built specifically to be cloud native. OpenEBS uses the *Container Attached Storage* paradigm [8], where controllers that provision volumes and manage features such as data replication, themselves run in containers. This provides storage with all of the advantages of the cloud native methodology, such as agility and flexibility. It also enables the storage to be managed like any other resource in a cloud native cluster. We used OpenEBS's cStor storage engine version 2.0.0.

Ceph [64] is a widely used file storage system that is built on top of the RADOS object store [65]. We used the Rook operator for Ceph [35], which handles the deployment and management of a Ceph cluster. The Rook management layer allows Ceph to be managed in a cloud native fashion, using Kubernetes objects and standard Kubernetes management tools. We used Rook version 1.4.1 and Ceph version 15.2.4, with Ceph's BlueStore storage backend.

Both Ceph and OpenEBS provide storage by aggregating the local storage attached to each cluster node. Volumes are provisioned from this combined storage pool and are formatted with Ext4 prior to being attached to a Pod. Ceph and OpenEBS both come with CSI drivers that interface with Kubernetes.

Both OpenEBS and Ceph also offer volume replication for high availability use cases. With volume replication, data written to a volume by a Pod is transparently copied across several volume replicas, which are ideally situated in different availability zones. This enables the cluster to tolerate the loss of one or more hosts—depending on the replication factor—without suffering any data loss. The trade-off is that volume replication often comes at a cost of increased I/O latencies and an increase in network and disk utilization.

Ceph has an additional high availability mechanism using erasure coding, which encodes data into chunks using a forward error-correction code and then replicates those chunks. The use of a forward error-correction code means that fewer replicas are needed to provide the same availability guarantees, and hence less disk space is needed overall. However, erasure coding uses more CPU and RAM than basic data replication.

In our experiments, we use Ceph and OpenEBS in three ways: without replication, in triple-replication mode, and Ceph (only) in erasure-coded mode (ec). In addition to Ceph and OpenEBS, in some evaluations we used a null storage provider that implements the CSI functions involved in provisioning and attaching volumes. The null driver simply returns success to most CSI functions without performing actual work. The null driver does, however, maintain a list of provisioned volumes so the *ListVolumes* CSI function returns an accurate result. We use the null driver as a baseline to show the maximum possible performance of the underlying Kubernetes cluster.

Each evaluation was conducted five times and unless otherwise noted has a standard deviation of less than 20%.

5.2 Performance of Control Operations

In Section 3.1 we described the importance of control operations in cloud native workflows. In this section, we demonstrate how the performance of these operations can vary across different storage providers and configurations. We looked at two common storage control operations: volume provisioning and attaching.

Our goal was to time how long it took each storage provider configuration to provision a volume and attach that volume to a Pod. To do so, we timed how long it took to create and run new Pods that were attached to volumes. The time to create and run a Pod with an attached volume includes the time taken by the storage provider to provision and then attach that volume. Any additional overhead related to running the Pod is constant across storage configurations.

We ran this test with 1, 10, 20, 30, 40, 50, 60, and 70 parallel Pod creations. Each test ran for five minutes, where we maintained a fixed parallelism level N by starting a new Pod whenever one Pod was created; there were always N Pods in the process of being created. The workload run by each Pod simply exited immediately, so Pods finished running as soon as they started.

We repeated each run five times. Figure 4 shows CDFs for Pod start time across all of the Pods created during each of the five runs, for six storage provider configurations. We show CDFs only for three degrees of parallelism (1, 30, and 70) because the CDFs for the intermediate parallelism values follow the trends that are visible from these three. Figure 5 shows the overall volume creation and attachment rate per minute for different parallelism levels. These rates are averaged across each of the five runs and had a standard deviation under 11% of the mean, except for OpenEBS which had standard deviations of up to 30% of the mean. This higher standard deviation can be attributed to the polling architecture which is used throughout Kubernetes and OpenEBS [29], which causes some actions to take sometimes significantly different amounts of time depending on which side of the poll the resource becomes available.

As expected, Pod creation is fastest with the null storage provider. The storage provider configurations with no replication are slightly faster than their replicated counterparts. This is also expected, since volumes with replication require additional resources to be allocated during provisioning.

As the number of simultaneous Pod creations increases, we noticed that subsets of Pods took an increasingly long time to start (see Figure 5). Eventually, each of the six storage configurations reached a point where its Pod creation rate plateaus. Note that Pod creation goes through three states: initially it is in a "Pending" state before it can be assigned to a Node. Once the Kubernetes scheduler has assigned the Pod a Node to run on, it moves it to a "Creating" state where container images are downloaded and volumes are mounted. Then, the Pod enters the "Running" state.

As an initial investigation, we counted how many Pods were in each state to identify the bottleneck. We observed that for the null storage provider and the three Ceph configurations, the rate that Pods moved from "Pending" to "Creating" and then from "Creating" to "Running" equalizes when the number of simultaneous Pod creations reaches around 50. At this point, increasing the number of simultaneous Pod creations only increased the number of Pods in the "Pending" state, and did not increase the overall Pod creation rate.

The situation is different for the two OpenEBS configurations.

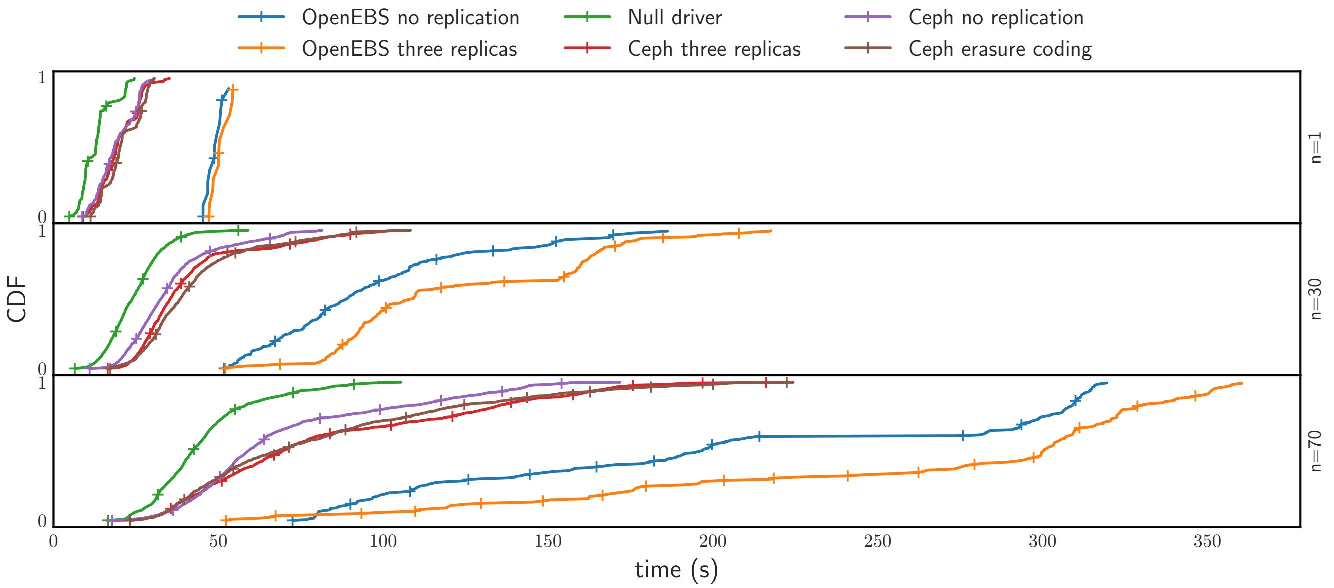


Figure 4: CDFs of time required to create and attach volumes for different storage provider configurations. n is the number of simultaneous volume creations. For all storage configurations, increasing the number of simultaneous volume operations increased the average time to create and attach an individual volume.

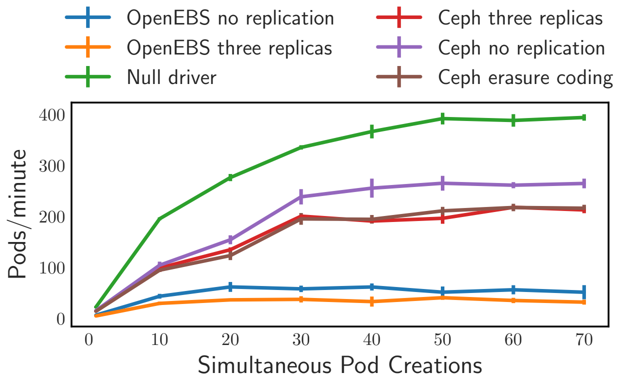


Figure 5: Volume creations and attachments per minute, for different numbers of simultaneous operations. The vertical lines at each point shows the standard deviation for volume creation and attachment rate at that point.

As shown in Figure 5, these configurations plateau at a lower rate of around 30 simultaneous Pod creations. When observing the Pod transitions for these configurations, we saw that the rate at which Pods moved from “Creating” to “Running” was low compared to the rate that Pods moved from “Pending” to “Creating” resulting in all Pods being in either “Creating” or “Running” states throughout the test. The Pods in the “Creating” state were all waiting for OpenEBS to finish provisioning and attaching a volume for the Pod. So, increasing the number of simultaneous Pod creations did not increase the overall Pod creation rate, since that rate was limited by how fast OpenEBS was able to provision and attach volumes.

From these experiments we see that although all three storage providers have scalability limits in terms of how many simultaneous Pod creations they support, the source of their limits appear to be different. Whereas the null storage provider and Ceph are limited by the scheduling stage of Pod creation, OpenEBS is limited by its own volume creation and attachment rate.

Overall, the experiment shows that there can be significant differences in the performance of control operations across different storage providers and configurations. This highlights the need to systematically benchmark these kinds of operations to understand their bottlenecks and improve upon them. Conducting this experiment without CNSBench would require starting different numbers of Pods using a tool such as `kubectl`. Whenever a Pod finishes being created, a new one needs to be started, which would be cumbersome to coordinate manually.

5.3 Impacts on I/O Workloads

In this section, we demonstrate the impact that control operations, in particular snapshotting a volume, can have on the I/O workload that uses the volume. As described in Section 3.1, control operations are executed far more often in cloud native environments than they are elsewhere. Snapshotting is especially common and users take frequent snapshots of their volumes for a number of reasons: periodically, during a long running task to checkpoint progress, prior to making some significant change so rollback to a known good point is possible, or to protect themselves against attacks such as ransomware.

Although previously these operations were executed too infrequently to have a noticeable effect on an I/O workload, this is no longer guaranteed to be the case in cloud native environments. Due to differences in the design and architecture of different storage providers, the degree to which these control operations impact an I/O workload can vary significantly.

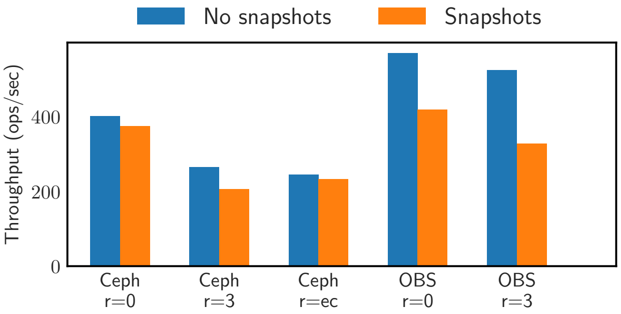


Figure 6: Effect of snapshotting on I/O workload. r=0 indicates zero volume replicas, r=3 indicates three volume replicas, and r=ec indicates erasure coding.

To evaluate the impact of snapshotting operations, we used CNSBench to run three instances of MongoDB [26] with ten clients each. The clients ran YCSB Workload A [46] (consisting of a mix of reads and updates) for twenty minutes to reach steady state; the volumes holding the MongoDB databases were snapshotted every thirty seconds.

Figure 6 shows the per-client throughput in terms of operations per second for five storage provider configurations, with and without snapshotting. The throughput values are averaged across all thirty YCSB clients.

Overall the results show that snapshotting reduces the throughput across all configurations. The decrease in throughput is more noticeable for OpenEBS (27% and 38% for zero and three volume replica configurations, respectively) than for Ceph (up to 22% for three volume replicas but as low as 5% and 6% for erasure coding and zero replication configurations, respectively). We found that although the average throughput decreased with snapshotting across all OpenEBS YCSB clients, the decrease was more pronounced for some clients than others. For those clients, we observed that the maximum latency reported by YCSB was much higher than the average maximum latency. In addition, these clients reported extended periods (30+ seconds) when zero operations were executed. During these periods with zero operations, the Mongo database reported that some queries were taking a long time to be processed.

One possible explanation is the fact that OpenEBS quiesces and suspends I/O while a snapshot operation is in progress [30]. During that time, any writes issued by Mongo cannot complete. Some of these periods of suspended I/O lasted several seconds, which could explain the periods when no operations could be executed by the clients and the reduction in overall throughput. We analyzed the distribution of throughputs for all clients and found a long tail with many clients timing out after several quiescing periods, then retrying.

Ceph does not quiesce [6] I/O during a snapshot and we did not observe the same spikes in maximum latency that we observed with OpenEBS. We did observe some of the same periods with zero completed operations that we saw with OpenEBS, and also observed the same complaints of slow queries from the Mongo logs. One possible explanation is that there was an increased load on Ceph: with snapshots, around four times as many objects were created in the underlying RADOS object pools compared to no snapshotting.

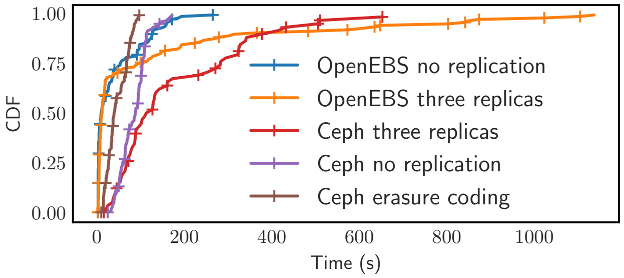


Figure 7: CDF of snapshot creation times for different storage provider configurations.

To create a new snapshot in Kubernetes, users create a Snapshot resource. This resource is created immediately. However, the underlying snapshot is not necessarily ready right away. Figure 7 shows a CDF of how long it took after creating a new Snapshot resource until the storage provider reported that the snapshot was actually ready to be used.

Both Ceph and OpenEBS implement copy-on-write snapshots, so it is expected that for most storage configurations, snapshots became available nearly as fast as the Snapshot resources were created. However, some configurations exhibited a long tail where snapshots took several minutes to become ready. For example, although the median time to become ready for snapshots on OpenEBS with three volume replicas was 12 seconds, 10% took longer than 310 seconds and 5% took longer than 702 seconds. The interface between Kubernetes and the storage provider's CSI driver is the Kubernetes Snapshot Controller [39]. When we analyzed the logs for this container, we found that the CreateSnapshot CSI calls for some snapshots were timing out due to slow I/O on the underlying disks used by the storage provider. For some unlucky snapshot instances the CreateSnapshot call would repeatedly timeout, resulting in snapshot creation times of several *minutes*. One interesting observation was that even when the Snapshot Controller aborted its CreateSnapshot call (due to the timeout), the storage provider would still finish creating a snapshot. However, the Kubernetes Snapshot Controller had already timed out, thus missing the successful response from the storage provider.

Running this experiment without CNSBench would require specifying and creating each of the resources required to run MongoDB and YCSB (Pods, PVCs, Services, etc.). Then, while YCSB ran, the user would need to create snapshots of each of the volumes being used by specifying the snapshot resources in YAML and instantiating the resources with a tool such as `kubectl`.

5.4 Orchestration

One of the core CNSBench capabilities is to make it easy to run various mixes of I/O workloads. This is needed since the alternative, to manually choose and assemble workloads together

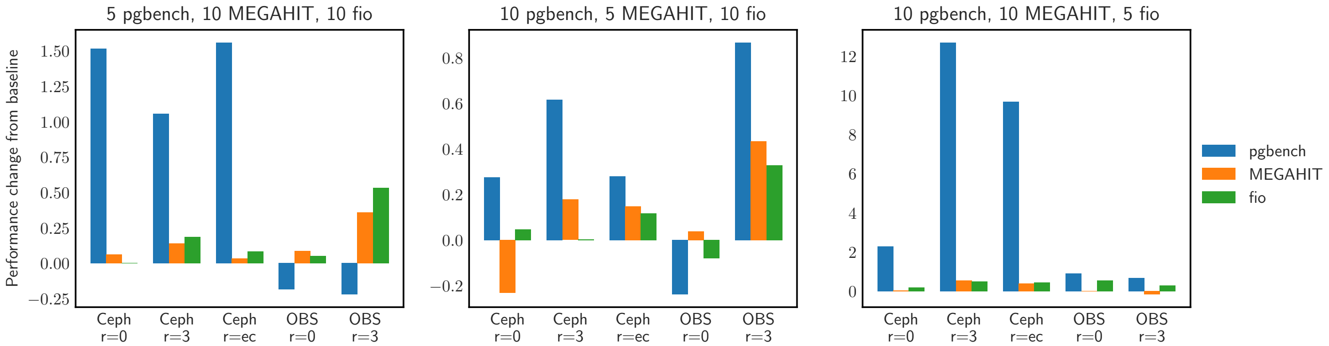


Figure 8: Change in performance compared to baseline, for three different ratios of I/O workload on five different storage configurations.

to form a representative combined workload, is infeasible due to the diversity of workloads in cloud native environments.

One potential use case for this task is to determine which storage configuration is best suited for a particular set of workloads. Another might be to help influence scheduling decisions, such as which workloads to run simultaneously.

To demonstrate CNSBench's orchestration capabilities, we ran multiple instances of three different workloads: (1) MEGAHIT [53], a bioinformatics tool that processes genetic data; (2) fio [15] for generating an intense I/O workload of mixed random reads and writes; and (3) the PostgreSQL [34] database with a workload generated using its benchmark tool pgbench [32]. Each instance of the PostgreSQL workload ran a distinct pair of database and client. Out of each of the workloads, fio was the most I/O intensive, followed by pgbench. Both fio and pgbench spent most of their time waiting for I/O, whereas MEGAHIT was mostly CPU bound.

We tested four different workload mixes: a baseline with ten independent instances of each workload, and then three additional mixes with ten instances of two of the workloads and five of the third. For MEGAHIT and fio we measured the total time to run a fixed load; for pgbench we measured the average throughput after running for ten minutes. This was necessary since the different storage provider configurations performed significantly different, so it would be impractical to evaluate using a fixed amount of work.

Figure 8 shows the changes in runtime and throughput, normalized to the baseline values, for different workload mixes and storage providers. The baseline throughputs for pgbench are 5.2, 0.37, 0.38, 85, and 16 operations per second for Ceph (no volume replication), Ceph (three volume replicas), Ceph (erasure coding), OpenEBS (no volume replication), and OpenEBS (three volume replicas), respectively. MEGAHIT had baseline runtimes of 309, 910, 609, 185, and 324 seconds, and fio had baseline runtimes of 427, 816, 699, 923, and 2478 seconds, respectively.

The largest increase in performance of 3.2× is for pgbench when the number of fio instances is reduced. This makes sense: the Ceph storage configurations shows the largest increase in pgbench performance, since pgbench's baseline performance on Ceph is much worse than on OpenEBS so there is a larger potential for improvement. Also, pgbench and fio are both I/O-intense workloads, *i.e.*, reducing the number of fio instances would help pgbench, but not MEGAHIT.

Benchmark	**Lines**
Volume creation and attachment § 5.2	19
YCSB and MongoDB, no snapshots § 5.3	35
YCSB and MongoDB, with snapshots § 5.3	54
Multiple workloads § 5.4	72–117

Table 1: Number of lines needed to specify CNSBench benchmarks used during evaluation.

The workload that had the overall smallest impact on performance is MEGAHIT. This is also expected as fio and pgbench are mainly I/O bound while MEGAHIT is mainly CPU bound and hence reducing the number of MEGAHIT instances does not free up significant I/O resources.

These results demonstrate the variability in storage provider performance, and the utility of being able to easily compose and run diverse sets of workloads at various mixes. Conducting this experiment on Kubernetes without CNSBench would require creating all of the resources required for a workload (PVCs, Pods, Services, etc.) manually, for example by specifying them in YAML and passing the YAML to a tool such as `kubectl`. To run multiple instances of a workload, the user would need to specify multiple copies of each resource, making sure to give each copy a unique name and updating references to resources accordingly. This would need to be repeated for each workloads mix being evaluated. Synchronizing the start of each workload would need to be done manually. For example, to synchronize the start of multiple MongoDB+YCSB workloads the user would need to first start each MongoDB database pod, then wait for the databases to be initialized, and then run each instance of their YCSB benchmark.

5.5 Benchmark Usability

Requirement 4 in Section 3.2 states that CNSBench should be easy for users to configure and run. Although usability is often subjective, one metric that can be used to estimate ease of use is the number of lines necessary for specifying a workload. Table 1 shows the number of lines needed to specify each of the benchmarks used in this evaluation section.

Overall a user can specify the complex, distributed, and diverse workloads in just 19–117 lines of configuration. The workloads used in Section 5.4 require slightly longer specifications as they contain multiple instances of the same sub-workload, which currently results in duplication in the CNSBench's benchmark specification. We plan to eliminate such repetitions in the future to make using CNSBench even simpler.

6 Related Work

Classic storage benchmarks. Storage benchmarking is an old and complex topic with many applicable techniques and intricate nuances [59]. Therefore, it is not surprising that the array of tools for benchmarking and corresponding studies is extensive. Filebench [60], fio [15], SPEC SFS [36], and IOZone [45] are just a few examples of popular file system benchmarks. For a comprehensive survey of file system and storage benchmarks we refer the reader to a study by Traeger et al. [63].

The majority of such benchmarks generate a single, stationary workload per run, which is not representative of cloud native environments. Few benchmarks have built-in mechanisms to dynamically increase the load, in order to discover the peak throughput where diminishing returns (*e.g.*, due to thrashing) begin to take over. For example, measuring NFS throughput via SPEC SFS [58] and process scheduling throughput using AIM7 [61].

Filebench [42, 60] comes with several canned configurations [56] and even has its own Workload Modeling Language (WML) [66]. It, however, is not distributed (cannot run in a coordinated manner across multiple containers) and, though WML is flexible for encoding stationary workloads, is still limited in creating dynamically changing workloads. In our experience, adding support for distributed and temporally varying workloads to Filebench's WML is a difficult task. Therefore, in CNSBench, we exploited the orchestration capabilities of cloud native environments and delegated these tasks to a higher level (*i.e.*, the CNSBench controller and the Kubernetes orchestrator itself). This further allowed us to support any existing benchmarks as canned I/O generators.

RocksDB [48] is a popular key-value store with canned, preconfigured workloads using a db_bench driver to create random/sequential reads/writes and mixes thereof. One can run these workloads in any order and configure their working-set size. However, that is still a manual process with little flexibility, and no support for control operations (which is true for the previously mentioned benchmarks as well).

Object storage benchmarks. In recent years the need to test the performance of cloud storage has motivated academia and industry to develop several micro-benchmarks for that task such as YCSB [46] and COSBench [67]. YCSB is an extensible workload generator that evaluates the performance of different cloud-serving key-value stores. COSBench measures the performance of cloud object storage services and comes with plugins for different cloud providers. Unlike these benchmarks, CNSBench focuses on workloads that run in containers and require a file system interface.

Cloud native benchmarks. TailBench [50] provides a set of interactive macro-benchmarks: web servers, databases for speech recognition, and machine translation systems to be executed in the cloud. Similarly, DeathStarBench [49] is a benchmark suite for microservices and their hardware-software implications for cloud and edge systems. Both TailBench and DeathStarBench target cloud applications and are not explicitly storage benchmarks.

7 Conclusion

Although measuring storage performance was always an important topic, its relevance has escalated in recent years due to the increased demand to reliably move containerized applications across clouds. Furthermore, I/O patterns of applications have evolved, exhibiting higher density, diversity, dynamicity, and specialization than before. Perhaps most importantly, storage services now experience a high rate of *control operations* (*e.g.*, volume creation, formatting, snapshotting), which directly impact the performance of applications that call them and indirectly influence the I/O of other applications in a cluster. Existing storage benchmarks, however, are not able to model these new cloud native scenarios and workloads holistically and faithfully.

In this paper we presented the design of CNSBench—a storage benchmarking framework that containerizes legacy I/O benchmarks, orchestrates their concurrent runs, and concurrently generates a stream of control operations. CNSBench is easy to configure and run, while still being versatile enough to express a high variety of real-world cloud native workloads. We used CNSBench to evaluate two cloud native storage backends—OpenEBS and Ceph—and found several differences. For example, our evaluation shows that the maximum rate of control operations varies significantly across storage technologies and configurations by a factor of up to 8.5×.

Future work. We plan to work on extending the library of I/O workloads with I/O "kernels" that represent microservices, and also improve the benchmark specification language to make the syntax more concise and avoid having to duplicate sub-workloads. Further, we will work on collecting I/O and control operation traces from production environments, analyze them, and create corresponding profiles for CNSBench. Our longer term plans including finding and fixing performance bugs using CNSBench, and even developing our own efficient storage solution.

We hope our benchmark will be adopted by storage and cloud native communities, and look forward to contributions.

8 Acknowledgments

We thank the USENIX FAST anonymous reviewers and our shepherd Xiaosong Ma for their helpful feedback. We also thank Mike Ferdman for the use of his OpenStack cluster. This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; and NSF awards CCF-1918225, CNS-1900706, CNS-1729939, and CNS-1730726.

References

[1] Amazon Web Services (AWS). *https://aws.amazon.com/*.

[2] Bloomberg: An early adopter's success with Kubernetes at scale. *https://www.cncf.io/case-studies/bloomberg/*.

[3] Building Applications with Serverless Architectures. *https://aws.amazon.com/lambda/serverless-architectures-learn-more/*.

[4] Building large clusters. *https://kubernetes.io/docs/setup/best-practices/cluster-large/*.

[5] Ceph. *https://ceph.io/*.

[6] Ceph Snapshots. *https://docs.ceph.com/en/latest/rbd/rbd-snapshot/*.

[7] Cloud Native Computing Foundation. *https://www.cncf.io/*.

[8] Container Attached Storage is Cloud Native Storage (CAS). *https://www.cncf.io/blog/2020/09/22/container-attached-storage-is-cloud-native-storage-cas/*.

[9] Container Storage Interface (CSI) Specification. *https://bit.ly/3bqQX4b*.

[10] Containerization. *https://www.ibm.com/cloud/learn/containerization*.

[11] Docker. *https://docker.com/*.

[12] Docker Hub. *https://hub.docker.com/*.

[13] Dynamic Provisioning and Storage Classes in Kubernetes. *https://bit.ly/2Uh3Qbw*.

[14] Ephemeral volumes. *https://kubernetes.io/docs/concepts/storage/ephemeral-volumes/*.

[15] fio. *https://github.com/axboe/fio*.

[16] Going Cloud Native: 6 essential things you need to know. *https://www.weave.works/technologies/going-cloud-native-6-essential-things-you-need-to-know/*.

[17] Google Cloud. *https://cloud.google.com/*.

[18] IBM Cloud. *https://www.ibm.com/cloud*.

[19] Improve kubectl cp, so it doesn't require the tar binary in the container #58512. *https://github.com/kubernetes/kubernetes/issues/58512*.

[20] Kibana. *https://www.elastic.co/kibana*.

[21] kubectl cp to work on stopped/completed pods #454. *https://github.com/kubernetes/kubectl/issues/454*.

[22] Kubernetes. *https://kubernetes.io/*.

[23] Kubernetes Object Management. *https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/*.

[24] Kubernetes Storage. *https://kubernetes.io/docs/concepts/storage/*.

[25] Microsoft Azure. *https://azure.microsoft.com/*.

[26] MongoDB. *https://www.mongodb.com/*.

[27] News UK Keeps New Content and Capabilities Coming Fast with Amazon EKS and New Relic. *https://blog.newrelic.com/product-news/news-uk-content-capabilities-amazon-eks-new-relic/*.

[28] OpenEBS. *https://openebs.io/*.

[29] OpenEBS cStor CSI driver. *https://github.com/openebs/cstor-csi/blob/master/pkg/driver/controller_utils.go#L243*.

[30] OpenEBS replication.c. *https://github.com/openebs/istgt/blob/replication/src/replication.c\#L1958*.

[31] Operator pattern. *https://kubernetes.io/docs/concepts/extend-kubernetes/operator/*.

[32] pgbench. *https://www.postgresql.org/docs/10/pgbench.html*.

[33] Portworx Kubernetes Snapshots and Backups. *https://docs.portworx.com/portworx-install-with-kubernetes/storage-operations/kubernetes-storage-101/snapshots/*.

[34] PostgreSQL. *https://www.postgresql.org/*.

[35] Rook. *https://rook.io/*.

[36] SPEC SFS 2014. *https://www.spec.org/sfs2014/*.

[37] Sysdig 2019 Container Usage Report. *https://sysdig.com/blog/sysdig-2019-container-usage-report/*.

[38] Tools for monitoring resources. *https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/*.

[39] Volume Snapshot & Restore - Kubernetes CSI Developer Documentation. *https://kubernetes-csi.github.io/docs/snapshot-restore-feature.html*.

[40] What is a Container? *https://www.docker.com/resources/what-container*.

[41] A hybrid and multicloud strategy for system administrator. Technical Report #F21608_0220, Red Hat, 2020.

[42] George Amvrosiadis and Vasily Tarasov. Filebench github repository, 2016. *https://github.com/filebench/filebench/wiki*.

[43] Andrew Bartels, Dave Bartolett, John Rymer, Matthew Guarini, Charlie Dai, and Alyssa Danilow. The public cloud market outlook, 2019 to 2022: Public cloud growth continues to power tech spending. Technical report, Forrester, July 2019.

[44] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[45] Don Capps and Tom McNeal. Analyzing NSF client performance with IOzone. NFS Industry Conference, 2002.

[46] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[47] Paul Dix. Benchmarking LevelDB vs. RocksDB vs. HyperLevelDB vs. LMDB Performance for InfluxDB. *https://bit.ly/365KSL2*, 2014.

[48] Facebook. RocksDB. *https://rocksdb.org/*, September 2019.

[49] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An Open-source

Benchmark Suite for Microservices and their Hardware-software Implications for Cloud & Edge Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[50] Harshad Kasture and Daniel Sanchez. TailBench: A Benchmark Suite and Evaluation Methodology for Latency-critical Applications. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016.

[51] Sachin Katti, John Ousterhout, Guru Parulkar, Marcos Aguilera, and Curt Kolovson. Scalable control plane substrate.

[52] Stefan Kolb. *On the Portability of Applications in Platform as a Service*, volume 34. University of Bamberg Press, 2019.

[53] Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. In *Bioinformatics*, 2015.

[54] Jack McElwee and Allan Krans. Public cloud benchmark: First calendar quarter 2020. Technical report, Technology Business Research, July 2020.

[55] Alex Merenstein, Vasily Tarasov, Ali Anwar, Deepavali Bhagwat, Lukas Rupprecht, Dimitris Skourtis, and Erez Zadok. The case for benchmarking control operations in cloud native storage. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[56] Filebench pre-defined personalities, 2016. *http://filebench.sourceforge.net/wiki/index.php/Pre-defined_personalities*.

[57] Frank Della Rosa. Implementation of microservices architecture hastens across industries. Technical Report #US46108319, IDC, 2020.

[58] SPEC SFS®2014. *https://www.spec.org/sfs2014/*.

[59] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2011.

[60] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *USENIX ;login:*, 41(1), 2016.

[61] AIM Technology. AIM multiuser benchmark - suite VII version 1.1. *http://sourceforge.net/projects/aimbench*, 2001.

[62] Johannes Thönes. Microservices. *IEEE Software*, 32(1), 2015.

[63] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P Wright. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2), 2008.

[64] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 307–320, Seattle, WA, November 2006. ACM SIGOPS.

[65] Sage Weil, Andrew Leung, Scott Brandt, and Carlos Maltzahn. RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage (PDSW)*, 2007.

[66] Filebench workload model language (WML), 2016. *https://github.com/filebench/filebench/wiki/Workload-Model-Language*.

[67] Qing Zheng, Haopeng Chen, Yaguang Wang, Jian Zhang, and Jiangang Duan. COSBench: Cloud Object Storage Benchmark. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2013.

Concordia: Distributed Shared Memory with In-Network Cache Coherence

Qing Wang, Youyou Lu*, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu*

Tsinghua University

Abstract

Distributed shared memory (DSM) is experiencing a resurgence with emerging fast network stacks. Caching, which is still needed for reducing frequent remote access and balancing load, can incur high coherence overhead. In this paper, we propose CONCORDIA, a DSM with fast in-network cache coherence backed by programmable switches. At the core of CONCORDIA is FLOWCC, a hybrid cache coherence protocol, enabled by a collaborative effort from switches and servers. Moreover, to overcome limitations of programmable switches, we also introduce two techniques: (*i*) an ownership migration mechanism to address the problem of limited memory capacity on switches and (*ii*) idempotent operations to handle packet loss in the case that switches are stateful. To demonstrate CONCORDIA's practical benefits, we build a distributed key-value store and a distributed graph engine on it, and port a distributed transaction processing system to it. Evaluation shows that CONCORDIA obtains up to 4.2×, 2.3× and 2× speedup over state-of-the-art DSMs on key-value store, graph engine and transaction processing workloads, respectively.

1 Introduction

Distributed Shared Memory (DSM) enjoyed a short heyday (circa early 1990s) by offering a unified global memory abstraction. Yet, it later failed to entertain a greater audience due to the unsatisfying performance atop the low-speed network [28]. Recent advancement in high-performance network technologies prompts a new look into DSM. With optimizations across the network stack (e.g., RDMA), the bandwidth can now surge to 100Gbps or even 200Gbps [1], and the latency drops to less than 2μs [14]. Researchers take this opportunity to systematically rethink the DSM design and have achieved a string of successes in both academia and industry [4, 20, 47, 51, 61]. For example, Microsoft has developed FaRM [4], a fast RDMA-based DSM; on top of FaRM, engineers have built key-value stores [25], distributed transaction engines [26, 59], and a graph database called A1 [19] (A1 is used by Microsoft's Bing search engine).

But there is still at least one more hurdle to cross: cache coherence. Caching is still important to DSM for obtaining competitive performance (e.g., local memory has 2-4× higher throughput and 12× lower latency against RDMA [60]) and balancing load (e.g., caching the hot data to multiple servers [27]). Yet, distributed caching always comes with coherence, which is notorious for its complexity and overhead[1]. Specifically, coherence incurs excessive communication for coordination, such as tracking cache copies' distribution, invalidation and serializing conflicting requests (i.e., requests targeting the same cache block), thereby severely impacting the overall throughput. For example, even in the RDMA-based ccNUMA [27], introducing a slight dose of cache coherence by increasing the write ratio from 0 to 5% can reduce the performance by 50%.

The advent of programmable switches has changed the landscape of various classic system architectures [23, 33, 34, 38, 41, 42, 56, 57, 65, 67, 68]. Here, we argue that leveraging a customizable switch to reduce the overhead of cache coherence in DSM is promising. First, since a switch is the centralized hub of inter-server communication, reconfiguring it to handle cache coherence can significantly reduce coordination between servers. Second, the latest switches can usually process several billion packets per second, enabling them to quickly handle coherence requests. Third, the switch owns an on-chip memory, which allows storing cache block metadata in the switch. Moreover, the on-chip memory can support atomic read-modify-write operations [33], thereby easing the effort to synchronize conflicting coherence requests.

However, simply shoehorning all cache coherence logic into switches can be impractical. First, cache coherence protocols are too complicated for programmable switches [13, 35] which only have limited expressive powers due to their restricted programming model and demanding timing requirements [49]. Second, the on-chip memory is usually small (e.g., 10-20 MB), making it unlikely to accommodate the metadata of the entire cache set. Third, failure handling can be tricky. Cache coherence protocol generally uses a state machine to perform the state transitions. Hence, if deployed on switches, a common fault, such as a packet loss, could lead a switch into an erroneous state (e.g., deadlock by repeated locking due to retransmission).

In this paper, we present the CONCORDIA, a high-performance rack-scale DSM with in-network cache coherence. The core of CONCORDIA is FLOWCC (in-Flow Cache Coherence), a cache coherence protocol that is jointly supported by switches and servers. In FLOWCC, switches, as the data plane, serialize conflicting coherence requests and

*Jiwu Shu and Youyou Lu are the corresponding authors. {shujw, luyouyou}@tsinghua.edu.cn

[1] "There are only two hard things in Computer Science: cache invalidation and naming things."—Phil Karlton

multicast them to the destinations, reducing coordination between servers. Servers, on the other hand, act as the control plane that performs state transitions and sends corresponding updates to switches. Specifically, we utilize the on-chip memory to store metadata of cache blocks and implement reader-writer locks for concurrency control.

CONCORDIA also designs an ownership migration mechanism to manage the metadata of cache blocks. The mechanism moves the ownership of cache blocks between switches and servers dynamically according to the coherence traffic, namely only keeping the hot ones in the switches. To handle packet loss, we make all operations both in servers and switches idempotent, guaranteeing exactly-once semantics.

We implement CONCORDIA with Barefoot Tofino switches and commodity servers and evaluate it with three real datacenter applications: distributed key-value storage, distributed graph computation and distributed transaction processing. Experimental results show that CONCORDIA obtains up to 4.2×, 2.3× and 2× speedup on key-value store, graph engine and transaction processing, respectively, over two state-of-the-art RDMA-based DSMs, Grappa [51] and GAM [20].

To sum up, we make the following contributions:

- We propose CONCORDIA, a high-performance rack-scale DSM that incorporates an in-network cache coherence protocol, namely FLOWCC.
- We design an ownership migration mechanism and idempotent operations to overcome the restriction of current programmable switches.
- Evaluation shows that CONCORDIA gains significant performance improvement against two state-of-the-art DSMs in various applications.

2 Background

In this section, we provide background on cache coherence protocols and programmable switches.

2.1 Cache Coherence Protocols

Cache coherence protocols are studied extensively in both the systems and architecture communities [50, 53]. We briefly describe the two main protocol types used in DSMs below.

Directory-based protocols keep track of servers that hold cache copies. Each data block has a home node[2] that keeps the states and locations of cache copies. The home node updates and notifies the state of the data block to all cache copies, and serializes conflicting cache coherence requests. The limitation is that the home nodes induce extra round trips, and the home nodes for hot data are under heavy load.

Snooping protocols do not keep track of cache copies. Instead, they broadcast cache coherence requests to all the servers. The limitation is that the broadcast can easily overwhelm the network, and wastes the CPU cycles of servers that do not contain the requested cache block. In addition, since ordered and reliable broadcast systems are equivalent to consensus [24], it is intractable to coordinate conflicting cache coherence requests in snooping protocols.

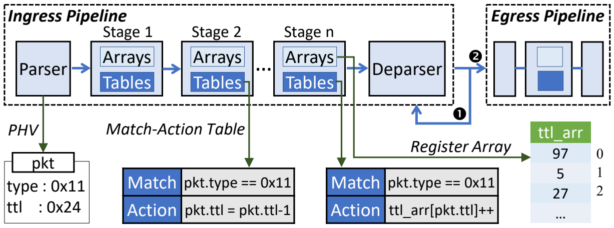


Figure 1: Pipelines in Programmable Switches. In this figure, for a packet whose `type` field is `0x11`, the switch reduces its `ttl` (i.e., time to live) field via the first match-action table, and then updates the register array `ttl_arr` via the second table. The `k`th item in `ttl_arr` records the number of packets whose `ttl` field is `k`.

2.2 Programmable Switch

Emerging programmable switches like Barefoot Tofino [2] provide programmable capacity. Such a switch follows reconfigurable match table (RMT) architecture [18] and usually has multiple ingress and egress pipelines. Each pipeline contains multiple stages, and packets are processed by these stages in sequential order, as shown in Figure 1.

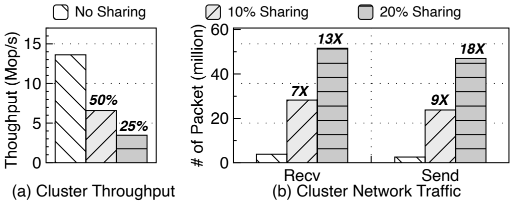
Developers can program three components for switches: the parser, register arrays and match-action tables. The parser defines packet formats. A register array is a collection of memory items (e.g., `tll_arr` in the Figure 1); we can read, write, and conditionally update these items via index numbers (i.e., positions). A match-action table specifies (*i*) a *match key* from a set of packet fields (e.g., in the Figure 1, the `type` field is the match key of both tables), and (*ii*) a set of actions, each of which consists of instructions about modifying packet fields and register arrays. A register array or match-action table belongs to only one stage of a certain pipeline, which can be specified by developers.

When a packet arrives at an ingress port, the parser analyzes it and generates a packet header vector (PHV), which is a set of header fields (e.g., UDP port). The PHV is then passed to match-action tables in the ingress pipeline in a stage-by-stage manner. If the specific fields of the PHV match an entry in a match-action table, the corresponding action is executed. Before leaving the ingress pipeline, the packet is reassembled by the deparser. Then there are two cases (realized by setting a metadata field): ❶ the packet is *resubmitted*, i.e., it re-enters the ingress pipeline. ❷ the packet is switched to the egress pipeline, experiencing processing similar to the ingress pipeline, and it finally is emitted via an egress port.

We summarize two properties of an RMT pipeline, which can simplify the design of stateful protocols in switches:

P1. Atomicity Property. Due to the pipelined architecture, only one packet is processed in a stage at any time. In other words, operations for multiple register arrays in the same stage are atomic.

[2] In this paper, we use "server" and "node" interchangeably.



(a) Cluster Throughput (b) Cluster Network Traffic

Figure 2: Impact of Cache Coherence.

P2. Ordering Property. Suppose there are two packets A, B, and they are being processed in stage S_A and S_B, respectively. If $S_A < S_B$ in time t (e.g., A in stage 1 and B in stage 2), $S_A < S_B$ holds at any time after t within this pipeline.

3 Motivation

This section revisits cache coherence under fast network environments via an experiment and discusses challenges in designing an in-network cache coherence protocol.

3.1 Revisit Cache Coherence with Fast Network

To understand the performance impact of cache coherence, we use a micro benchmark to evaluate GAM [20], a state-of-the-art RDMA-based DSM backed by a directory-based protocol. In this benchmark, each node in the 8-node cluster launches four threads to issue 8-byte write/read operations to global memory with a write ratio of 50%. Here, we define the sharing ratio as the percentage of operations that access shared data. By varying the ratio from 0 (i.e., no sharing) to 20%, we can see, in Figure 2(a), that the throughput degrades by 75%. Further, we collect the packets received and sent by all nodes (Figure 2(b)). The number of packets across network increases dramatically (up to 18×) when more data is shared because of expensive distributed communication in cache coherence protocols. From the benchmark, we conclude that *even with fast networks, existing cache coherence protocols dramatically limit system performance*.

3.2 Challenges

There are three challenges to design a fast cache coherence protocol using programmable switches:

- *The mismatch between the complexity of cache coherence protocols and the restricted expressive power of programmable switches*. Existing cache coherence protocols are intricate, because of complex state transitions in the face of concurrent and asynchronous requests. However, the expressive power of programmable switches is limited: all procedures must be represented as match-action tables. Moreover, the processing pipeline must meet the hardware resource and timing requirements of switch ASICs [34]. For example, tables with dependencies must be placed in different stages.
- *Limited switch memory capacity*. Current programmable switches have limited on-chip memory capacity (10-20MB) [37]. Furthermore, we need to reserve some memory to serve normal network protocols. Thus, switches are unable to manage the coherence of all cache blocks.
- *Packet loss*. Switch buffer overflow can cause packet loss. Existing in-network systems such as NetCache [34] and NetChain [33] rely on client-side retries to address this problem. However, it is hard to guarantee exactly-once semantics by simply retransmitting lost packets in DSMs, considering that a cache coherence request always involves multiple round trips and packets.

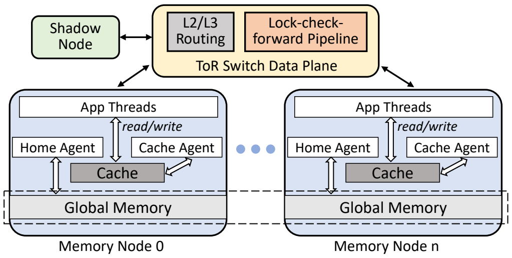


Figure 3: CONCORDIA Overview. Global memory from all memory nodes constitutes a logically unified address space (the dashed box).

4 CONCORDIA Overview

CONCORDIA is a rack-scale DSM that leverages programmable switches to accelerate cache coherence. Figure 3 shows its overview, which consists of a set of memory nodes, a top-of-rack (ToR) switch, and a shadow node.

Memory nodes. Memory nodes run distributed applications and provide memory for them. Each memory node divides its DRAM into two parts: a global memory and a private local write-back cache. The global memory from all memory nodes constitutes a logically unified 64-bit address space: | node id: 16-bit | offset: 48-bit | ; each data block has a constant *home node*, which is specified by the *node id* field. The local cache is organized in *cache blocks*, which are the unit of data transfer between the local cache and global memory. A cache block is uniquely identified by its *tag* (e.g., the tag of a 4KB cache block is the highest 52 bits of its address). There are three components in a memory node: (*i*) *application threads* execute application logic and access global memory via linearizable write/read interfaces, which interact with the local cache; (*ii*) the *home agent* manages part of the global memory space within its node; (*iii*) the *cache agent* performs invalidation and data transfers for data that is cached on its memory node.

Switch. In addition to routing normal packets using standard L2/L3 protocols, the switch is responsible for executing part of the cache coherence protocol (e.g., serializing and multicasting requests) via the lock-check-forward (LCF) pipeline.

Shadow node. The shadow node helps migrate the ownership of cache blocks between the switch and home agents by recording coherence traffic of cache blocks.

As in other coherence protocols, each cache block may be in one of three states, depending on how it is cached and whether it is shared: *Unshared* (no node has it in the local cache), *Shared* (some nodes share it with read permission), and *Modified* (one node caches an exclusive copy with write permission). This state is reflected in the cache block's *global status*. The *copyset* is the set of nodes that hold the corre-

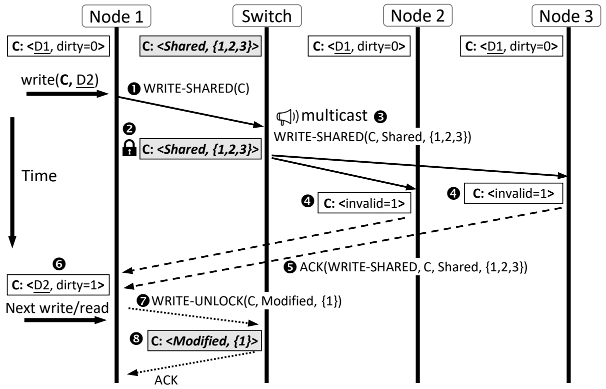


Figure 4: An Example of FLOWCC. An application thread (i.e., requester) in node 1 writes cache block C with data D2.

sponding cache block. We denote the global status and copyset of a cache block as its *global metadata*. *Owners* manage the coherence for a cache block, and store its global metadata. Ownership (and global metadata) can be migrated between the home node's home agent and the switch, depending on how actively the cache block is shared.

4.1 Key Ideas

1) Separating data and control planes. In CONCORDIA, the switch only does what it is proficient at, i.e., routing packets as the data plane; it multicasts cache coherence requests and serializes conflicting ones via in-network locks (locking can be regarded as controlling route paths of conflicting requests). In contrast, servers, as control planes, perform state transitions and send corresponding updates to the switch. Such a separation simplifies the data plane design of the switch to overcome its restricted expressive power (§5.1).

2) Unifying switch- and server-based coherence processing. Due to the limited on-chip memory capacity of the switch, CONCORDIA lets the switch only manage the coherence of hot data, and resorts to servers for processing the cold data. According to the coherence traffic that a cache block induces, CONCORDIA dynamically migrates its ownership between servers and the switch (§5.2).

3) Minimizing the number of modifications to switch state. For every operation that modifies switch state, i.e., the values stored in the switch's register arrays, we must deal with the corresponding case of packet loss, at the cost of consuming precious hardware resources of the switch and complicating the system design. Thus, we strive to minimize the number of switch state modifications. Specifically, in each coherence event, i.e., the process of executing a cache coherence request, the switch only modifies its state twice: ① acquiring locks; ② releasing locks and installing new metadata of the targeted cache block. We make the two modifications idempotent, to handle packet loss (§5.3).

4.2 Example

We illustrate the overall operation of FLOWCC protocol with an example, as shown in Figure 4.

An application thread (i.e., requester) in node 1 issues a write to cache block C, whose ownership is in the switch. Since C is already cached by node 1 and is *not* dirty, the requester generates a cache coherence request (i.e., WRITE-SHARED) with the tag of C, and sends it to the switch (❶).

After receiving the request, the switch acquires the write lock for cache block C, to prevent conflicting cache coherence events (❷). Then, it multicasts the request to cache agents in node 2 and 3, according to the value of the copyset (❸). Of note, the multicast requests contain the global metadata of C (i.e., global status *Shared* and the copyset).

Upon receiving multicast requests, the cache agents in node 2 and 3 invalidate their local cache block copies by marking them as invalid (❹). Then they send ACKs, which contain the global metadata of cache block C, to the requester (❺).

The requester waits for ACKs from the other cache agents that are indicated in the copyset of ACKs (i.e., {2, 3}). Once all ACKs are received, the requester installs new data into its local cache and marks it as dirty (❻). At this time, the cache block C is in a coherent state in CONCORDIA. Finally, the requester sends an asynchronous unlock request to the switch (❼), to allow concurrent or subsequent coherence requests targeting the same cache block to make progress. This unlock request contains new global metadata of the cache block. The switch releases the corresponding lock, and uses information in the unlock request to install new global metadata (❽).

Overall, this cache coherence request is processed in *only a single round-trip* (compared with directory-based protocols, note that unlock is asynchronous) with *much less network traffic* (compared with snoop protocols, note that requests are only multicast to memory nodes that hold data copies).

The example above can experience a host of undesirable situations, for which we elaborate our solutions in §5. These situations include: (*i*) the lock is occupied; (*ii*) the request is invalid when entering the switch's pipeline; (*iii*) the ownership is not in the switch; (*iv*) a network packet is dropped.

5 CONCORDIA Design In Depth

In this section, we first describe FLOWCC protocol (§5.1). Next, we detail how CONCORDIA migrates ownership (§5.2) and handles packet loss (§5.3). Finally, we discuss practical issues of CONCORDIA (§5.4).

5.1 FLOWCC Protocol

At the core of CONCORDIA is FLOWCC, a write-invalidate protocol, which divides coherence responsibility between the switch and servers. The switch serializes conflicting requests and multicasts them to correct home/cache agents via the lock-check-forward (LCF) pipeline. Servers perform state transitions and update the switch data plane.

5.1.1 Protocol Details

In FLOWCC, a cache coherence event is handled collaboratively by switches and servers in the following four phases:

Field	Meaning
type	type of the cache coherence request
tag	tag of the requested cache block
node_id	node id of the requester
global_status	global status of the requested cache block
copyset	the set of nodes that hold the cache block
is_data_provider	whether the receiver is the data provider
is_in_switch	whether the switch manages the coherence

Table 1: Packet Format.

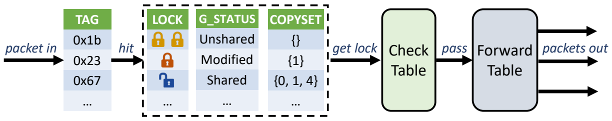


Figure 5: Lock-check-forward Pipeline. The three arrays in the dashed box belong to the same stage; the orange lock is a read lock, and the red one is a write lock. The switch just finished the lock phase of two READ-MISS requests to the cache block with `0x1b` tag.

➤ *Request Generation*

Application threads (i.e., requesters) access global memory by issuing write/read operations to the local cache, and generate *cache coherence requests* in case of cache miss and cache eviction. Table 1 shows the packet format of requests. There are five types of cache coherence requests (i.e., *type* field): ①② WRITE-MISS/READ-MISS for cache miss, ③WRITE-SHARED when issuing a write operation to a clean cache block, ④⑤EVICT-MODIFIED/EVICT-SHARED when evicting a dirty/clean cache block. The *tag* and *node_id* fields identify the requested cache block and the node of the requester, respectively. The last four fields are filled by the switch.

➤ *Lock-check-forward Pipeline*

In the LCF pipeline, we store a reader-writer lock and global metadata for each cache block. By leveraging this state, the LCF pipeline serializes conflicting requests and multicasts them to destinations.

On-chip state storage. Figure 5 shows the LCF pipeline, which consists of four register arrays and two match-action tables. The *TAG* array is a hash table that stores the *tag* of cache blocks by using 64-bit slots. The *LOCK* array contains 16-bit reader-writer locks; for each lock, the most significant bit indicates a writer, and the other 15 bits count the number of readers (i.e., 2^{15} concurrent readers at most). Lock and unlock operations are supported by conditional update of the register array. The *G-STATUS* array records the global status of cache blocks with 8-bit slots. The *COPYSET* array maintains cache block copysets by using a bitmap structure. The last three register arrays are placed in the same stage, so they can be manipulated together in an atomic manner (P1 in §2.2). The *check table* verifies the validity of cache coherence requests. The *forward table* routes requests to the correct cache/home agents.

Request processing. The LCF pipeline processes a request in three steps: (*i*) lock the requested cache block for concurrency control. (*ii*) check the request's validity. (*iii*) forward the request to its final destinations.

When a request enters the LCF pipeline, the switch first hashes the *tag* field of the request into an index number, and then uses the index number to search the *TAG* array for the *tag* (i.e., check if *TAG*[hash(*tag*)] equals *tag*). On failure, the switch forwards the request to its home agent [3] and sets the *is_in_switch* field to false. After finding the *tag*, the switch tries to acquire the read lock if the request is READ-MISS; otherwise, acquires the write lock. Note that we protect EVICT-SHARED with write locks, since two concurrent EVICT-SHARED requests to the same cache block may cause different state transitions (detailed later, see Figure 6). In parallel, the corresponding global metadata (i.e., values in *G-STATUS* and *COPYSET* array) is filled into the request. The switch returns a failed ACK to the requester when failing to acquire the lock.

Match Key pkt.type	Action
READ-MISS	check_succ ← pkt.node_id ∉ pkt.copyset
WRITE-MISS	
WRITE-SHARED	check_succ ← pkt.node_id ∈ pkt.copyset ∧ pkt.global_status == *Shared*
EVICT-SHARED	
EVICT-MODIFIED	check_succ ← pkt.node_id ∈ pkt.copyset ∧ pkt.global_status == *Modified*

Table 2: Check Table. If the `check_succ` is true (i.e., the request is valid), the switch passes the request to the forward table; otherwise, the switch releases the lock and sends a failed ACK to the requester.

Match Key (pkt.type, pkt.global_status)	Action
(READ-MISS, *Unshared*)	forward to the request's home agent
(WRITE-MISS, *Unshared*)	
(READ-MISS, *Modified*)	forward to the cache agent in pkt.copyset
(WRITE-MISS, *Modified*)	
(READ-MISS, *Shared*)	select a cache agent in pkt.copyset as the data provider, and forward to it
(WRITE-MISS, *Shared*)	multicast to cache agents in pkt.copyset, select a cache agent as the data provider
(WRITE-SHARED, *Shared*)	multicast to cache agents in pkt.copyset except pkt.node_id [4]
(EVICT-SHARED, *Shared*)	return to the requester
(EVICT-MODIFIED, *Modified*)	

Table 3: Forward Table. Requests multicast/forwarded to cache agents (except for READ-MISS requests) indicate invalidation.

Once the switch acquires the lock successfully, it passes the request to the check table (shown in Table 2), to filter out invalid requests caused by concurrent events. Let us consider an example of invalid requests: immediately after a requester issues a WRITE-SHARED request `W` for cache block `A`, the cache agent in the same node invalidates `A`; it is possible that, due to `W` being delayed by OS scheduler or network, the global status of `A` is no longer *Shared* or the requester no longer holds `A` when `W` enters the LCF pipeline. Thus, to ensure that only valid requests are forwarded to destination nodes, the switch checks the *global_status* and *copyset* in the request. If the check fails, the switch resubmits the request to release the acquired lock, and then sends a failed ACK to the requester. Compared with server-based mechanisms, such

[3] For simplicity, we denote *the home agent of a request or a cache block* as the home agent that resides in the requested cache block's home node.

[4] If copyset only contains the requester, the switch returns an ACK.

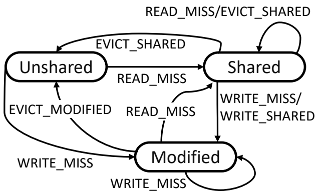


Figure 6: State Transitions of Global Status. Requesters generate global status according to this diagram, and piggyback it on unlock requests. When evicting a clean cache block (i.e., EVICT-SHARED), the global status changes from *Shared* to *Unshared* if the copyset is empty; otherwise, the global status remains unchanged.

an in-network check reduces the load of servers (checking validity of requests and sending failed ACKs).

If the request passes the check table successfully, the forward table (see Table 3) sends it to its final destinations according to the *(type, global_status)* pair. Specifically, there are three cases: (*i*) If no copy of the requested cache block exists, i.e., *global_status* is *Unshared*, the switch routes the request to its home agent for fetching data in global memory. (*ii*) If cache block copies need to be invalidated, the switch multicasts the request to the cache agents in the *copyset*. (*iii*) For an eviction request, the switch returns it to the requester directly, since the corresponding lock has been acquired.

To support efficient cache-to-cache data transfer, we adopt an in-network selection mechanism. For READ-MISS/WRITE-MISS requests, if shared nodes exist (i.e., *copyset* is not empty), the switch selects a cache block data provider among them using a load balancing strategy and sets *is_data_provider* field in the corresponding request. The current load balancing strategy is to randomly choose a data provider from shared nodes; compared with other complex strategies, the random selection does not consume any precious on-chip memory.

➤ ***Invalidation and Data Transfer***

Cache agent. Upon receiving the request, the cache agent invalidates corresponding cache block for WRITE-MISS/WRITE-SHARED. Then, the cache agent sends an ACK to the requester. If the cache agent is a data provider, it also transfers cache block data.

Home agent. Upon receiving a request, if *is_in_switch* field is *true*, the home agent replies to the requester with an ACK, which contains the corresponding cache block data from global memory. The remaining cases, i.e., the switch does not own the ownership of the requested cache block, are discussed in §5.2.

Note that all ACKs from cache/home agents carry the global metadata of the requested cache block, so as to help requesters reach coherence.

➤ ***Requester-driven Coherence Control***

The requester waits for ACKs from home/cache agents, and then performs state transitions and updates the switch data plane. Specifically, upon receiving a failed ACK, the requester retries the read/write operation. Otherwise, the requester waits for all ACKs needed for reaching coherence. Then, for READ-MISS/WRITE-MISS, the requester installs cache block data; for EVICT-MODIFIED, it writes back the cache block data to the home node. After that, the requester generates new global metadata (i.e., *copyset* and *global status*): the new copyset is (*i*) $\{id\}$ for WRITE-MISS/WRITE-SHARED or (*ii*) $cs_{old} + \{id\}$ for READ-MISS or (*iii*) $cs_{old} - \{id\}$ for eviction requests, where cs_{old} is the copyset in the ACKs and *id* is the node id of the requester. The new global status is generated according to the state transitions diagram in Figure 6. Finally, the requester sends an unlock request to the switch, with the new global metadata. The unlock request is asynchronous, enabling the requester to execute subsequent read/write operations immediately.

On receiving an unlock request, the switch releases the lock (i.e., modifies the *LOCK* array), and updates the global metadata (i.e., values in *G-STATUS* and *COPYSET* arrays). For read lock release, the new value in the *COPYSET* array is the union of the old one and the *copyset* field in the unlock request, considering maybe there are concurrent READ-MISS requests. Atomicity property (P1 in §2.2) guarantees the atomicity of update to these three arrays, since they are in the same stage.

5.1.2 Correctness

FLOWCC is based on the state transitions of classic MSI protocols. Compared with directory-based MSI protocols, it has two main changes: ① leveraging in-switch locks to serialize conflicting requests; ② leveraging the switch to multicast invalidation messages. Here, we prove FLOWCC is correct by proving the following two invariants are correct [27, 50]. We assume all messages are reliable (§5.3 describes how CONCORDIA handles packet loss).

INVARIANT 1. Exactly one writer can update a memory location at a time (*Single-Writer-Multiple-Readers Invariant*).

PROOF. Before getting the write permission of a cache block, a thread must acquire the in-switch write lock and revoke the write/read permission of other servers via invalidation; thus, only one thread can write a cache block at any time.

INVARIANT 2. If a cache block is valid, it must hold the most recent value updated by writers (*Data-Value Invariant*).

PROOF. On write requests, the switch invalidates all the copies by multicasting messages and only the requester owns the most recent data at the end; on read requests, the most recent data is populated into the requester's local cache. Thus, any operations manipulate the latest data.

5.2 Ownership Migration

To overcome limited on-chip memory capacity in the switch, CONCORDIA explicitly limits the number of cache blocks that the switch manages coherence for, using an *ownership migration mechanism*. Specifically, CONCORDIA migrates ownership between home agents and the switch dynamically: if a cache block has heavy coherence traffic, its home agent migrates its ownership to the switch. Similarly, if a cache block only induces light coherence traffic, its ownership is migrated in the opposite direction.

Home agents handle cache coherence requests for cache blocks they manage (the *is_in_switch* field in these requests is *false*). Specifically, home agents process cache coherence requests in the same way as the LCF pipeline of the switch: they store global metadata for multicast and use reader-writer locks to serialize conflicting requests.

5.2.1 Migration Requests

We design two type of requests for ownership migration.

- ADD-TO-SWITCH. When a home agent needs to migrate the ownership of a cache block to the switch, it acquires the write lock for the cache block (to prevent conflicting requests), and then sends an ADD-TO-SWITCH request to the switch. The request consists of the cache block's *tag*, *global status* and *copyset*. The switch inserts these three values into *TAG*, *G-STATUS*, and *COPYSET* array, respectively. Ordering property P2 (in §2.2) ensures other requests will see all the updates or none. The home agent releases the lock after receiving an ACK from the switch.
- REMOVE-FROM-SWITCH. When a home agent is informed by the shadow node to migrate the ownership of a cache block from the switch to itself (see §5.2.2), it sends a REMOVE-FROM-SWITCH request to the switch with the *tag* of the cache block. Upon receiving the request, the switch acquires the cache block's write lock in the *LOCK* array. On success, the switch resubmits the request. When receiving the resubmitted request, the switch clears the *tag* in the *TAG* array; then, it releases the lock and sends an ACK with up-to-date *global status* and *copyset* (i.e., global metadata) to the home agent. The home agent stores these two values locally.

5.2.2 Migration Workflow

Figure 7 shows the ownership migration of a cache block.
Home agents ➟ Switch. Home agents identify hot cache blocks managed by themselves and migrate their ownership to the switch. We divide time into continuous epochs (e.g., 10ms), and each home agent records the hotness of cache blocks in the current epoch. If a cache coherence request needs to be multicast to *n* cache agents, the hotness of corresponding cache block is incremented by *n*. At the end of an epoch, the home agent migrates the top-k (e.g., 1000) hottest cache blocks to the switch by issuing ADD-TO-SWITCH.
Switch ➟ Home agents. We introduce a *shadow node* to collect hotness statistics for cache blocks managed by the switch. The shadow node duplicates the switch's *TAG* array into its in-memory array called *SHADOW-TAG*. Cache agents record the number of invalidations for cache blocks managed by the switch, and report their statistics to the shadow node at the end of each epoch. The shadow node applies these statistics to *SHADOW-TAG*. The shadow node scans the *SHADOW-TAG* array periodically to remove the coldest cache blocks by informing home agents to issue REMOVE-FROM-SWITCH for these cache blocks.

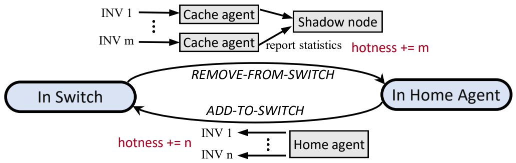


Figure 7: Ownership Migration of a Cache Block. INV means invalidation messages; the migration is performed by home agents.

Handling collisions. Since the switch manages the *TAG* array as a hash table (recall §5.1.1), the candidate locations of an ADD-TO-SWITCH operation may be occupied by other cache blocks (i.e., hash collisions). In such a case, the switch returns a failed ACK to the home agent that issues the ADD-TO-SWITCH; the home agent will retry the ADD-TO-SWITCH at the next epoch (if the corresponding cache block is still hot). The switch copies the ACK of every ADD-TO-SWITCH to the shadow node: if the ACK indicates success, the shadow node adds the corresponding *tag* into the *SHADOW-TAG*; otherwise, it removes the coldest cache block in the conflicting locations. By leveraging the centralized shadow node to handle collisions, CONCORDIA can ensure that though each home agent independently chooses its hottest cache blocks, the switch can manage the globally hottest cache blocks.

5.3 Packet Loss Handling

In the FLOWCC protocol, the switch is stateful due to storing locks and global metadata; thus, traditional end-to-end mechanisms, e.g., TCP retransmission, can not handle packet loss correctly. We make operations both in servers and the switch idempotent, to address this problem[5].

5.3.1 Server Idempotence

We use sequence numbers to guarantee idempotence of server operations [40]. Each requester assigns a unique requester-local sequence number for a cache coherence event; all requests (i.e., cache coherence requests and unlock requests) in the cache coherence event contain the sequence number. Sequence numbers are allocated in increasing integer order. On suspecting a lost request via timeout, the requester retransmits it. We set the timeout value of cache coherence requests to 6 RTTs (round-trip times) and unlock requests to 3 RTTs.

Each home/cache agent maintains a *LAST-EXECUTED* table, which records the largest sequence number ever received from each requester. When a home/cache agent receives a cache coherence request, it compares the request's sequence number with the corresponding value in the *LAST-EXECUTED* table. If the request's sequence number is larger, the home/cache agent executes the request, updates the *LAST-EXECUTED* table, and responds. If the request's sequence number is lower, the request is ignored. If the two are the same, the home/cache agent sends an ACK without modifying any state.

[5] For space reasons, in this subsection (§5.3), we only present how to handle packet loss in the FLOWCC protocol. We use the same mechanism for the ownership migration.

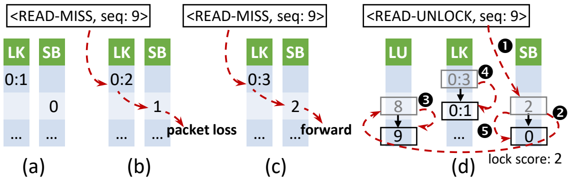


Figure 8: Idempotent Lock/Unlock Operations. The boxes on the top show packets with only the relevant details (*seq*: sequence number). LK is the *LOCK* array, and SB is the *SCOREBOARD* array, and LU is the *LAST-UNLOCK* array. Each 16-bit reader-writer lock in *LOCK* has the form x:y, where x is most significant bit marking the writer and y is the other 15 bits counting the number of readers. **(a)** Initialization state. **(b)** A requester issues a READ-MISS. The switch acquires the read lock by increasing the reader counter, and then increases the requester's lock score. Unfortunately, the READ-MISS is dropped after leaving the switch pipeline. **(c)** Upon timeout, the requester re-sends the READ-MISS with the same sequence number. After updating the *LOCK* and *SCOREBOARD* arrays, since the lock score is more than 0, the switch thinks the lock is acquired successfully. **(d)** The switch handles an unlock request.

5.3.2 Switch Idempotence

Since modifications to the *TAG*, *G-STATUS*, and *COPYSET* arrays in the switch are already idempotent, we only need to make lock and unlock operations idempotent. We introduce a *SCOREBOARD* array and a *LAST-UNLOCK* array to solve this problem, at the cost of slight on-chip memory usage.

The *SCOREBOARD* array is in the LCF pipeline and is placed before the check table to **make lock operations idempotent**. For each requester, it records a *lock score*, which refers to the number of successful lock acquisitions. When the switch processes a cache coherence request and manages to acquire the lock, the requester's lock score is incremented. The switch passes the request to the subsequent check table *only if the lock score is greater than 0*. For read lock operations, the lock score may be greater than 1 due to retransmitted requests; Figures 8(a)-(c) show an example of such a case.

The *LAST-UNLOCK* array is placed before the LCF pipeline to **make unlock operations idempotent**. For each requester, it records the largest sequence number of executed unlock requests, to avoid repeated execution of the same unlock request. When receiving an unlock request, the switch reads the requester's lock score (❶ in Figure 8(d)), and then resubmits the request with the lock score (❷). If the sequence number in the resubmitted request is larger than the value in the *LAST-UNLOCK*, the switch updates the *LAST-UNLOCK* array (❸) and executes the unlock operation (❹). For a read unlock operation, the switch subtracts lock score from the reader counter field of the reader-writer lock. Finally, the requester's value in the *SCOREBOARD* is reset (❺).

Using only one lock score for a requester disables asynchronous unlock requests, since we cannot allow requests in two cache coherence events to manipulate the same lock score concurrently. Thus, to enable asynchronous unlock requests, each requester is associated with two lock scores in *SCOREBOARD* array: one for requests with odd sequence numbers, the other for even ones. Before issuing an unlock request, a requester must ensure that it has received the ACK of the unlock request in the last cache coherence event.

5.4 Practical Issues

Scalability. We focus on rack-scale DSMs in this paper, following the growing tread towards rack-scale computers, which have the potential as building blocks for datacenters (e.g., Microsoft Rack-scale computers [9], Facebook Open-Rack [3], Intel RSD [7]). By packing many servers into the same rack, a rack-scale computer can provide extremely high network bandwidth and low communication latency, to efficiently support various data-intensive applications (e.g., parameter servers [48]). CONCORDIA abstracts a rack into one giant machine with massive cache-coherent shared memory, easing the programming on rack-scale computers.

Within a rack, CONCORDIA is capable of scaling well. In our FLOWCC protocol, the most difficult tasks to scale (i.e., concurrency control and multicast) are offloaded to the switch; during a cache coherence event, involved home/cache agents only process constant-time tasks (independent of cluster size). The switch can process several billion packets per second, which is enough to handle coherence traffic within a rack.

If CONCORDIA spans multiple racks, each ToR switch will execute the LCF pipeline for home nodes that reside in its rack. Yet, there are several challenges that we must address. ① It is impractical to use a bitmap to encode the *copyset* as we do now, considering the number of servers in large-scale clusters; we need to design a more compact *copyset* format like coarse vectors [31]. ② To avoid cache invalidation storms in a large scale, a weaker memory consistency level (e.g., acquire-release consistency [61]) may be needed. A full exploration is our future work.

Crash safety. We handle failures of different components.

- *Switch.* If a switch fails, operators can replace it with a backup switch [34], but lost global metadata needs to be restored. Here, we mark the set of cache blocks managed by the crashed switch as S. After all application threads abort their ongoing write/read operations, the system enters into a recovery process: ① Every cache agent gathers the local states (i.e., dirty bit) of cache blocks that are in both S and its cache, and then sends them to corresponding home agents. ② By analyzing replies, home agents reconstruct the global metadata of S; now, the ownership of S is transferred to home agents safely. ③ Finally, the shadow node clears its *SHADOW–TAG*. Leveraging the information stored in the shadow node may accelerate recovery, and we leave it for future work.
- *Memory nodes.* We rely on applications atop CONCORDIA to mask failures of memory nodes, instead of implementing an internal fault-tolerance mechanism. Take the three applications in our evaluation (§7) as examples: (*i*) In-memory key–value stores are typically used to cache frequently accessed data of back-end databases [52]. Hence, we recover lost data from back-end databases. (*ii*) Transaction processing systems

can record transaction logs to remote servers [36, 63], to tolerate failures of memory nodes. (*iii*) For a graph computation engine, the most common way to achieve fault-tolerance is checkpointing [29, 51].

When a memory node fails but does not release locks, conflicting requests from other nodes cannot proceed. We adopt a conservative "stop-the-world" mechanism to address this problem: once notified that there is a failed memory node, all application threads in CONCORDIA drain their ongoing write/read operations, then halt. Next, all the locks in CONCORDIA are released safely. After that, the system continues to run. Using more flexible mechanisms, e.g., leasing [30], will consume extra on-chip memory and complicate the design of the switch data plane; we leave it as our future work.

• *Shadow node*. When the serving shadow node crashes, we designate a new shadow node among backups with the help of ZooKeeper [32]. The new shadow node reconstructs the *SHADOW-TAG* by reading the *TAG* array of the switch.

6 Implementation

We implement a prototype of CONCORDIA in approximately 7600 lines of C++ and 1500 lines of P4 [17]. Like TreadMarks [15] and GAM [20], CONCORDIA is a user-space DSM system. Applications are linked to the CONCORDIA library and call the read/write interface. In each memory node, the cache agent and home agent run on two different background threads by default, but it is easy to scale up them.

Network. We use RoCE (RDMA over Converged Ethernet) to enable high-performance network communication. All the cache block data is sent via RDMA WRITE verbs over Reliable Connected (RC) mode to avoid data copying. Other packets (e.g., cache coherence requests) use RDMA Raw Packets [10]; we fill a UDP header and use a reserved source port for these packets. The switch executes FLOWCC for these packets, and sets the UDP destination port to the queue pair number (qpn) of the targeted queue pair. Each Raw Packet queue pair registers a steering rule to intercept the packets received by the NIC whose source port equals reserved port and destination port equals its qpn.

Cache. The cache in each memory node is organized into a bucket-based hash table. When looking up a cache block, we hash its *tag* into a bucket, which contains a certain number of entries (e.g., 8). Each entry records a cache block's local metadata, including *dirty bit*, *tag*, *timestamp* and a pointer to the cache block data. We record the current time into the *timestamp* field when accessing a cache block, and employ an LRU policy for cache eviction. By default, the cache block size is 4KB, which achieves a good balance between network bandwidth and latency in the high-speed RDMA environment.

Switch data plane. The switch data plane is written in P4 and is compiled to Barefoot Tofino ASIC [2]. The *COPYSET* array is comprised of 32-bit slots, since our ToR switch owns 32 ports. We use one ingress pipeline to process the cache coherence traffic by realizing our LCF pipeline. Since the memory resources in a single stage of the ingress pipeline are limited, we scatter the values of *TAG*, *LOCK*, *G-STATUS*, and *COPYSET* arrays across 10 stages, forming set-associative structures. Thus, a cache block has multiple candidate locations in different stages: if the hash value of its tag is `k`, its tag and global metadata can be stored in the `k`th items of arrays in any stage. Our current implementation can manage 375K cache blocks (i.e., about 1.5GB cache data) in the switch, consuming about 6MB on-chip memory.

Synchronization primitive. We expose per-cache-block reader-writer locks in the FLOWCC protocol as a synchronization primitive (i.e., rwlock) to applications.

Global memory allocation. We use a two-level approach to allocate global memory for applications [60]. Application threads select a memory node and send allocation requests to its home agent. The home agent returns a huge free chunk (i.e., 32MB). Then application threads perform allocation locally in a fine-grained way, to reduce remote access.

7 Evaluation

This section presents the performance evaluation of CONCORDIA with a set of micro benchmarks and three applications. All experiments are conducted on a cluster of 8 machines, each with two 12-core Intel Xeon E5-2650 v4 2.20GHz CPUs, 128GB DRAM, and one 100Gbps Mellanox ConnectX-5 network adapter. A 3.3Tbps Barefoot Tofino switch (32 ports) connects all of the machines. All machines run the CentOS 7.4.1708 distribution and the 3.10.0 Linux kernel.

7.1 Systems in Comparison

We compare CONCORDIA with two state-of-the-art DSMs:

Grappa. Grappa [6, 51] is a DSM *without* a cache for data-intensive applications. Instead of fetching data to computation, Grappa ships computation to data via delegate operations. Message transmission in Grappa is done purely in user-mode using MPI, which in turn uses RDMA verbs.

GAM. GAM [5, 20] is a recent DSM that implements a directory-based cache coherence protocol over RDMA. In addition to application threads, GAM uses two background threads to handle cache coherence events by default. We disable GAM's logging scheme for a fair comparison.

7.2 Micro Benchmarks

In micro benchmarks, we launch a four-thread process in each memory node to generate mixed read-write workloads. Each operation in workloads accesses an 8-byte object in the global memory, which is the same as in GAM [20]. The working set of these micro benchmarks is 8GB, and the cache size is 1GB. The access pattern of the workloads is controlled via three parameters: *read ratio*, *data locality*, and *sharing ratio*. The *read ratio* is the percentage of read operations. The *data locality* is the probability of an operation accessing the same cache block as the previous one. The *sharing ratio* is the percentage of operations that access data shared across all nodes, and the shared data is about 250MB. We run all

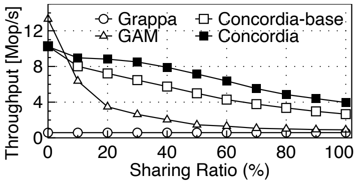


Figure 9: Performance Impact of Sharing Ratio.

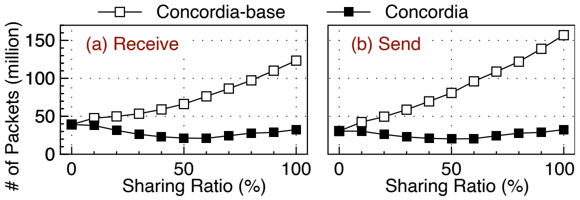


Figure 10: Communication Reduction from LCF Pipeline. The figure shows the number of packets received/sent by home agents.

the micro benchmarks on 8 nodes. By default, the read ratio is 50% and the data locality is 0%. The baseline version of CONCORDIA (i.e., CONCORDIA-base), in which servers manage ownership of all cache blocks, is also evaluated.

7.2.1 Sharing Ratio

Figure 9 presents the results with various sharing ratio. Because the operations are very small (i.e., 8 bytes read/write), which induces too many remote delegation operations and worker scheduling events in Grappa, Grappa's throughput is far lower than other systems.

When the sharing ratio is 0%, i.e., no cache coherence traffic, GAM has higher throughput than CONCORDIA and CONCORDIA-base. This is because the cache in GAM is a fully associative mapping structure, which causes fewer cache block evictions than the set-associative cache of CONCORDIA.

As the sharing ratio becomes higher, the introduced cache coherence traffic becomes severe, which causes performance degradation of all the DSMs. The throughput of GAM is reduced by 17× when the sharing ratio increases from 0% to 100%. However, CONCORDIA-base's throughput is only reduced by 3.8× even with considerable coherence traffic (i.e., 100% sharing ratio), and it outperforms GAM by 1.25× to 3.5× when shared data access exists. The reason is that CONCORDIA-base offloads home node tasks such as invalidation aggregation to the requesters, thus reducing the load on home nodes. CONCORDIA outperforms CONCORDIA-base by 1.3× to 1.48× when sharing ratio is larger than 20%. This is mainly because our in-network protocol takes home nodes off the critical path of cache coherence as well as reduces network hops and coordination, leveraging the extremely high processing power of the switch.

To quantitatively understand the effect of the LCF pipeline, we collect the number of packets sent and received by all home agents in CONCORDIA and CONCORDIA-base, as shown in Figure 10. When the sharing ratio grows, the number of packets sent and received by CONCORDIA-base's home agents increases dramatically due to increased coherence traffic. However, owing to the help of the LCF pipeline, home agents in CONCORDIA send and receive a steady number of packets, regardless of the sharing ratio. Such a result indicates that the LCF pipeline effectively mitigates load for home agents and reduces coordination between servers, which ensures CONCORDIA's performance gain over CONCORDIA-base. Of note, though CONCORDIA reduces transferred packets significantly (up to 4.8×), it improves throughput by up to 1.48× (Figure 9). This is because part of the accesses are served by the local cache (39% in case of 100% sharing ratio) and do not incur coherence traffic; these local accesses cannot benefit from the LCF pipeline.

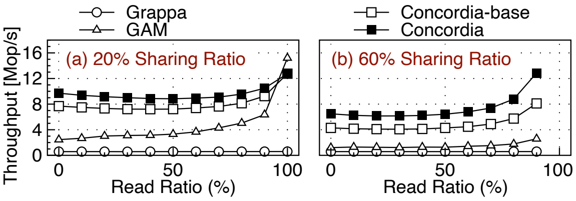


Figure 11: Performance Impact of Read Ratio.

7.2.2 Read Ratio

This experiment studies how the read ratio affects the performance. Figure 11 shows the results. When the read ratio is lower than 50%, the throughput of these four DSMs is stable regardless of read ratio, since the performance of systems except Grappa is bottlenecked by coherence traffic. CONCORDIA outperforms CONCORDIA-base by 1.22× and GAM by 3.5× to 3.9× in case of light data sharing (i.e., 20% sharing ratio) and outperforms CONCORDIA-base by 1.48× to 1.5× and GAM by 4.5× to 5.1× in case of heavy data sharing (i.e., 60% sharing ratio). CONCORDIA performs better because the FLOWCC protocol accelerates coherence processing.

The throughput of GAM, CONCORDIA-base, and CONCORDIA grows as the read ratio increases from 50% to 100%, since the coherence traffic becomes light. More specifically, ① GAM outperforms CONCORDIA in case of 100% read ratio and 20% sharing ratio. This is because no coherence traffic exists and GAM triggers fewer cache eviction events than CONCORDIA. ② With 100% read ratio and 60% sharing ratio, the throughputs of GAM and CONCORDIA/CONCORDIA-base are 16Mops and 60Mops, respectively, which we do not plot in the figure to avoid obscuring other results. This is because (*i*) there is almost no cache eviction, and (*ii*) CONCORDIA's cache structure is optimized for fast access, but GAM needs to acquire/release four locks and maintain LRU lists when accessing a cache block, which severely stalls CPU pipeline even without data race.

7.2.3 Data Locality

Figure 12 investigates how data locality affects the throughput of these systems. With higher data locality, the performance of CONCORDIA and GAM improves substantially as a result of the cache. On the contrary, since Grappa does not use a cache, it can not benefit from the locality. We do not plot results when the locality is 100% because the corresponding

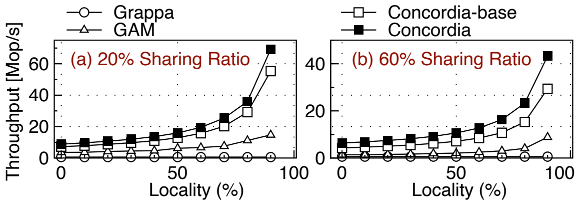


Figure 12: Performance Impact of Data Locality.

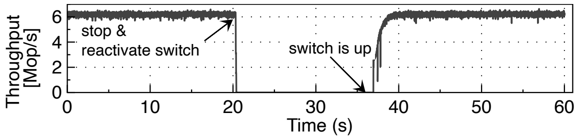


Figure 13: Switch Failure Handling.

values are too high: 23Mops for GAM and 133Mops for Concordia and Concordia-base, which would obscure other results. This is because Concordia's cache structure is much faster, as stated in §7.2.2.

7.2.4 Switch Failure Handling

Here, we evaluate how Concordia handles a switch failure. We set the sharing ratio to 60%. Figure 13 shows the total throughput over time. At time 20 s, we stop the switch by killing its daemon process, and then reactivate it; since the switch cannot route packets, the throughput immediately drops to 0. After initialization of the switch ASIC and drivers (about 16 seconds), the switch is up and can route packets. Then, Concordia recovers the global metadata of cache blocks managed by the previous switch instance, consuming about 1.9 seconds; after this step, Concordia can continue to run its workloads. Finally, Concordia migrates the ownership of shared cache blocks to the switch (1.5 seconds), and the throughput reaches a stable peak.

7.3 Distributed Key-Value Store

We build a distributed key-value store atop Concordia, which is implemented by a distributed array of buckets across nodes. A key is hashed to a bucket, and then PUT/GET operations are translated into a series of DSM calls (i.e., write, read, lock and unlock). GAM has a similar version of key-value store implementation. For Grappa, each thread maintains part of the key-value store and handles delegation requests.

We run skewed workloads using a non-uniform key popularity that follows a Zipf distribution of skewness 0.99, which is the same as YCSB's [22]. The keyspace size is 64 million, and we use fixed 8-byte keys and 128-byte values. 95% GET workloads are read-intensive, and 50% GET workloads are write-intensive. Each node launches a four-thread process to generate workloads. The cache size is 2GB. Figure 14 shows the results with varied node counts.

For read-intensive workloads, when there are more than 2 nodes, Concordia outperforms Grappa by 3.9× to 5× and GAM by 1.2× to 2.5×. Grappa has the lowest performance among the three systems, because each PUT/GET operation needs remote delegation and the nodes serving the most popular objects suffer from hot spots in the presence of skew. The cache absorbs many GET operations, and thus the throughputs of GAM and Concordia are higher than that of Grappa. The workload with 5% PUT operations causes a small amount of cache coherence traffic in Concordia and GAM. However, GAM is more vulnerable to coherence traffic than Concordia, due to its traditional directory-based protocol design, and its throughput plateaus or even decreases somewhat when the node count becomes larger. In contrast, Concordia can benefit from the cache, while minimizing coherence overhead with the help of FlowCC protocol.

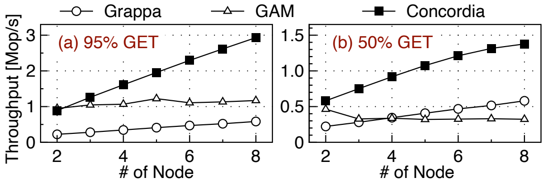


Figure 14: Throughput of Key-Value Store.

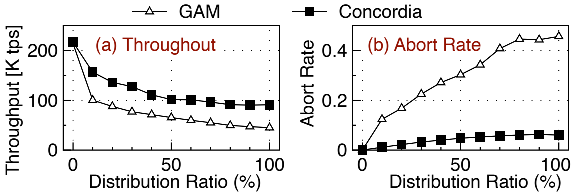


Figure 15: Performance of TPC-C Benchmark.

For write-intensive workloads, Concordia outperforms GAM by 1.2× to 4.2× and Grappa by 2.3× to 2.6×. GAM's total throughput drops and is lower than Grappa's when adding more nodes because of heavy coherence overhead. The ownership migration in Concordia lets the switch manage coherence traffic of the hottest cache blocks in skewed workloads, which guarantees Concordia's scalability.

7.4 Transaction Processing

We port the transaction engine of GAM into Concordia; this engine implements two-phase locking for concurrency control and non-waiting scheme for deadlock prevention. In this experiment, we use TPC-C [11] to compare the performance of Concordia against GAM. All tables and indices are uniformly distributed in the global memory, following the growing trend towards shared-everything architectures [64, 66]. We launch 4 threads in each node, and each thread holds a TPC-C warehouse (i.e., 32 warehouses in total). Each thread accesses other warehouses with a probability called the *distribution ratio*. The cache size is 2GB.

Figure 15(a) shows the throughput of the TPC-C benchmark. Both systems have almost the same throughput when there is no data sharing. However, Concordia outperforms GAM by 1.55× to 2× when different threads access the same warehouses. This is because Concordia's FlowCC protocol is faster than GAM's, causing lower transaction execution time and further reducing transaction aborts due to contention. Figure 15(b) plots the transaction abort ratio. The abort ratio of the two systems increases when the distribution ratio gets

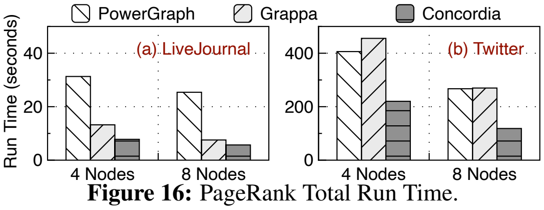


Figure 16: PageRank Total Run Time.

higher, but CONCORDIA has a much lower abort ratio.

7.5 Distributed Graph Computing

We build a distributed graph processing engine on CONCORDIA. Similar to PowerGraph [29], we store graphs using a vertex-centric representation with random graph partitioning. For a graph algorithm, each thread executes the gather, apply and scatter phases on a set of vertices.

We compare CONCORDIA's graph engine with Grappa and PowerGraph using two real datasets: LiveJournal (4M vertices, 34.7M directed edges) [8] and the latest Twitter graph (61M vertices, 14B directed edges) [12, 39] . We run PageRank [54] with CONCORDIA, PowerGraph, and Grappa under 4 and 8 node settings, and each node launches 4 threads. We use PowerGraph's default threshold criteria, which results in the same number of iterations for all systems.

Figure 16 shows the total PageRank run time. CONCORDIA outperforms Grappa by 1.3× to 2.3× and PowerGraph by 1.8× to 4.4×. This is because, CONCORDIA's network stack takes full advantage of RDMA performance, using WRITE verbs without any data copying. In addition, CONCORDIA's cache mechanism minimizes the amount of data transferred across the network.

8 Related Work

Distributed Shared Memory. In the past few decades, many DSMs and cache coherence approaches have been proposed [15, 16, 21, 44, 58, 62]. IVY [44] provides sequential consistency at the cost of frequent cache invalidations. Other systems [15,21,58] relax the consistency model or avoid false sharing to reduce communication overheads. CONCORDIA provides strong consistency to ease the programming, while minimizing coherence overheads.

As recent RDMA technology makes the latency and throughput of the network approach that of memory, new DSMs have emerged [20, 25, 47, 51, 61]. FaRM [25, 26, 59] offers general distributed transactions to global shared memory. Octopus [47] redesigns a distributed file system over a shared persistent memory pool. Hotpot [61] leverages non-volatile memory to incorporate both distributed memory and distributed storage. Different from the above systems, CONCORDIA focuses on cache coherence and accelerates it by exploiting new programmable network hardware.

In-Network Computation. Emerging network hardware like programmable switches poses new opportunities for in-network computation [49, 55]. NetCache [34] and DistCache [46] propose in-switch caches for load balancing. NetChain [33] designs a replicated, in-switch key-value store for distributed coordination. IncBricks [45] supports caching in the network using a programmable network middlebox. These in-network caching or key-value store systems need to keep data in the network hardware and servers coherent. They adopt similar *server-based* mechanisms to solve this problem. For example, in IncBricks, servers record the sharers list and issue invalidation commands to network accelerators when clients send SET or DELETE requests. In contrast, CONCORDIA exploits the programmable network to coordinate coherence among the cache of servers and only stores the cache's metadata in the switch.

The most similar work to ours is Pegasus [43]. Pegasus replicates the most popular objects to distribute load and leverages switches to track servers that store replicated objects. When a client issues a read request, the switch routes the request to a replica by a load-aware scheduling policy. When a client issues a write request, the switch resets the replica set to include only one server by a version-based mechanism. Pegasus's protocol is specialized and simplified for *client-server model* systems. In contrast, our FLOWCC protocol is designed for the general cache coherence problem in *symmetric model* systems (i.e., DSMs), with complex state transitions and concurrency control. In addition, CONCORDIA is a DSM but Pegasus is an object store.

9 Conclusion

We present CONCORDIA, a rack-scale DSM with in-network cache coherence; it divides coherence responsibility between switches and servers to reduce coherence overhead within the functionality and resource limit of switches. Specifically, CONCORDIA incorporates (*i*) a protocol that leverages switches to serialize and multicast requests, (*ii*) a mechanism that moves the ownership of cache blocks between switches and servers dynamically, and (*iii*) a method that makes operations in both servers and switches idempotent. CONCORDIA significantly outperforms existing solutions.

We believe that our in-network coherence protocol can also benefit other systems such as distributed file systems and hardware memory disaggregation. As storage hardware becomes extremely fast (e.g., non-volatile memory), new distributed file systems need microsecond-scale coherence for metadata caching, which can be provided by our in-network coherence protocol. Our in-network coherence protocol can also enable compute node caching for shared data in hardware disaggregated memory architectures; such caching is indispensable for reducing network accesses.

10 Acknowledgements

We sincerely thank our shepherd Kimberly Keeton for helping us improve the paper. We also thank the anonymous reviewers for their feedback. This work is supported by the National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 62022051, 61832011, 61772300, 61877035), and Huawei (Grant No. YBN2019125112).

References

[1] Introducing Amazon EC2 C5n Instances Featuring 100 Gbps of Network Bandwidth. "https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-c5n-instances/", 2018.

[2] Barefoot Tofino. "https://barefootnetworks.com/products/brief-tofino/", 2020.

[3] Facebook OpenRack. "https://engineering.fb.com/data-center-engineering/open-rack/", 2020.

[4] FaRM Project. "https://www.microsoft.com/en-us/research/project/farm/", 2020.

[5] GAM Repository. "https://github.com/ooibc88/gam", 2020.

[6] Grappa Repository. "https://github.com/uwsampa/grappa", 2020.

[7] Intel RSD. "https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html/", 2020.

[8] LiveJournal Dataset. "http://snap.stanford.edu/data/soc-LiveJournal1.html", 2020.

[9] Microsoft Rack-Scale Computers. "https://www.microsoft.com/en-us/research/project/rack-scale-computing/", 2020.

[10] RDMA Raw Packet. "https://community.mellanox.com/s/article/raw-ethernet-programming--basic-introduction---code-example", 2020.

[11] TPC-C. "http://www.tpc.org/tpcc/", 2020.

[12] Twitter Dataset. "http://an.kaist.ac.kr/traces/WWW2010.html", 2020.

[13] Dennis Abts, Steve Scott, and David J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, page 11.2, USA, 2003. IEEE Computer Society.

[14] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote Memory in the Age of Fast Networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 121–127, New York, NY, USA, 2017. ACM.

[15] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, February 1996.

[16] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. Technical report, Pittsburgh, PA, USA, 1993.

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[18] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[19] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. A1: A Distributed In-Memory Graph Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 329–344, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[21] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 152–164, New York, NY, USA, 1991. ACM.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[23] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, New York, NY, USA, 2015. ACM.

[24] Xavier Défago, André Schiper, and Péter Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[25] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association.

[26] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. ACM.

[27] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 21:1–21:15, New York, NY, USA, 2018. ACM.

[28] Kourosh Gharachorloo. The Plight of Software Distributed Shared Memory. In *Invited talk at 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*. Citeseer, 1999.

[29] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[30] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, page 202–210, New York, NY, USA, 1989. Association for Computing Machinery.

[31] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-based Cache Coherence Schemes. In *Scalable shared memory multiprocessors*, pages 167–192. Springer, 1992.

[32] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[33] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-RTT Coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 35–49, Berkeley, CA, USA, 2018. USENIX Association.

[34] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[35] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark Tuttle, and Yuan Yu. Checking Cache-Coherence Protocols with TLA+. *Form. Methods Syst. Des.*, 22(2):125–131, March 2003.

[36] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, November 2016. USENIX Association.

[37] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. Generic External Memory for Switch Data Planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 1–7. ACM, 2018.

[38] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs First-class Datacenter Citizens. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 863–879, Berkeley, CA, USA, 2019. USENIX Association.

[39] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[40] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 71–86, New York, NY, USA, 2015. ACM.

[41] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 104–120, New York, NY, USA, 2017. ACM.

[42] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network

Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 467–483, Berkeley, CA, USA, 2016. USENIX Association.

[43] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 387–406. USENIX Association, November 2020.

[44] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '86, pages 229–239, New York, NY, USA, 1986. ACM.

[45] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 795–809, New York, NY, USA, 2017. ACM.

[46] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable Load Balancing for Large-scale Storage Systems with Distributed Caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, pages 143–157, Berkeley, CA, USA, 2019. USENIX Association.

[47] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An RDMA-enabled Distributed Persistent Memory File System. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 773–785, Berkeley, CA, USA, 2017. USENIX Association.

[48] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter Hub: A Rack-Scale Parameter Server for Distributed Deep Neural Network Training. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 41–54, New York, NY, USA, 2018. Association for Computing Machinery.

[49] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. Thoughts on Load Distribution and the Role of Programmable Switches. *SIGCOMM Comput. Commun. Rev.*, 49(1):18–23, February 2019.

[50] Vijay Nagarajan, Daniel Sorin, Mark Hill, and David Wood. A Primer on Memory Consistency and Cache Coherence, Second Edition. *Synthesis Lectures on Computer Architecture*, 15:1–294, 02 2020.

[51] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 291–305, Berkeley, CA, USA, 2015. USENIX Association.

[52] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, page 385–398, USA, 2013. USENIX Association.

[53] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, August 1991.

[54] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[55] Dan R. K. Ports and Jacob Nelson. When Should The Network Be The Computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 209–215, New York, NY, USA, 2019. ACM.

[56] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 150–156, New York, NY, USA, 2017. ACM.

[57] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[58] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 297–306, New York, NY, USA, 1994. ACM.

[59] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast General Distributed Transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 433–448, New York, NY, USA, 2019. ACM.

[60] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 69–87, Berkeley, CA, USA, 2018. USENIX Association.

[61] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 323–337, New York, NY, USA, 2017. ACM.

[62] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *SIGARCH Comput. Archit. News*, 20(1):5–44, March 1992.

[63] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[64] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 449–462, Santa Clara, CA, February 2020. USENIX Association.

[65] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 126–138, New York, NY, USA, 2020. Association for Computing Machinery.

[66] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

[67] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection. *Proc. VLDB Endow.*, 13(3):376–389, November 2019.

[68] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240. USENIX Association, November 2020.

*e*MRC: Efficient Miss Ratio Approximation for Multi-Tier Caching

Zhang Liu
University of Colorado Boulder

Hee Won Lee*
Samsung Electronics

Yu Xiang
AT&T Labs Research

Dirk Grunwald
University of Colorado Boulder

Sangtae Ha
University of Colorado Boulder

Abstract

Many storage cache allocation methods use the *miss ratio curve* (MRC) to improve cache efficiency. However, they have focused only on single-tier cache architectures and require the whole MRC as input for cache management, while modern datacenters embrace hierarchical caching architectures to maximize resource utilization. Generating the MRC for multi-tier caches – we call it the *miss ratio function* – is far more challenging due to different eviction policies and capacities in each cache tier. We introduce *e*MRC, a multi-dimensional miss ratio approximation technique, to enable efficient MRC generation for multi-tier caching. Our approach uses a novel multi-dimensional performance cliff removal method and convex hull approximation technique to efficiently generate a multi-dimensional MRC without cliffs using a small number of sampling points. To demonstrate the benefits of *e*MRC, we designed ORCA, a multi-tier cache management framework that orchestrates caches residing in different hierarchies through *e*MRC and provides efficient multi-tier cache configurations to cloud tenants with diverse service level objectives. We evaluate the performance of our *e*MRC approximation technique and ORCA with real-world datacenter traces.

1 Introduction

Caching is often provisioned on multiple tiers in cloud storage systems. When client applications running in virtual machines (VMs) generate block IOs to remote storage media, the requests pass through a series of intermediate layers, such as user-space libraries, hypervisors, and actual storage nodes. Each layer presents a caching opportunity, and thus most cloud providers adopt multi-tier caching architectures to maximize the utilization of scattered resources in the system [11].

Multi-tier caching necessitates an effective, low overhead cache management scheme as arbitrary cache configurations for multiple tiers may not benefit tenants due to their side effects such as double caching effects [27]. Configuring caches for tenants with diverse service level objectives (SLOs) requires efficient and accurate cache performance analysis for *each* tier of the cache.

It is well known that effective cache management requires a good understanding of IO workload characteristics. Without understanding the workload, systems usually rely on trial-and-error tuning methods that can be very inefficient. The *miss ratio curve* (MRC) is a useful tool to capture workload characteristics and tune system behavior. The MRC represents the relationship between cache size and the corresponding cache miss ratio. Assuming workloads are relatively stable over time, the MRC derived from observed IO traces is known to work effectively for single-tier caches [13].

The general workflow of utilizing miss ratio information for cache management is as follows: 1) the IO streams of tenants are analyzed, and a miss ratio function is generated to represent the miss ratio for any given cache configurations for each tenant; 2) based on tenants' SLOs, a cache management framework allocates the optimal cache size for each tenant. In this workflow, the performance primarily depends upon how quickly it estimates the miss ratio of a particular sized cache for each tenant, given that the cache management framework needs to handle multiple tenants, each with a different IO pattern in a datacenter.

Evaluating the miss ratios for all possible cache sizes is a very time-consuming task. SHARDS [25] and Miniature Simulation [24] allow rapid MRC construction with reduced overhead, and other efficient techniques have also been proposed [7, 10, 26].

These previous techniques are either specific to single-tier caches or are inefficient for multi-tier caches. As shown in Figure 1, there are *performance cliffs* on the miss ratio surface where miss ratios change dramatically with small changes in cache size. The miss ratio surface is a multi-dimensional function representing the relationship between cache size in each tier of the cache hierarchy and the corresponding cache miss ratios. While evaluating cache configurations using multi-tier

[*] Hee Won Lee conducted this project when he was at AT&T Labs Research.

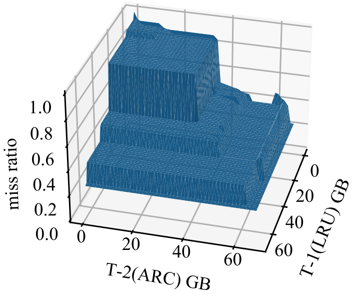


Figure 1: Miss ratio surface of MSR `web_2` trace.

cache simulators such as *PyMimircache* [8, 30] is possible, and MRC cliff removal techniques [4, 6] are available for single-tier caching, to the best of our knowledge, there are no known algorithms that can remove performance cliffs in miss ratio functions for multi-tier caches, or that can generate continuous miss ratio functions efficiently.

In this paper, we present our *e*MRC approach to achieving the *efficient, rapid* generation of multi-dimensional miss ratio functions for multi-tier caching. *e*MRC is enabled by our *convex hull algorithm* and *cache partitioning algorithm*. The convex hull algorithm efficiently divides the whole multi-dimensional space into multiple regions without full knowledge of the multi-dimensional MRC. This convex hull algorithm only requires a small number of sampling points that significantly reduces the computation time. The cache partitioning algorithm then removes performance cliffs for multi-tier caching within any region bounded by data points with known miss ratio values; the resulting miss ratio function is always equal to or better than the original miss ratio function without *e*MRC's cache partitioning.

We also developed the ORCA cache orchestration framework for multi-tenant, multi-tier caching that leverages *e*MRC. We evaluate both *e*MRC and ORCA using real-world IO traces released by Microsoft [1].

Our key contributions can be summarized as follows:

- We are the first to provide an algorithm, *e*MRC, that removes performance cliffs in multi-dimensional miss ratio functions for multi-tier caching.
- For *e*MRC, we develop a technique called *Convex Hull Approximation*. In terms of the number of sampling points required, it speeds up MRC generation by 14 times for two-tier caching and 4,527 times for four-tier caching.
- ORCA uses *e*MRC to efficiently provide effective cache configurations for tenants with diverse SLOs by selecting optimal cache sizes and replacement policies.
- We evaluate our *e*MRC approximation method with real datacenter traces and validate that ORCA provides effective multi-tier cache configurations and boosts the performance for various types of mixed workloads.

2 Background

In this section, we will review three concepts upon which our *e*MRC approximation is built.

2.1 Sampling and Statistical Similarity

Statistical similarity means a smaller sampled IO trace can be used to estimate the miss ratio of the original IO trace. Kessler et al. [12] defined the "*10% sampling goal*" for *statistical similarity*: "A method meets the 10% sampling goal if, at least 90% of the time, it estimates the trace's true misses per instruction with $\leq$ 10% relative error using $\leq$ 10% of the trace", and showed that constant-bits sampling satisfies such a goal. Constant-bits sampling means selecting the IO entries that have the same value in some address bits.

Spatial sampling is a recently proposed technique to sample IO traces with *statistical similarity*. It means taking the hash values of IO addresses A and then using modulus P and a threshold T on the hash value to determine what fraction of IO operations to sample, $hash(A)\, mod\, P < T$. The resulting sampling rate is $R = T/P$.

Statistical similarity has been used in various MRC based studies, e.g., accelerating trace-driven simulations [24, 25] and altering cache behaviors by using shadow partitions [4].

2.2 Talus Cache Partitioning

Talus [4] is a recently proposed cache partitioning algorithm that can remove MRC performance cliffs for single-tier caching. Talus utilizes *statistical similarity* of spatial sampled IO streams. Assuming the original miss ratio is $m(x)$ for cache size x, passing a fraction[1] ρ of an original IO stream into a proportionally smaller cache partition of size $x' = \rho x$ will result in a statistically similar miss ratio:

$$m'(x') = m(\frac{x'}{\rho}), \text{ where } 0 \leq \rho \leq 1 \qquad (1)$$

The Talus algorithm divides a single cache into two partitions so that the cache has a miss ratio that interpolates between two points on the MRC of an original unpartitioned cache. To demonstrate the Talus algorithm, we use an MRC example shown in Figure 2. A performance cliff exists between cache sizes α and β, which are two convex hull points. If their cache miss ratios are $m(\alpha)$ and $m(\beta)$, respectively, Talus partitioning technique can provide cache miss ratio $m(x)$ whose value is between $m(\alpha)$ and $m(\beta)$. The small-dotted MRC is the miss ratio after applying the Talus algorithm. The Talus MRC curve is convex and lower than the MRC of the unpartitioned cache.

Now let us assume we want to configure cache partitions so that the overall miss ratio is lowered from 66% to 53% with a cache size of $x = 0.99$ GB that falls between $\alpha = 0.23$ GB and

[1] The fraction means the sampling rate of spatial sampling.

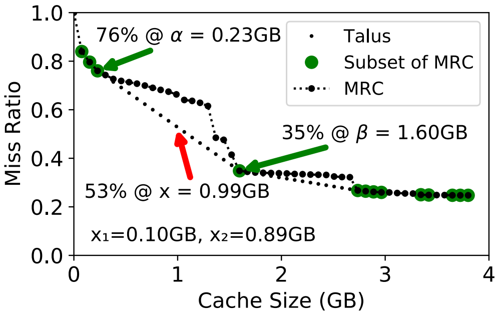


Figure 2: Talus miss ratio cliff removal.

$\beta = 1.60$ GB. We can determine the miss ratio using the Talus equation: $m_{\text{talus}}(x) = \frac{\beta-x}{\beta-\alpha}m(\alpha) + \frac{x-\alpha}{\beta-\alpha}m(\beta)$. Hence, for the example of Figure 2, we can obtain:

$$\begin{aligned} m_{\text{talus}}(0.99) &= \frac{1.60-0.99}{1.60-0.23}m(0.23) + \frac{0.99-0.23}{1.60-0.23}m(1.60) \\ &= 0.45 \times 0.76 + 0.55 \times 0.35 \\ &= 0.53. \end{aligned}$$

To achieve this miss ratio, Talus method allocates two cache partitions of size x_1 and x_2 such that $x = x_1 + x_2$. A fraction ρ of the IOs will pass through the first cache partition of size $x_1 = \rho\alpha$ where $\rho = \frac{\beta-x}{\beta-\alpha}$, and the remaining fraction $(1-\rho)$ of the IOs will pass through the second cache partition of size $x_2 = x - x_1$.

The resulting miss ratio for the first partition is $m_1(x_1)$, which equals $m(x_1/\rho)$ by Equation 1. And the miss ratio for the second partition, $m_2(x_2)$, equals $m(\frac{x-x_1}{1-\rho})$.

The resulting Talus miss ratio is:

$$\begin{aligned} m_{\text{talus}}(x) &= \rho m_1(x_1) + (1-\rho)m_2(x_2) \\ &= \rho m(\frac{x_1}{\rho}) + (1-\rho)m(\frac{x-x_1}{1-\rho}) \\ &= \frac{\beta-x}{\beta-\alpha}m(\alpha) + \frac{x-\alpha}{\beta-\alpha}m(\beta) \end{aligned}$$

In sum, we want to have the *Talus cache* behave like a mix of cache sizes α and β. The sample point $x = 0.99$ GB results in $\rho = \frac{\beta-x}{\beta-\alpha} = (1.60-0.99)/(1.60-0.23) = 0.45$. Thus, 45% of the IOs will use a cache partition of $x_1 = \rho\alpha = 0.45 \times 0.23 = 0.10$ GB and they will have a miss ratio of $m(x_1/\rho) = m(\alpha) = 0.76$. The remaining 55% of the references will use a cache size of $x - x_1 = 0.99 - 0.10 = 0.89$ GB with a miss ratio of $m(\frac{x-x_1}{1-\rho}) = m(\beta) = 0.35$. Across all the IOs, the miss ratio will be $0.45 \times 0.76 + 0.55 \times 0.35 = 0.53$.

As this process is repeated for different points x between α and β, the resulting miss ratio curve will be a linear interpolation between the miss ratios at α and β.

2.3 Rapid MRC Generation

Existing research has mainly focused on using a subset of the original trace to accelerate MRC generation. Some of them focused on stack-based eviction policies, while others can apply to all eviction policies.

Stack-based cache eviction policies have the *inclusion property*, meaning that for the same input IO, the cache of a larger size always contains all cached items in the cache of a smaller size. Simple cache eviction policies such as LRU and LFU are stack-based policies, but more complex eviction policies such as ARC [17] or MQ [32] are not stack-based. SHARDS [25] is based on the stack distance algorithm of Mattson *et al.* [16], which generates the MRC with a single pass of the workload trace. With the help of spatial downsampling, SHARDS can significantly reduce the computation time and memory footprint for long traces while generating relatively accurate MRCs.

Miniature-Simulation [24] is another recent advance in MRC generation for both stack and non-stack based eviction policies. It shows that heavy spatial downsampling can be used to generate a relatively accurate MRC by running a separate scaled-down simulation for each cache size.

3 *e*MRC: Miss Ratio Approximation for Multi-Tier Caching

We will begin with an illustrative example to intuitively show how *e*MRC works. After that, we will demonstrate that spatial sampling also works for more than one cache tier in Section 3.2. We will show how *e*MRC can remove multi-dimensional performance cliffs in a cliff region bounded by data points with known miss ratios in Section 3.3. We explain how to partition the whole multi-dimensional MRC into multiple regions efficiently to apply *e*MRC for the entire space in Section 3.4. To simplify explanation, we first address two-tier caching and then generalize our algorithm for three or more tier caching in Section 3.5.

3.1 Illustrative Example for *e*MRC

In single-tier caching, Talus achieves a convex MRC by partitioning the cache into two partitions and letting the miss ratio of each partition be related to one of the boundary point miss ratios. In two-tier caching, as the example in Figure 3 illustrates, there are four boundary points. *e*MRC partitioned the last tier cache (2nd tier) into four partitions, with each partition related to one of the boundary point miss ratios.

In the particular example in Figure 3, there is a cliff region in the MRC of a workload using two-tier caching. The four boundary point miss ratios are: $M(2,2) = 0.9$, $M(2,6) = 0.5$, $M(6,2) = 0.5$ and $M(6,6) = 0.5$. If we use the two tiers of caches without partitioning, the miss ratio for 3GB of tier-1 cache and 3GB of tier-2 cache is $M(3,3) = 0.9$.

*e*MRC ensures that if we partition the tier-1 cache into two partitions of [1.5GB, 1.5GB] and the tier-2 cache into four partitions of [1.125GB, 1.125GB, 0.375GB, 0.375GB], and then divide the IOs into the different partitions using

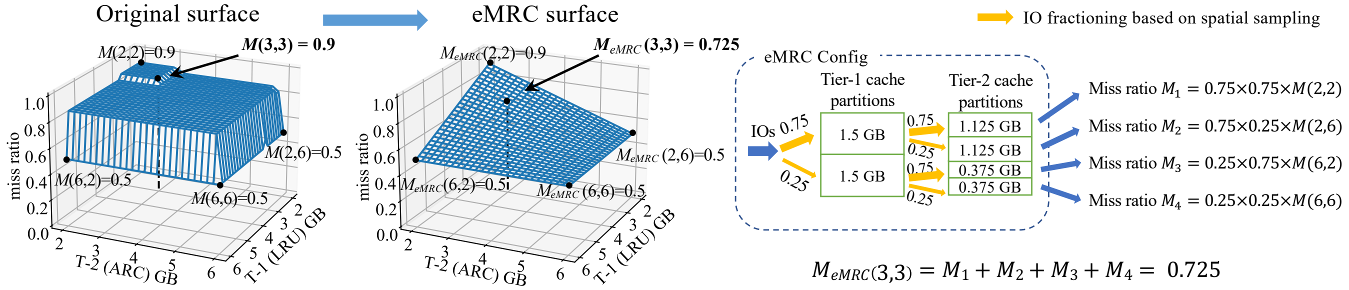


Figure 3: Example of how *e*MRC removes performance cliffs.

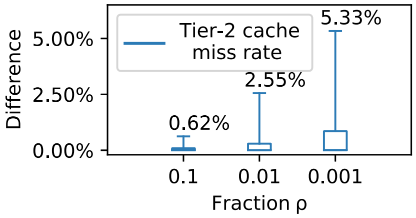


Figure 4: Relative errors in approximating cache miss ratios by sampling at rate ρ across all workloads. The error bars represent the 10th & 90th percentile values.

spatial sampling based hash functions, we have the following relation:

$$\begin{bmatrix} M_1 \\ M_2 \\ M_3 \\ M_4 \end{bmatrix} = \begin{bmatrix} 0.75\times0.75 & 0 & 0 & 0 \\ 0 & 0.75\times0.25 & 0 & 0 \\ 0 & 0 & 0.25\times0.75 & 0 \\ 0 & 0 & 0 & 0.25\times0.25 \end{bmatrix} \begin{bmatrix} M(2,2) \\ M(2,6) \\ M(6,2) \\ M(6,6) \end{bmatrix}$$

This means each of the miss ratios after the four tier-2 cache partitions is only related to one of the boundary point miss ratios. For example, $M_1 = 0.75\times0.75\times M(2,2)$, meaning no matter how the miss ratios change at the boundary points, M_1 is only related to the miss ratio at $M(2,2)$ on the original surface.

The resulting miss ratio using *e*MRC for 3GB tier-1 cache and 3GB tier-2 cache is now $M_{eMRC}(3,3) = 0.725$ compared to the original miss ratio at $M(3,3) = 0.9$. If we apply *e*MRC to all the cache configurations in the bounded region, we can obtain a convex *e*MRC surface as shown in Figure 3.

In Section 3.3, we show how *e*MRC determines the partition parameters for cliff removals in detail.

3.2 Spatial Sampling and Statistical Similarity for Multi-tier Caching

To construct the *e*MRC miss ratio function, we need to partition caches in each tier repeatedly; this requires spatial sampling to work in multiple cache tiers. Here we are empirically validating that a spatially sampled IO stream can also be used to estimate the miss ratio at the tier-2 cache, extending Kessler's assumption concerning calculating the tier-1 cache miss ratio accurately.

We validated this using the MSR IO traces [1]. Figure 4 shows the approximation error in miss ratio at the tier-2 cache when sampling at different sampling rates ρ before the tier-1 cache, with *no* sampling before the tier-2 cache. We generated the result shown in Figure 4 as follows. For each trace, we evaluated the first 10M entries with 2,601 different tier-1 and tier-2 cache configurations. The tier-1 and tier-2 caches can take 51 different cache sizes incrementing from 0 to *max_cache_size*, based on the unique item count in the first 10M entries of each trace. We calculate the relative difference on the tier-2 cache miss ratio using sampled vs. unsampled traces. We show a bar plot in Figure 4 with more than 70K data points for each spatial sampling rate (i.e., ρ = 0.1, 0.01, and 0.001). The result shows sampling at a 10% rate causes at most 0.62% relative difference across 90% of the data points. Kessler's sampling goal is met even when approximating the miss ratio using only 0.1% of the original trace.

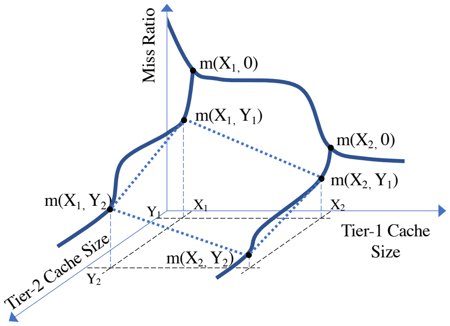


Figure 5: Illustration of a performance cliff region.

This demonstrates that spatial sampling can be used to estimate the tier-2 cache miss ratio. This property supports the construction of the *e*MRC convex miss ratio function and enables rapid cache simulation to explore key miss ratio points in the multi-dimensional miss ratio function.

3.3 Partitioning Scheme for Cliff Removal

This section demonstrates how *e*MRC removes performance cliffs for two-tier caching systems, which can be easily extended to work with multi-tier caches.

Figure 5 illustrates a miss ratio surface in a two-tier caching system. We assume that the *discrete* miss ratio surface $m(x,y)$ is generated from a single IO stream and there are four miss ratio data points, i.e., $m(X_1,Y_1)$, $m(X_1,Y_2)$, $m(X_2,Y_1)$ and $m(X_2,Y_2)$, which surround a performance cliff region. How can we remove the performance cliff using only the four given

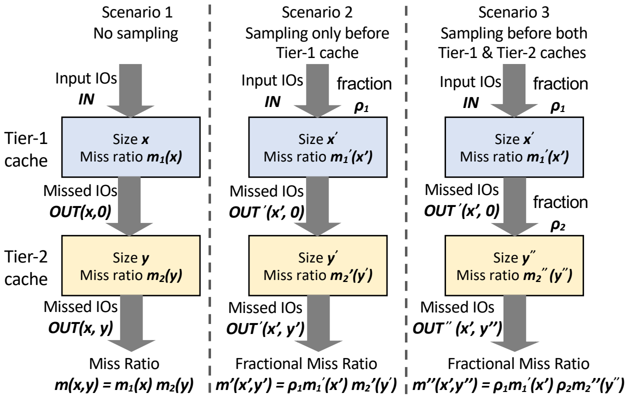


Figure 6: IO sampling scenarios in two-tier cache system.

data points? The Talus algorithm only works for a single-tier cache because it needs two data points using the same input IO stream. For example, in Figure 5, Talus cannot apply between $m(X_1,Y_1)$ and $m(X_2,Y_1)$ because these two data points use different input IO streams; one is the missed IOs when the tier-1 cache size is X_1, and the other is when the tier-1 cache size is X_2.

The *e*MRC algorithm removes performance cliffs for two (or more) tier cache systems using the partitioning scheme outlined in Figure 8.

We present our assumption and theorems that lead to the construction of a miss ratio surface $m_{eMRC}(x,y)$ without performance cliffs. Figure 6 shows three scenarios that cover all possible IO sampling cases in two-tier cache systems. Scenarios 1 and 2 are used in the proof of Theorem 1. All three scenarios are used in the proof of Theorem 2.

- **Scenario 1: No sampling.** Let $m_1(x)$ and $m_2(y)$ denote the cache miss ratios observed at the tier-1 and tier-2 caches, respectively. $OUT(x,0)$ is the missed IO stream after the tier-1 cache, and $OUT(x,y)$ is the one after the tier-2 cache. Then the miss ratio after the tier-1 cache $m(x,0)$ equals $m_1(x)$, and the miss ratio after the tier-2 cache $m(x,y)$ equals $m_1(x)m_2(y)$.
- **Scenario 2: Sampling only before the tier-1 cache.** When a fraction ρ_1 of an IO stream is processed by two cache tiers, $m_1'(x')$ and $m_2'(y')$ denote the cache miss ratios observed at the tier-1 and tier-2 caches, respectively. The miss ratio after the tier-1 cache $m'(x',0)$ will become $\rho_1 m_1'(x')$ due to a fraction ρ_1 of an input IO stream. Then the *fractional miss ratio* is $m'(x',y') = \rho_1 m_1'(x')m_2'(y')$. The fractional miss ratio is the miss ratio when a fraction of an IO stream is used.
- **Scenario 3: Sampling before both tier-1 & tier-2 caches.** We sample the missed IO stream after the tier-1 cache of size x' (which is denoted by $OUT'(x',0)$) again with fraction ρ_2, and feed that into another tier-2 cache of size y''. Then the fractional miss ratio $m''(x',y'')$ equals $\rho_1 m_1'(x')\rho_2 m_2''(y'')$.

With these three scenarios described, we introduce our assumptions and theorems.

Assumption 1 *If $x' = \rho_1 x$ and $y' = \rho_1 y$, then the miss ratio at the tier-2 cache $m_2'(y')$ and $m_2(y)$ are statistically similar.*

This was shown empirically in §3.2. With this assumption we can now find the relationship between the fractional miss ratios $m'(x',y')$, $m''(x',y'')$ in Scenarios 2 & 3 and the miss ratio $m(x,y)$ in Scenario 1 for a two-tier cache system.

Theorem 1 *For an input IO stream that is processed by two tiers of caches with the resulting miss ratio surface $m(x,y)$, a fraction ρ_1 of spatial sampled IO stream will have a fractional miss ratio surface:*

$$m'(x',y') = \rho_1 m(\frac{x'}{\rho_1}, \frac{y'}{\rho_1})$$

Proof: In a two-tier caching setup, when a fraction ρ_1 of an input IO stream passes through the tier-1 cache of size x' and then the tier-2 cache of size y', the fractional miss ratio is $m'(x',y') = \rho_1 m_1'(x')m_2'(y')$. From the Talus theorem, we find $m_1'(x') = m_1(x'/\rho_1)$. Under Assumption 1, $m_2'(y') = m_2(y'/\rho_1)$. Putting them all together, we obtain:

$$m'(x',y') = \rho_1 m_1(\frac{x'}{\rho_1})m_2(\frac{y'}{\rho_1}) = \rho_1 m(\frac{x'}{\rho_1}, \frac{y'}{\rho_1}).$$

Theorem 2 *For a given IO stream that is processed by two tiers of the caches with resulting miss ratio surface $m(x,y)$, downsampling with fraction ρ_1 before the tier-1 cache of size x' and again downsampling with fraction ρ_2 before the tier-2 cache of size y'' results in new fractional miss ratio surface:*

$$m''(x',y'') = \rho_1\rho_2 m(\frac{x'}{\rho_1}, \frac{y''}{\rho_1\rho_2})$$

Proof: From the Talus theorem, we find $m_2''(y'') = m_2'(y''/\rho_2)$. Hence, $m''(x',y'') = \rho_1 m_1'(x')\rho_2 m_2'(\frac{y''}{\rho_2})$. As $m'(x',y') = \rho_1 m_1'(x')m_2'(y')$, $m''(x',y'') = m'(x',\frac{y''}{\rho_2})\rho_2$. Applying Theorem 1, we obtain:

$$m''(x',y'') = \rho_1\rho_2 m(\frac{x'}{\rho_1}, \frac{y''}{\rho_1\rho_2}).$$

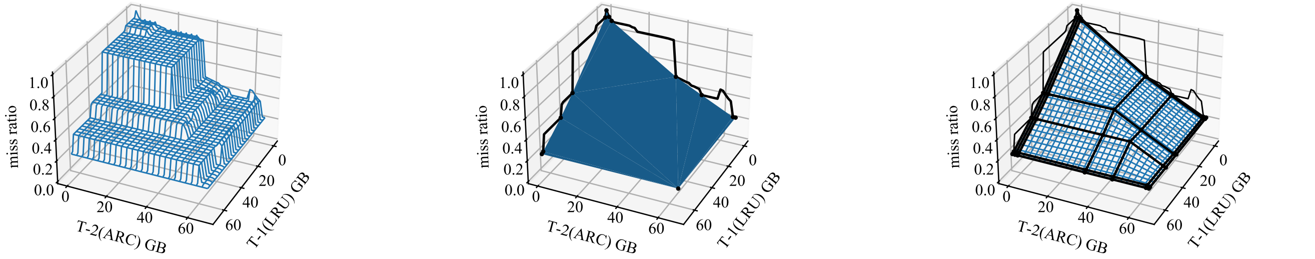
We now use Theorem 2 to remove cliffs in the miss ratio surface. Let $m(x,y)$ denote a region of the miss ratio surface of a two-tier caching system without any partitioning, for $x \in [X_1,X_2], y \in [Y_1,Y_2]$, where $m(X_1,Y_1), m(X_1,Y_2), m(X_2,Y_1)$ and $m(X_2,Y_2)$ are known values. The $m_{eMRC}(x,y)$ can be implemented by partitioning both cache tiers.

Figure 8 illustrates the partitioning scheme. The final miss ratio consists of four different fractional miss ratios after the tier-2 cache. Hence, $m_{eMRC}(x,y)$ is the summation of all the four miss ratios:

$$\begin{aligned} m_{eMRC}(x,y) = m_1(&\sigma_1 x, \rho_1\sigma_2 y) \\ +m_2(&\sigma_1 x, \rho_1(1-\sigma_2)y) \\ +m_3(&(1-\sigma_1)x,(1-\rho_1)\sigma_2 y) \\ +m_4(&(1-\sigma_1)x,(1-\rho_1)(1-\sigma_2)y). \end{aligned} \tag{2}$$

where

$$\rho_1 = \frac{X_2-x}{X_2-X_1}, \sigma_1 = \rho_1\frac{X_1}{x}, \rho_2 = \frac{Y_2-y}{Y_2-Y_1}, \sigma_2 = \rho_2\frac{Y_1}{y}. \tag{3}$$



(a) Original MRC, constructed by evaluating 2,601 cache configurations.

(b) Ideal convex MRC, constructed from original MRC.

(c) eMRC, constructed using 2 MRCs on the edge and 30 additional data points selected by convex hull approximation.

Figure 7: Convex hull approximation applied to MSR trace `web_2`.

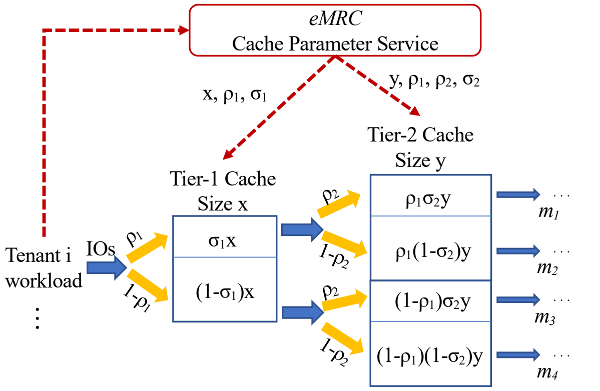


Figure 8: *e*MRC partitioning scheme for two-tier caching.

Since σ_1 and σ_2 are dependent upon ρ_1 and ρ_2, the values of ρ_1 and ρ_2 dictate the convex shape of the $m_{eMRC}(x,y)$ in the region of $x \in [X_1, X_2]$ and $y \in [Y_1, Y_2]$.

Equations 2 and 3 can be combined into:

$$\begin{aligned} m_{eMRC}(x,y) = {} & m_1(\rho_1 X_1, \rho_1\rho_2 Y_1) \\ & + m_2(\rho_1 X_1, \rho_1(1-\rho_2)Y_2) \\ & + m_3((1-\rho_1)X_2, (1-\rho_1)\rho_2 Y_1) \\ & + m_4((1-\rho_1)X_2, (1-\rho_1)(1-\rho_2)Y_2). \end{aligned} \tag{4}$$

By applying Theorem 2 into m_1, m_2, m_3, m_4 in Equation 4, we finally obtain:

$$\begin{aligned} m_{eMRC}(x,y) = {} & \rho_1\rho_2 m(X_1,Y_1) \\ & + \rho_1(1-\rho_2)m(X_1,Y_2) \\ & + (1-\rho_1)\rho_2 m(X_2,Y_1) \\ & + (1-\rho_1)(1-\rho_2)m(X_2,Y_2). \end{aligned} \tag{5}$$

In summary, given a cliff region in $m(x,y)$ with four known boundary values, we can remove the cliffs with our *e*MRC partitioning scheme and parameter set $\{\rho_1, \rho_2\}$.

3.4 Convex Hull Approximation

Figure 7b is an *ideal convex MRC* of the known miss ratio surface, shown in Figure 7a. It is generated using the convex hull algorithm from quickhull [3]. The ideal convex MRC represents the best case in multi-tier cache management. But it requires an algorithm to remove miss ratio cliffs in regions bounded by three arbitrary points in space, which does not exist.

Our partitioning scheme can achieve cliff removals in a "grid" region $(x,y), x \in [X_1, X_2], y \in [Y_1, Y_2]$. To apply it for the whole miss ratio surface, we partitioned the surface using the ideal convex MRC.

In particular, this is a four-step process: (1) Obtain the original MRC; (2) Generate the ideal convex MRC using the convex hull algorithm; (3) Construct *e*MRC grid regions; (4) Perform cliff removal in each region.

The process is very time-consuming because of step (1). In our case, for an MRC surface with cache size resolution $res = 51$, a total of 2,601 miss ratio data points have to be evaluated.

But we found that *the vertices of the ideal convex MRC almost entirely reside on the edge of the surface, where the tier-1 or tier-2 cache size is 0*. And we can find those vertices by constructing 2D convex hulls of the two MRCs, where the tier-1 or tier-2 cache is 0, respectively (black curves in Figure 7b).

Based on our evaluation with all traces, we observed that: at least 81% of the vertices in the *Ideal Convex MRC* are associated with the convex hull points of 2D MRCs along the two edges plus one additional cache configuration when both the tier-1 and tier-2 caches take the maximum values.

We now can greatly simplify the process of applying *e*MRC to the whole surface with new steps (1) and (2): (1) Obtain the two MRCs along the edges where the tier-1 or tier-2 cache is 0, respectively. (2a) Obtain the vertices along the two edges using convex hull algorithm. (2b) Obtain the additional miss ratio values on *e*MRC grid points (highlighted in Figure 7c).

In the particular case outlined in Figure 7, our new approach only needs to evaluate two MRCs along the edges plus 30 additional data points, while the conventional approach requires knowledge of all 2,601 data points.

3.5 Extension to Multi-Tier Caching (3+)

To generalize *e*MRC for N-tier caching, we first extend our notations used in Figures 5, 6 and 8 to vectors:

- $B = \{\mathbf{b} \mid b_i \in \{X_1^i, X_2^i\}, \forall i \in \{1,2,...,N\}\}$ for the set of boundary coordinates that defines a multi-dimensional hypercube cliff region.
- $\mathbf{x} = [x_1, x_2, ...x_N]$ for the cache size on each tier within the cliff region, $X_1^i \leq x_i \leq X_2^i, \forall i \in \{1,2,...,N\}$.
- $\boldsymbol{\rho} = [\rho_1, \rho_2, ...\rho_N]$ for the fraction of an IO stream before each tier.
- $\boldsymbol{\sigma} = [\sigma_1, \sigma_2, ...\sigma_N]$ for the cache partition ratio on each tier.
- $m(\mathbf{x})$ is the original miss ratio function without any cache partitioning.

Theorem 3 (Extension of Theorem 2) *For an input IO stream that is processed by N tiers of the cache with resulting miss ratio function $m(\mathbf{x})$, when a fraction $\boldsymbol{\rho}$ is applied to the IO stream before each cache tier, the new fractional miss ratio function is:*

$$m^N(\mathbf{x}) = \prod_{i=1}^{N} \rho_i \cdot m(\mathbf{x}'), \text{ where } \mathbf{x}' = \left[\frac{x_1}{\rho_1}, \frac{x_2}{\rho_1 \rho_2}, ..., \frac{x_N}{\prod_{i=1}^{N} \rho_i}\right].$$

Extension of Equation 3. The *e*MRC partitioning parameter for each cache tier is defined using the following equations:

$$\rho_i = \frac{X_2^i - x_i}{X_2^i - X_1^i}, \forall i \in \{1,2,...,N\}$$

$$\sigma_i = \rho_i \frac{X_1^i}{x_i}, \forall i \in \{1,2,...,N\}.$$

Extension of Equation 5. The miss ratio after the tier-N cache is composed of 2^N different miss ratios (from 2^N partitions; see Figure 8). Hence, $m_{eMRC}(\mathbf{x})$ is computed by aggregating all the 2^N miss ratios $m_{eMRC}(\mathbf{x}) = \sum_{i=1}^{2^N} m_i^N$, where m_i^N is the miss ratio after each partition at the tier-N cache.

By applying Theorem 3 into this equation, we finally obtain our *e*MRC miss ratio function:

$$m_{eMRC}(\mathbf{x}) = \sum_{\mathbf{b} \in B} \left(\prod_{i=1}^{N} f_i \cdot m(\mathbf{b}) \right),$$

$$\text{where } f_i = \begin{cases} \rho_i, & b_i = X_1^i \\ 1 - \rho_i, & b_i = X_2^i \end{cases}, \forall i \in \{1,2,...,N\}.$$

4 *e*MRC-based Cache Orchestration

We present the ORCA (ORchestration for CAches) multi-tier cache orchestration framework, which identifies the optimal cache configuration that meets tenants' diverse SLO requirements using *e*MRC while also minimizing the cloud provider's cost. ORCA is designed for a cloud provider to serve customers with elastic cache resources. This objective can be achieved by providing each tenant with the lowest possible cache cost to meet its SLO and accommodate as many tenants as possible.

While our approach addresses N-tier caching systems for $N \geq 2$, we consider a two-tier caching distributed storage system to simplify our discussion. The system consists of a faster and more expensive tier-1 cache (e.g., DRAM) closer to the clients and a slower and relatively cheaper tier-2 cache (e.g., NVMe SSD) while having a central storage backend (e.g., SATA SSD). For each tenant i with a specific SLO requirement in IOPS (which is a key SLO metric for storage IO), denoted as R_{IOPS}^i, our cache orchestration framework will find an optimal cache configuration $\{P_1^i, P_2^i, T_1^i, T_2^i\}$ satisfying tenant i's SLO with a provisioned IOPS $P_{IOPS}^i \geq R_{IOPS}^i$ while minimizing overall cache resources allocated, where P_j^i is the cache eviction policy at tier-j for tenant i, and T_j^i is the allocated cache capacity at tier-j for tenant i.

4.1 SLO Modeling with *e*MRC

To find an optimal cache size and policy configuration for a given SLO, we first need to map *e*MRC's miss ratio information to IOPS. The provisioned IOPS of a storage workload depends on both the cache configuration $\{P_1^i, P_2^i, T_1^i, T_2^i\}$ and the performance of the underlying hardware.

An exact analysis of IOPS performance is complicated with MRCs, as it requires all the input IO to be reorganized into an equal chunk size (e.g., 4KB), making it difficult to study any performance benefit of large sequential IOs. We treat both read and write IOs as cache references during MRC generation, modeling a simple write-back cache policy. We present an analytical lower bound of provisioned IOPS using 4KB random IO performance at the tier-1 cache ($IOPS_{T_1}$), the tier-2 cache ($IOPS_{T_2}$), and the storage backend ($IOPS_B$). Miss ratios for each cache tier, M_1^i and M_2^i, as well as the joint miss ratio M^i can be obtained from a *e*MRC miss ratio function for tenant i: $(M^i, M_1^i, M_2^i) = F_{eMRC_i}(T_1^i, T_2^i, P_1^i, P_2^i)$.

The provisioned IOPS of tenant i can then be modeled as:

$$IOPS^i = \frac{1}{\dfrac{1 - M_1^i}{IOPS_{T_1}} + \dfrac{M_1^i - M^i}{IOPS_{T_2}} + \dfrac{M^i}{IOPS_B}} \tag{6}$$

where $1 - M_1^i$, $M_1^i - M^i$ and M^i represents the probability that a storage IO is executed at the tier-1 cache, the tier-2 cache and storage back-end, respectively.

The IOPS function $F_{IOPS}(T_1^i, T_2^i, P_1^i, P_2^i)$ from Eq. 6 is visualized by a 3D plot in Figure 9. This *IOPS surface* shows the relationship of provisioned IOPS with respect to the cache size at each tier for a specific cache policy combination, e.g., tier-1 uses LRU and tier-2 uses ARC.

*e*MRC surface is convex, and the corresponding IOPS surface will be an always increasing surface for the cache size in each tier, according to Eq. 6.

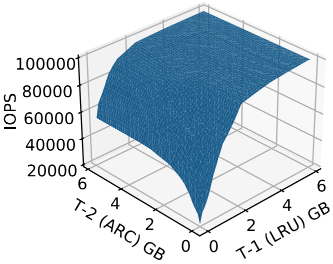


Figure 9: IOPS surface translated from *e*MRC surface for MSR `proj_3` trace.

4.2 Cache Orchestration with *e*MRC

ORCA provides an optimal cache configuration $\{T_1^i, T_2^i, P_1^i, P_2^i\}$ that minimizes the overall cost of cache resources while meeting each tenant's SLO requirements. The cost on cache resource of tenant i with capacity configuration $\{T_1^i, T_2^i\}$ can be denoted as $C^i = C_1 \times T_1^i + C_2 \times T_2^i$, where C_1 and C_2 are the unit costs for the tier-1 and tier-2 caches, respectively.

For M tenants, the problem can be formulated as follows:

$$\begin{aligned}
&\text{minimize} && \sum_{i=1}^{M} C^i && \text{(7a)}\\
&\text{subject to} && C^i = C_1 T_1^i + C_2 T_2^i && \text{(7b)}\\
& && IOPS^i = F_{IOPS}(T_1^i, T_2^i, P_1^i, P_2^i) && \text{(7c)}\\
& && IOPS^i \geq R_{IOPS}^i, \forall i \in \{1, \ldots, M\} && \text{(7d)}\\
&\text{var.} && \{T_1^i, T_2^i, P_1^i, P_2^i\} && \text{(7e)}
\end{aligned}$$

The unit cost of the tier-1 and tier-2 caches (C_1 and C_2) are adjustable by the cloud provider. By adjusting the ratio of C_1/C_2, the cloud provider will be able to shift utilization between the tier-1 and tier-2 caches.

4.3 Two-Stage ORCA Optimization

The cache orchestration problem is challenging even with *e*MRC. We need to search through tons of configuration candidates for each tenant's optimal multi-tier cache configuration, with all the SLO and cache capacity constraints, which would incur a heavy computation overhead. Because an online cache orchestration system requires an efficient and yet accurate algorithm to adapt to large-scale, dynamic workloads, we propose an efficient Two-stage ORCA optimization algorithm. The ORCA optimization splits the optimization over its objective (minimizing cache resource cost) and its constraints (required IOPS and cache sizes), significantly reducing the search space for the optimization problem.

We propose Algorithm 1 to construct the set B^i, which contains cache configurations that are just enough to satisfy $R_{IOPS}^i + D^i$ for tenant i, where R_{IOPS}^i is the requested IOPS from tenant i and D^i is an additional margin added to account for the prediction error caused by spatial downsampling and workload dynamics. We define the tier-1 cache size x_1, algorithm step m as a function of cache size resolution *res* and T_{MAX} [2]. We define x_2 as the tier-2 cache size and the algorithm step as n in a similar way. Then, for each cache policy configuration $\{P_1^i, P_2^i\}$, we first find the first cache configuration (via binary search) that satisfies the IOPS SLO when the tier-2 cache size is zero $x_2[0] = 0$. Then we search among its nearest neighbor points where $x_2[1] = \frac{1}{res-1} \cdot T_{MAX}$ to find the next candidate cache configuration.

Algorithm 1: ORCA-Space Search

```
Input: $T_{MAX}, F_{IOPS}, res, R_{IOPS}^i, D^i$
tier-1 cache size $x_1[m] = \frac{m}{res-1} \cdot T_{MAX}, m \in range(res)$
tier-2 cache size $x_2[n] = \frac{n}{res-1} \cdot T_{MAX}, n \in range(res)$
$R_{IOPS}^i = R_{IOPS}^i + D^i$
for each cache policy combination $(P_1^i, P_2^i)$ do
    Do binary search to find $m$ such that
    $F_{IOPS}(x_1[m-1], x_2[0]) < R_{IOPS}^i$ and
    $F_{IOPS}(x_1[m], x_2[0]) >= R_{IOPS}^i$
    Add $(x_1[m], x_2[0], P_1^i, P_2^i,)$ to $B^i$
    for n in [1, ..., res−1]) do
        $m' = m$
        use $(x_1[m'], x_2[n])$ as starting point to find m such
          that
        $F_{IOPS}(x_1[m-1], x_2[n]) < R_{IOPS}^i$ and
        $F_{IOPS}(x_1[m], x_2[n]) >= R_{IOPS}^i$
        Add $(x_1[m], x_2[n], P_1^i, P_2^i)$ to $B^i$
    end
end
Output: $B^i$, a reduced set of cache configurations
```

The algorithm will complete when the search reaches T_{MAX} for the tier-2 cache size; at this point, B^i will be constructed for a given cache policy configuration. We do this for all cache policy configurations to construct a complete set of B^i for each tenant i; such a set of B^i includes all cache configurations that are just enough to satisfy tenant i's SLO plus a margin. We then apply Algorithm 2 to perform optimization over cache configuration set B^i for each tenant i, which will provide the cache configuration with minimum cache resource cost, defined in Eq. (7a) and Eq. (7b). According to Eq. (7a), $\sum_{i=1}^{M} C^i$ gets minimized when each C^i gets minimized; thus Algorithm 2 aims at minimizing the cost for each tenant i over cache configurations that already meet the SLO requirements from Algorithm 1.

5 Performance Evaluation

We evaluated our *e*MRC approximation technique and ORCA on a server with two Xeon Gold 6142 CPUs at 2.6GHz with 384GB of memory, with a subset of Microsoft's MSR IO traces obtained from SNIA [1, 18]. We treat both read and write IOs as cache references during MRC generation, modeling a simple write-back cache policy. We break any large IOs into 4KB blocks. For IOs smaller than 4KB, we treat them as a full 4KB access. The traces we use all have more

[2] The unique IO entries in the trace.

Algorithm 2: ORCA-Cost Objective

```
for each cache tenant $i$ do
    Call ORCA-Space Search (Algorithm #1) to obtain
      cache configuration set $B^i$
    Initialize $c_{min}$=MAX
    for $b$ in $B^i$ do
        Compute current cost objective value $c$ from $b$
        if $c < c_{min}$ then
            $c_{min} = c$ , $b_{out} = b$
        end
    end
    Add $(i, b_{out}, c_{min})$ to $S$
end
Output: $S$
```

Server	Function	Traces Used
hm	Hardware monitoring	0, 1
mds	Media server	0, 1
prn	Print server	0, 1
proj	Project directories	0, 1, 2, 3, 4
prxy	Firewall/web proxy	0, 1
rsrch	Research projects	0
src1	Source control	0, 1, 2
src2	Source control	0, 1, 2
stg	Web staging	0, 1
ts	Terminal server	0
usr	User home directories	0, 1, 2
wdev	Test web server	0
web	Web/SQL server	0, 1, 2

Table 1: Microsoft MSR traces [1] used.

than one million 4KB IO entries after processing, their names and functions are shown in Table 1. We omitted small traces inappropriate for downsampling methods. In our evaluation for *e*MRC and ORCA, where we used the entire trace, we applied different spatial sampling rates ranging from 0.1 for traces with 1M entries to 0.0001 for traces with more than 600M entries.

For the ORCA evaluation, we assume that we have a storage cloud that uses DRAM as the tier-1 cache, NVMe SSDs as the tier-2 cache, and SATA SSD as the backend storage; and for random 4KB IOs they perform at $IOPS_{T1} = 10,000,000$, $IOPS_{T2} = 100,000$, and $IOPS_B = 20,000$.

5.1 *e*MRC Performance

Computation speedup with convex-hull approximation. The main challenge of utilizing miss ratio information in multi-tier caching is the exponentially increasing size of all cache configurations. With *e*MRC's convex-hull approximation, only a limited number of cache configurations will be evaluated to reduce the computation cost. We evaluated two to four tiers of caches using a mix of stack and non-stack based replacement policies (e.g., LRU→ARC→LRU→ARC) on all the traces we have with resolution $res = 51$, meaning the cache size of each tier can take 51 different values during MRC generation.

We compare the number of data points required to generate *e*MRC's continuous and convex miss ratio function with and without convex-hull approximation. Figure 10 shows the speedup factor for a different number of cache tiers. We can see that the benefits increase with the number of cache tiers. The convex-hull approximation resulted in 14x speedup on average for two cache tiers and 4,527x speedup on average for four cache tiers. We have more speedup for workloads with more cliff regions and less for the ones without many cliffs because the presence of cliffs reduces the number of data points that need to be evaluated. Most real workloads have some level of cliff features, and these features will help to speed up our algorithm. Figure 10 shows the number of data points required but the actual processing time will vary based on the eviction policies used in each cache tier because stack-based eviction policies can benefit from SHARDS, which can evaluate multiple cache sizes with a single pass of the trace. For the two-tier caching using LRU and ARC policy, the largest difference in computation time is 54 seconds for the `src1_0` trace with 200M IO entries with sample rate $R = 0.0002$ compared to 42 minutes for generating the whole original MRC (47 times faster). The smallest difference in computation time is 2 minutes versus 8.2 minutes for the `prxy_0` trace with 22M IO entries with $R = 0.02$ (4.1 times faster).

Accuracy of cliff removal. For each trace, we also evaluated 2,601 individual cache configurations for two-tier caching with *e*MRC partitioning parameters and compared it with the predicted miss ratio using *e*MRC. We use the Mean Absolute Error (MAE) to examine the distance between these two surfaces. As shown in Figure 11, we were able to achieve < 2% MAE for all the traces we use, meaning the *e*MRC partitioned caches act as predicted in removing performance cliffs.

Convex-hull approximation. Our convex-hull algorithm is based on the two miss ratio curves at the edge of the miss ratio surface; the convex hull points along the edges are the majority of the vertices of the ideal convex MRC. The shape of the original miss ratio surface dictates how much we can reduce the miss ratios by removing cliffs, and also the number of regions generated by the convex-hull algorithm, which is related to computation time. Figure 13 highlights the convex-hull algorithm with several traces with representative shapes.

For traces with shapes like `proj_2` and the `web_2` trace shown earlier, our algorithm is able to construct the whole *e*MRC surface with very few data points and significantly reduces the performance cliffs in the entire space.

For traces with shapes like `prxy_0`, the original surface does not have any significant cliffs; our algorithm needs more data points to construct the whole *e*MRC surface. Although it only reduces performance cliffs in small areas, it only needs to evaluate 25% cache size configurations compared with generating the complete miss ratio surface.

For the `prn_0` trace, the particular shape of the original MRC causes our algorithm to produce non-convex regions in

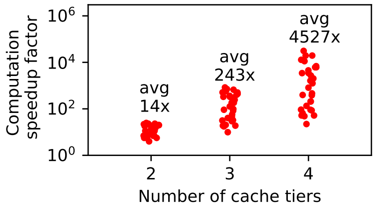


Figure 10: Comparison of data points required to generate the whole *e*MRC, with and without convex-hull approximation.

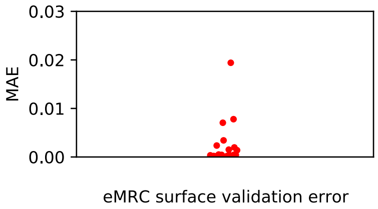


Figure 11: Error between the predicted *e*MRC surface versus the evaluated miss ratio surface using *e*MRC partitioning parameters over all traces.

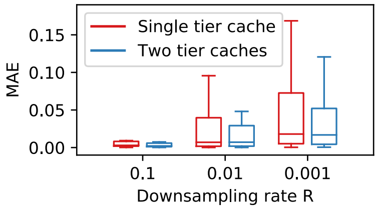


Figure 12: MRC generation error without any partitioning, with respect to different down-sampling factor *R*.

the area marked by red lines in Figure 13. This can be easily corrected by combining the multiple small regions in question into a single large region using the red lines' four boundary points. Since all cache simulations on the marked data points are already done, reconfiguring the grid regions and applying *e*MRC does not have any additional overheads. This also demonstrates the flexibility of our *e*MRC partitioning scheme; it can remove a cliff in any region marked by four boundary points with known miss ratio values.

Spatial downsampling in multi-tier cache. Other researchers have studied the effect of spatial downsampling for a single-tier cache. Our experiments show that it also works well for multi-tier caching. When downsampled with a factor R, the computation time is proportionally reduced by R while maintaining a relatively accurate miss ratio estimation. To evaluate the accuracy, we calculated the MAE between two discrete miss ratio surfaces with and without spatial downsampling using the first 10M entries of all the traces. Both surfaces are generated without any cache partitioning. Figure 12 shows the MAE distribution for different downsampling rates across the evaluation traces. Note that the traces are downsampled only once before the IO stream is processed at the tier-1 cache. We can see that single-tier and two-tier caches have similar error rates.

5.2 ORCA Performance

This subsection shows how our approach can help cloud providers determine efficient cache configurations for cloud tenants with diverse SLOs. We assume the SLOs of tenants are expressed in requested IOPS R_{IOPS}.

Cache optimization for a single tenant. We consider LRU and ARC cache policies. Then four different cache policy combinations are possible (four *e*MRC surfaces per trace). For all the traces, we use resolution $res = 51$. ORCA will first obtain each *e*MRC surface with cache partitioning. *e*MRC can then determine the cache partitioning parameters for each cache size combination in the first and second-tier caches. From the *e*MRC surface, we can derive the F_{IOPS} surface, as shown in Figure 14b. Because the MAE between our *e*MRC prediction and *e*MRC evaluation is very small, the F_{IOPS} surface is also very close to F'_{IOPS} evaluated values (with the Mean Absolute Percentage Error (MAPE) of 1% using all traces). From Figures 14a and 14b, we can see that by utilizing cache partitioning, we not only improved overall IOPS performance but also removed performance plateaus and local dips.

Figure 14b shows F_{IOPS} with *e*MRC cache partitioning where the tier-1 replacement policy is LRU and the tier-2 policy is ARC. Due to page limitations, we are only presenting the results for six traces here. In Figure 14b, the solid blue areas represent cache configurations that lead to $IOPS > R_{IOPS}$. Note that we do not have to search the entire F_{IOPS} surfaces, which contains 2,601 sampling points for every cache policy combination when using resolution $res = 51$. We are only interested in the sampling points that are just enough to meet the tenant's SLO, eliminating unnecessary solutions; hence they will be ≤ 51 cache size combinations for each cache policy pair. By using Algorithm 1, we can calculate the entire related cache configuration set B within a second.

Cache optimization for multiple tenants. In this experiment, we test how ORCA can effectively allocate cache resources for multiple tenants. Six tenants are using the six traces presented in Figure 14 with the corresponding requirements R_{IOPS}, respectively. We assume the cloud provider can handle many tenants and has sufficient cache resources to initially accommodate these six tenants but seeks to limit the resources to meet the requirements. In the ORCA optimization equation (Eq. 7b), the cost ratio between the two cache tiers $C1/C2$ can be tuned by the cloud provider to adjust utilization between them. This evaluation sets $C1/C2 = 10$, roughly representing the per-GB cost difference between DDR4 DRAM and datacenter NVMe SSDs.

Table 2 shows how ORCA allocates cache resources for the six tenants with provisioned IOPS P_{IOPS}. As seen in the table, tenants using `mds_1`, `proj_2` and `src1_0` traces with relatively low R_{IOPS} requirements will be only assigned the tier-2 cache, and other tenants will be assigned combined tier-1 and tier-2 caches.

6 Related Work

Research on efficient cache allocation among tenants has focused on single-tier caching architectures. Two recent

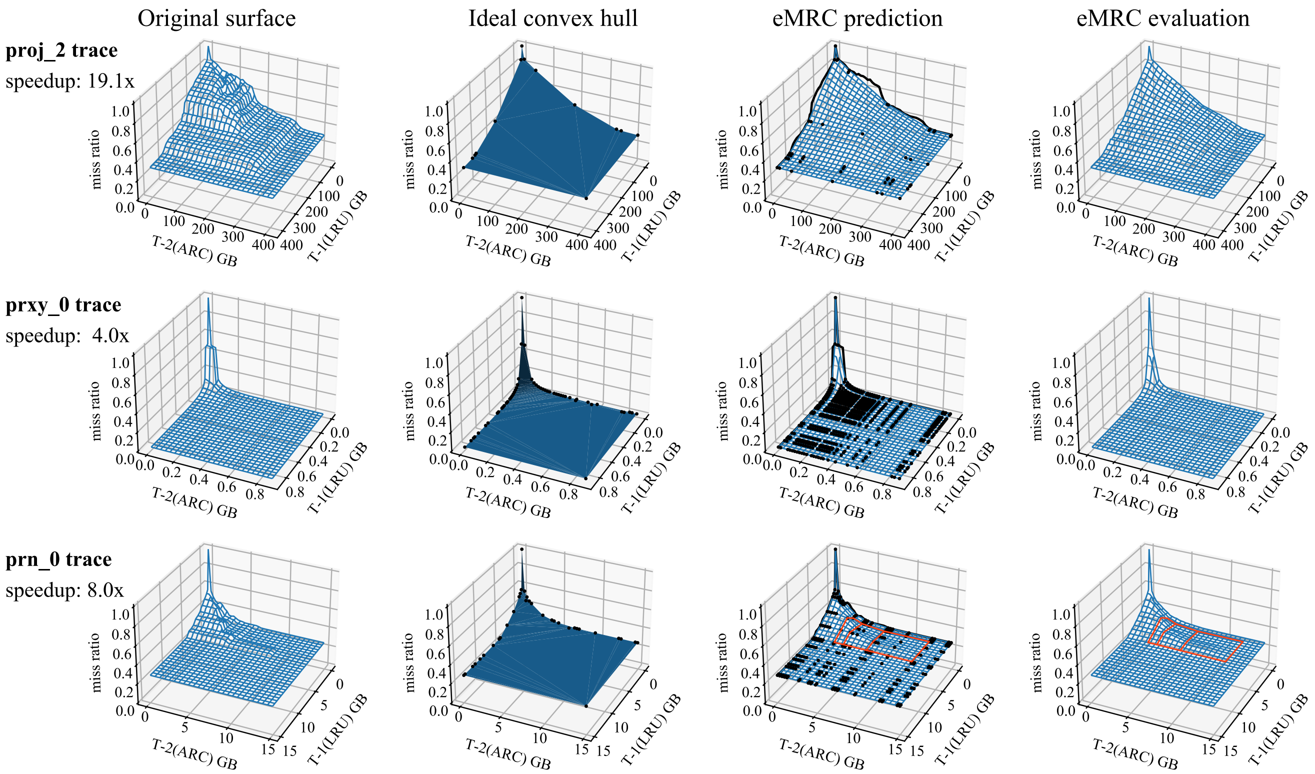


Figure 13: Effect of convex hull algorithm on various original miss ratio surface shapes.

Tenant trace/size	R	R_{IOPS}	P_{IOPS}	Tier1 cache GB/policy	Tier2 cache GB/policy
mds_1/ 25M	0.002	21500	21580	0	6.72/ARC
proj_2/ 340M	0.0002	40000	40397	0	294.29/LRU
proj_3/ 7.7M	0.01	60000	61012	0.12/ARC	1.63/ARC
src2_2/ 17M	0.004	60000	60081	15.79/LRU	19.84/LRU
src1_0/ 406M	0.0002	60000	61785	0	106.34/ARC
web_2/ 74M	0.001	60000	60361	20.08/ARC	65.60/ARC

Table 2: Cache optimization by ORCA for 6 tenants.

works [21, 22] present an approach allowing host-side page caches to be partitioned by VMs individually so those cache parameters can be configured independently between tenants. Cloudcache [2] proposes an on-demand cache management solution to meet each tenant's performance demand by introducing a new cache demand model. Recent MRC approximation techniques [7, 10, 24–26] also have focused on improving MRC efficiency for online cache management for single-tier caching.

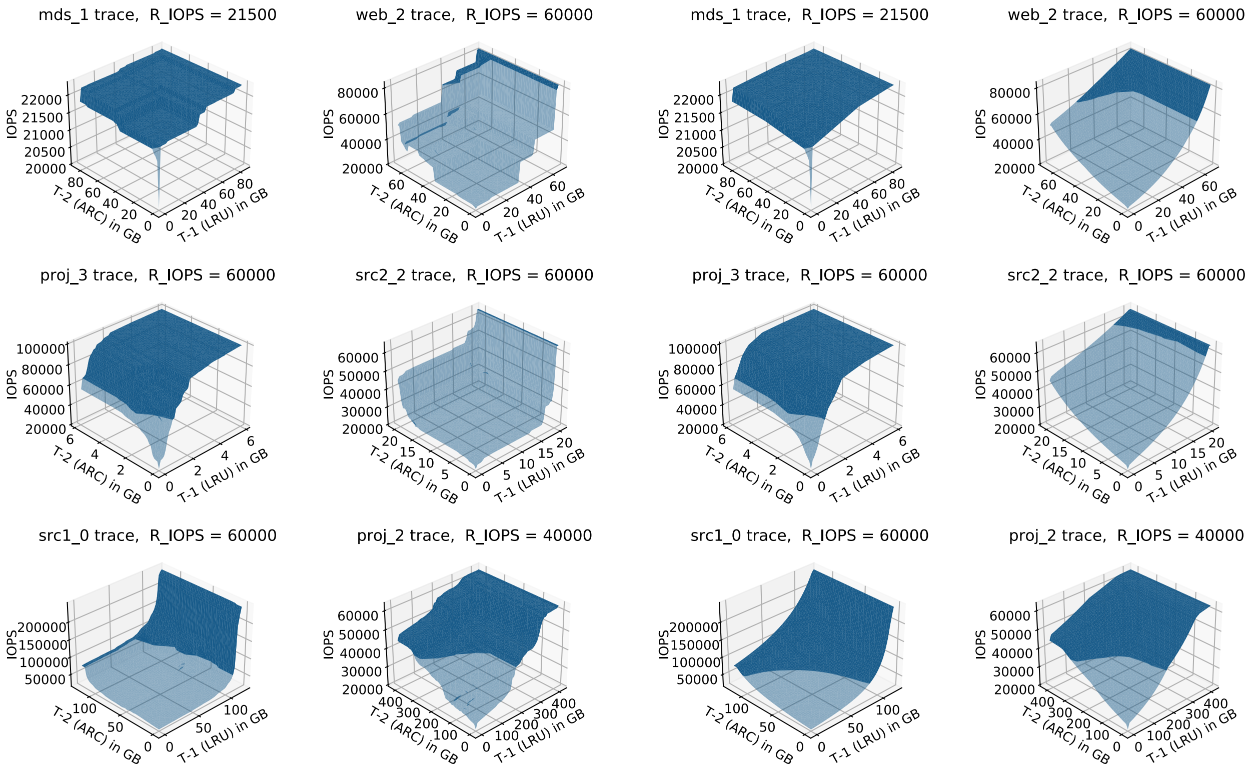
Existing efforts on multi-tier caching include works focusing on minimizing the duplication in cached data among different cache tiers to improve cache efficiency [9, 14, 15, 19, 27–29, 31, 32]. Stefanovici et al. [23] propose a software-defined cache allocation approach for multi-tier caching, which allows cloud providers to coordinate multiple tiers of cache to provide isolation and QoS for tenants.

Dynacache [5] is an LP (Linear Programming) solver to find an optimal slab cache configuration when the total cache size is fixed for single-tenant, single-tier cache scenarios. Cliffhanger [6] extends Dynacache. Cliffhanger uses a novel iterative algorithm to identify a local hit rate gradient without constructing the whole MRC and uses the gradient at different cache slabs to determine the optimal cache configuration. It also utilizes the Talus algorithm to remove performance cliffs. However, both Cliffhanger and Dynacache tackle the single-tier cache focusing on single application optimization, whereas our work targets multi-tier, multi-tenant caching systems.

7 Discussion

*e*MRC allows the efficient study of miss ratio profiles with built-in cliff removal for multi-tier caching systems. It is the first step towards efficient online cache management for multi-tier caching with two challenges: 1. how to efficiently recompute miss ratio profiles/MRCs periodically, 2. how to manage the boundary migration of cache partitions.

Recomputing MRC periodically. Prior works generate the MRC for single-tier cache in a fixed time or using a fixed rate periodically. Talus [4] is originally designed for CPU cache. It recalculates the MRC on a fixed interval (e.g., 10ms) and uses that to configure cache partitions for the next interval. SLIDE [24] in Miniature Simulation recalculates the MRC every 1M IO entries with Exponentially Weighted Moving Average (EWMA). Both Talus and SLIDE require the knowl-



(a) IOPS surfaces without cache partitioning.

(b) IOPS surfaces with cache partitioning.

Figure 14: IOPS surfaces with and without cache partitioning for 6 MSR traces.

edge of the whole MRC to perform cliff removal and cache management. Cliffhanger [6] is another Talus inspired work that can remove performance cliff without knowledge of the whole MRC, but due to the limited visibility, it can only work for one cliff. Talus based approaches require the knowledge of the entire MRC to achieve maximum utility. This makes multi-tier cache analysis and management challenging, as the computation time goes up exponentially. Our *e*MRC with a convex-hull approximation can construct multi-tier miss ratio functions with regional cliff removal while using limited data points without generating the whole original miss ratio function. This enables the timely and periodic generation of multi-dimensional miss ratio functions for multi-tier caching.

Managing the boundary migration of cache partitions. Our *e*MRC based approach relies heavily on cache partitioning within each workload, 2^N partitions for the Nth cache tier to be specific. For a practical online system, the size and partitioning parameters of caches will change over time, and it may cause some cached items to fall into the wrong partitions. Talus [4] builds on top of CPU cache partitioning schemes such as Vantage [20]. SLIDE [24] uses a shadow partitioning based approach to manage partition boundary migration. SLIDE uses a single unified cache to handle IO and defer partitioning decisions till eviction time.

Our future work includes incorporating such boundary migration mechanisms for cache partitions into our ORCA design and evaluating it with more real-world traces.

8 Conclusion

Our *e*MRC approximation technique enables efficient MRC generation for multi-tier caching. *e*MRC uses 1) a partitioning scheme to remove performance cliffs in a grid region with known boundary miss ratio values and 2) a convex hull approximation technique that generates all grid regions efficiently using a small number of sampling points. Based on *e*MRC, we also designed ORCA, a multi-tier cache orchestration framework that uses a lightweight two-stage algorithm that effectively provides efficient cache configurations for tenants with diverse SLOs. Our performance evaluation shows that our *e*MRC and ORCA are useful tools for multi-tier cache orchestration.

Acknowledgments

Thanks to our shepherd Carl Waldspurger and other anonymous reviewers for their valuable feedback and suggestions. This research is supported in part by NSF award #1705095.

References

[1] SNIA IOTTA repository. http://iotta.snia.org.

[2] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. Cloudcache: On-demand flash cache management for cloud computing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 355–369, Santa Clara, CA, 2016. USENIX Association.

[3] C Bradford Barber, David P Dobkin, David P Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

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

[5] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[6] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation NSDI 16)*, pages 379–392, 2016.

[7] D. Eklov and E. Hagersten. Statstack: Efficient modeling of lru caches. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 55–65, March 2010.

[8] Tyler Estro, Pranav Bhandari, Avani Wildani, and Erez Zadok. Desperately seeking... optimal multi-tier cache configurations. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[9] Binny S. Gill. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, 2008. USENIX Association.

[10] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 351–364, Denver, CO, 2016. USENIX Association.

[11] Dejun Jiang, Yukun Che, Jin Xiong, and Xiaosong Ma. ucache: A utility-aware multilevel ssd cache management policy. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 391–398. IEEE, 2013.

[12] Richard E. Kessler, Mark D Hill, and David A Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 43(6):664–675, 1994.

[13] R. Koller, A. J. Mashtizadeh, and R. Rangaswami. Centaur: Host-side ssd caching for storage performance control. In *2015 IEEE International Conference on Autonomic Computing*, pages 51–60, July 2015.

[14] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. Cachededup: In-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, Santa Clara, CA, 2016. USENIX Association.

[15] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-tier cache management using write hints. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 9–9, Berkeley, CA, USA, 2005. USENIX Association.

[16] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.

[17] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[18] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[19] L. Ou, X. He, M. J. Kosa, and S. L. Scott. A unified multiple-level cache for high performance storage systems. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 143–150, Sept 2005.

[20] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 57–68, 2011.

[21] P. Sharma, P. Kulkarni, and P. Shenoy. Per-vm page cache partitioning for cloud computing platforms. In *2016 8th International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–8, Jan 2016.

[22] Prateek Sharma and Purushottam Kulkarni. Singleton: System-wide page deduplication in virtual environments. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 15–26, New York, NY, USA, 2012. ACM.

[23] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 174–181, New York, NY, USA, 2015. ACM.

[24] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, Santa Clara, CA, 2017. USENIX Association.

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

[26] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 335–349, Broomfield, CO, 2014. USENIX Association.

[27] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATC '02, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

[28] C. Wu, X. He, Q. Cao, and C. Xie. Hint-k: An efficient multi-level cache using k-step hints. In *2010 39th International Conference on Parallel Processing*, pages 624–633, Sept 2010.

[29] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-all replacement for a multilevel cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 25–25, Berkeley, CA, USA, 2007. USENIX Association.

[30] Juncheng Yang. Pymimircache. https://github.com/1a1a11a/. Retrieved Dec. 2020.

[31] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):505–519, June 2004.

[32] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 91–104, Berkeley, CA, USA, 2001. USENIX Association.

The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus

Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan[†],
Rathijit Sen[‡], Kwanghyun Park[‡], Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin–Madison [†]*VMware Research* [‡]*Microsoft*

Abstract. We introduce *non-hierarchical caching* (NHC), a novel approach to caching in modern storage hierarchies. NHC improves performance as compared to classic caching by redirecting excess load to devices lower in the hierarchy when it is advantageous to do so. NHC dynamically adjusts allocation and access decisions, thus maximizing performance (e.g., high throughput, low 99%-ile latency). We implement NHC in Orthus-CAS (a block-layer caching kernel module) and Orthus-KV (a user-level caching layer for a key-value store). We show the efficacy of NHC via a thorough empirical study: Orthus-KV and Orthus-CAS offer significantly better performance (by up to 2×) than classic caching on various modern hierarchies, under a range of realistic workloads.

1 Introduction

The notion of a *hierarchy* (i.e., a *memory hierarchy* or *storage hierarchy*) has long been central to computer system design. Indeed, assumptions about the hierarchy and its fundamental nature are found throughout widely used textbooks [28, 46, 85]: “Since fast memory is expensive, a memory hierarchy is organized into several levels – each smaller, faster, and more expensive per byte than the next lower level, which is farther from the processor. [46]”

To cope with the nature of the hierarchy, systems usually employ two strategies: *caching* [3, 73] and *tiering* [5, 43, 93]. Consider a system with two storage layers: a (fast, expensive, small) performance layer and a (slow, cheap, large) capacity layer. With caching, all data resides in the capacity layer, and copies of hot data items are placed, via cache replacement algorithms, in the performance layer. Tiering also places hot items in the performance layer; however, unlike caching, it migrates data (instead of copying) on longer time scales. With a high-enough fraction of requests going to the fast layer, the overall performance approaches the peak performance of the fast layer. Consequently, classic caching and tiering strive to ensure that most accesses hit the performance layer.

While this conventional wisdom of optimizing hit rates may remain true for traditional hierarchies (e.g., CPU caches and DRAM, or DRAM and hard disks), rapid changes in storage devices have complicated this narrative within the modern storage hierarchy. Specifically, the advent of many new non-volatile memories [20, 54, 77] and low-latency SSDs [8, 13, 16] has introduced devices with (sometimes) overlapping performance characteristics. Thus, it is essential to rethink how such devices must be managed in the storage hierarchy.

To understand this issue better, consider a two-level hierarchy with a traditional Flash-based SSD as the capacity layer, and a newer, seemingly faster Optane SSD [8] as the performance layer. As we will show (§3.2), in some cases, Optane outperforms Flash, and thus the traditional caching/tiering arrangement works well. However, in other situations (namely, when the workload has high concurrency), the performance of the devices is similar (i.e., the storage hierarchy is actually not a hierarchy), and thus classic caching and tiering do not utilize the full bandwidth available from the capacity layer. A different approach is needed to maximize performance.

To address this problem, we introduce *non-hierarchical caching* (NHC), a new approach to caching for modern storage hierarchies. NHC delivers maximal performance from modern devices despite complex device characteristics and changing workloads. The key insight of NHC is that when classic caching would send more requests to the performance device than is useful, some of that excess load can be dynamically moved to the capacity device. This improves upon classic caching in two ways. First, by monitoring performance and adapting the requests sent to each device, NHC delivers additional useful performance from the capacity device. Second, NHC avoids data movement between the devices when this movement does not improve performance. While the idea of redirecting excess load to devices lower in the hierarchy applies to both caching and tiering, we focus on caching.

Previous work has addressed some of the limitations of caching [19, 56], offloading excess writes from SSDs to underlying hard drives. However, as we show (§6.4), they have two critical limits: they do not redirect accesses to items present in the cache (hits), and they do not adapt to changing workloads and concurrency levels (which is critical for modern devices).

We implement NHC in two systems: Orthus-CAS, a generic block-layer caching kernel module [32], and Orthus-KV, a user-level caching layer for an LSM-tree key-value store [64]. Under light load, Orthus implementations behave like classic caching; in other situations, they offload excess load at the caching layer to the capacity layer, improving performance. Through rigorous evaluations, we show that Orthus implementations greatly improve performance (up to 2×) on various real devices (such as Optane DCPM, Optane SSD, Flash SSD)

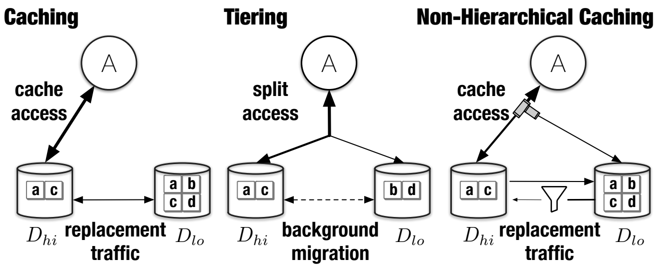


Figure 1: **Caching, Tiering, and Non-Hierarchical Caching.** *The figure shows the different approaches to managing a storage hierarchy. Caching copies data items to the performance layer upon a miss. Tiering splits access to each layer and migrates items in the background (on longer time scales). Non-hierarchical caching (§4), our new approach, offloads excess load at the performance layer to the capacity layer.*

and other simulated ones for a range of workloads (YCSB [35] and ZippyDB [31]). We show NHC is robust to dynamic workloads, quickly adapting to load and locality changes. Finally, we compare NHC against prior caching strategies and demonstrate its advantages. Overall, the non-hierarchical approach extracts high performance from modern storage hierarchies.

2 Motivation

In this section, we discuss classic solutions to storage hierarchy management. We then review current and near-future devices and discuss how they alter the storage hierarchy.

2.1 Managing the Storage Hierarchy

A storage hierarchy is composed of multiple heterogeneous storage devices and policies for transferring data between those devices. For simplicity, we assume a two-device hierarchy, consisting of a *performance* device, D_{hi}, and a *capacity* device, D_{lo}; commonly, D_{hi} is more expensive, smaller, and faster, whereas D_{lo} is cheaper, larger, and slower.

Traditionally, two approaches have been used for managing such a hierarchy: *caching* and *tiering* (Figure 1). With caching, popular (hot) data is copied from D_{lo} into D_{hi} (e.g., on each miss); to make room for these hot data items, the cache evicts less popular (cold) data, as determined by algorithms such as ARC, LRU, or LFU [4,65,67,74,89,104]. The granularity of data movement is usually small, e.g., 4-KB blocks.

Tiering [43,57,81], similar to caching, usually maintains hot data in the performance device. However, unlike caching, when data on D_{lo} is accessed, it is not necessarily promoted to D_{hi}; data can be directly served from D_{lo}. Data is only periodically migrated between devices on longer time scales (over hours or days) and longer-term optimizations determine data placement. Tiering typically does such migration at a coarser granularity (an entire volume or a large extent [43]). While caching can quickly react to workload changes, tiering cannot do so given its periodic, coarser-granularity migration.

Both classic caching and tiering, to maximize performance, strive to ensure that most accesses are served from the performance device. Most caching and tiering policies are thus designed to maximize hits to the fast device. In traditional hierarchies where the performance of D_{hi} is significantly higher than D_{lo}, such approaches deliver high performance. However, with the storage landscape rapidly changing, modern devices have overlapping performance characteristics and thus it is essential to rethink how such devices must be managed.

Example	Latency	**Read** (GB/s)	**Write** (GB/s)	**Cost** ($/GB)
DRAM	80ns	15	15	~7
NVDIMM	300ns	6.8	2.3	~5
Low-latency SSD	10us	2.5	2.3	1
NVMe Flash SSD	80us	~3.0	~2.0	0.3
SATA Flash SSD	180us	0.5	0.5	0.15

Table 1: **Diversified Storage Devices.** *Data taken from SK Hynix DRAM(DDR4, 16GB), Intel Optane DCPM [6,7], low-latency SSDs (Optane SSD 905P [8], Micron X100 SSD [13]), NVMe Flash SSD (Samsung 970 Pro [14,15]) and SATA Flash SSD (Intel 520 SSD [9]). Low-latency SSD and NVMe Flash SSD assume PCIe 3.0.*

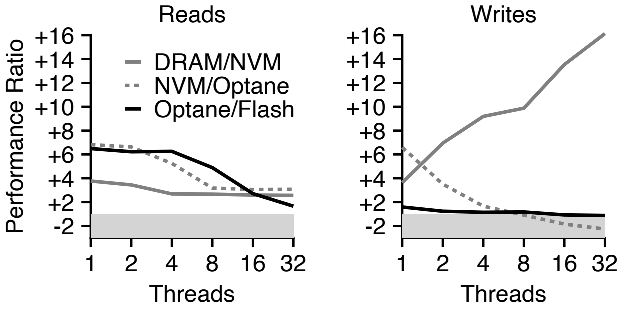


Figure 2: **Performance Ratios Across Modern Devices.** *The ratio of throughput, for varying concurrency, across device pairings. We disable the cache prefetcher and use non-temporal stores for DRAM and NVM. NVM is used as App-Direct mode. Note there is no value between -1 and +1.*

2.2 Hardware Storage Trends

As shown in Table 1, storage systems are undergoing a rapid transformation with a proliferation of high-performance technologies, including persistent memory (e.g., 3D XPoint memory [1,44]), low-latency SSDs (e.g., Intel Optane SSD [8], Samsung Z-SSD [16], and Micron X100 SSD [13]), NVMe Flash SSDs ([14,15]), and SATA Flash SSDs ([9]). Although a seeming ordering exists in terms of latency, bandwidth differences are less clear, and a total ordering is hard to establish.

To better understand the performance overlap of these devices, Figure 2 shows the throughput of a variety of real devices for both 4KB read/load and write/store while varying the level of concurrency. The figure plots the performance ratio between pairs of devices: DRAM/NVM plots the bandwidth of memory (SK Hynix 16GB DDR4) vs. a single Intel Optane DCPM (128GB); NVM/Optane uses the DCPM vs. the Intel 905P Optane SSD; finally, Optane/Flash uses the same Optane SSD and the Samsung 970 Pro Flash SSD. For any pair X/Y, a positive ratio ($\frac{X}{Y}$) is plotted if the performance of X is greater than Y; otherwise, a negative ratio ($\frac{-Y}{X}$) is plotted (in the gray region).

For reads with low concurrency, one can see significant differences between device pairs. Thus, one might conclude that a total ordering exists. However, for reads under high concurrency, the ratios change dramatically. In the most extreme case, the Optane SSD and Flash SSD have nearly identical performance. For writes, the results are even more intriguing; because of the low performance of NVM concurrent writes, in one case (NVM/Optane), the ratio changes from much better under low load to much worse under high load.

To summarize, the following are the key trends in the storage hierarchy. Unlike the traditional hierarchy (e.g., DRAM vs. HDD), the new storage hierarchy may not be a hierarchy; the performance of two neighboring layers (e.g., NVM vs. Optane SSD) can be similar. Second, the performance of new devices vary depending upon many factors including different workloads (reads vs. writes) and level of concurrency. Managing these devices with traditional caching and tiering is no longer effective. Given our focus on improving caching approaches in this paper, we next demonstrate the limitations of caching in modern hierarchies.

3 Characterizing Caching in Traditional and Modern Storage Hierarchies

We now explore caching in different storage hierarchies. Our goal is simple: to understand how caching performs in both traditional and modern hierarchies. In doing so, we hope to build towards a technique that addresses the limitation of caching when running on modern, complex devices and underneath a range of dynamic workloads.

For a deeper intuition, we first model caching performance. We then conduct an empirical analysis on real devices, filling in important details not captured by the model. We also model an approach that we call *splitting* to highlight the drawbacks of classic caching. In splitting, data is simply split across devices, and no migration is performed at run time. Splitting outperforms caching when accesses are optimally split between the performance and capacity devices. In contrast to caching and tiering, splitting is impractical: it is not suitable for workloads where popular items change over time; we use it only as a baseline to build up to our solution.

3.1 Modeling Caching Performance

We assume there are two devices, D_{hi} and D_{lo}, where each performs at a fixed rate, R_{hi} and R_{lo} ops/s; of course, real devices are more complex, with internal concurrency and performance that depends on the workload, but this simplification is sufficient for our purposes.

We also assume that the workload has either little concurrency (i.e., one request at a time) or copious concurrency (i.e., many requests at a time). This allows us to bound the caching performance between these extremes. We assume that the workload is read only; this simplifies our analysis in that we do not account for dirty writebacks upon a cache replacement.

3.1.1 Model

We develop a model of caching performance based on hit rate, $H \in [0,1]$. As stated above, we model two extremes: low and high concurrency. For one request at a time, the average time per request is:

$$T_{cache,1} = H \cdot T_{hit} + (1-H) \cdot T_{miss} \tag{1}$$

T_{hit} is simply the inverse of the rate of the fast device, i.e., $T_{hit} = \frac{1}{R_{hi}}$; T_{miss} is the cost of fetching the data from the slow device and also installing it in the faster device, i.e., $T_{miss} = \frac{1}{R_{hi}} + \frac{1}{R_{lo}}$, or $\frac{R_{hi}+R_{lo}}{R_{hi} \cdot R_{lo}}$.

The resulting bandwidth is the inverse of $T_{cache,1}$:

$$B_{cache,1} = \frac{R_{hi} \cdot R_{lo}}{H \cdot R_{lo} + (1-H) \cdot (R_{hi} + R_{lo})} \tag{2}$$

We now model concurrent workloads. Assume N requests. $H \cdot N$ are hits, $(1-H) \cdot N$ are misses. Note that only misses are serviced by the slow device, whereas *all* requests must be serviced by the fast one (data admissions). The time to process N requests on the slow or fast device is:

$$T_{slow}(N) = N \cdot (1-H) \cdot \frac{1}{R_{lo}} \tag{3}$$

$$T_{fast}(N) = N \cdot (1-H) \cdot \frac{1}{R_{hi}} + N \cdot H \cdot \frac{1}{R_{hi}} = N \cdot \frac{1}{R_{hi}} \tag{4}$$

Total time is the maximum of these two, i.e., whichever device finishes last determines the workload time.

$$\begin{aligned} T_{cache,many}(N) &= max(T_{slow}(N), T_{fast}(N)) && (5) \\ &= max(N \cdot \tfrac{1-H}{R_{lo}}, N \cdot \tfrac{1}{R_{hi}}) && (6) \end{aligned}$$

Dividing by N (not shown) yields the average time per request. Finally, the bandwidth can be computed, as it is the inverse of the average time per request:

$$B_{cache,many} = \frac{1}{max(\frac{1-H}{R_{lo}}, \frac{1}{R_{hi}})} \tag{7}$$

We model splitting performance based on the split rate, $S \in [0,1]$, which determines the fraction S of requests serviced at D_{hi}; the remaining requests $(1-S)$ are served at the tier D_{lo}. Compared to caching, splitting eliminates the cost of installing misses on the faster device. Its throughput can be computed as follows (in a similar way as caching, note the different formula for D_{hi}):

$$B_{split,1} = \frac{1}{\frac{1-S}{R_{lo}} + \frac{S}{R_{hi}}} \tag{8}$$

$$B_{split,many} = \frac{1}{\max(\frac{1-S}{R_{lo}}, \frac{S}{R_{hi}})} \tag{9}$$

3.1.2 Model Exploration

We explore different parameter settings with our model. Figure 3 shows the results for four settings, starting with a large difference in performance between D_{hi} and D_{lo}, and then slowly increasing the performance of D_{lo}.

The first graph shows a traditional hierarchy where the performance of D_{hi} is much (100×) higher than the performance of D_{lo}. This graph shows that both caching and splitting can deliver high performance on traditional hierarchies. The key is to direct as many requests as possible to D_{hi}. Caching and splitting perform well if nearly all requests hit in D_{hi}. Even

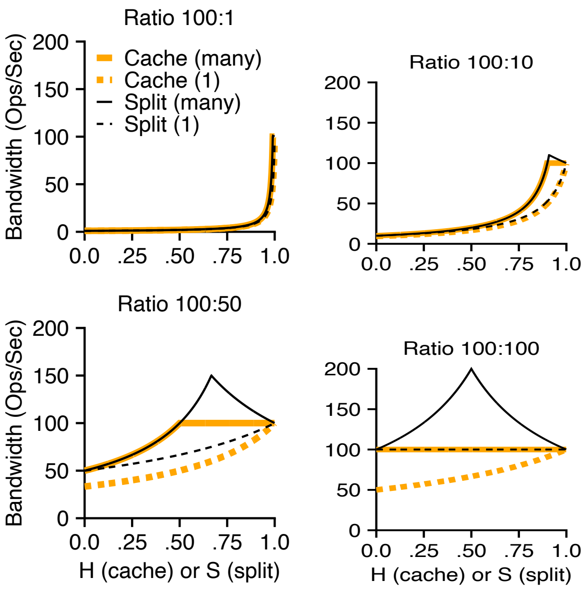


Figure 3: **Modeled Performance.** *This figure shows model-predicted throughput for caching and splitting across a range of different device performance levels. We show performance for high ("many") and low ("1") concurrency. The faster device performs at a fixed rate of 100 ops/sec.*

with 80% hit/split rate, overall performance is quite low, as the slow device dominates.

The next graph (upper right) examines a case where the performance ratio between the devices is still high (10×). Optimizing for a high hit/split rate still works well. Note the slight difference between the low and high concurrency cases; with higher concurrency, these approaches can achieve peak performance even with slightly less than a perfect hit rate, as outstanding requests hide the cost of misses.

The next two graphs represent modern hierarchies where the performance of D_{hi} is closer to that of D_{lo} (D_{hi} delivers bandwidth either 2× D_{lo} or equal to it). We make two important observations from these graphs.

First, classic caching is limited by the performance of D_{hi} and cannot realize the combined performance of both devices. Even with a 100% hit ratio, caching can only deliver 100 ops/sec as it does not utilize the bandwidth of D_{lo}. Splitting (with an optimal split rate) significantly outperforms caching, exposing critical limitations of caching in modern hierarchies.

Second, in modern hierarchies, maximizing the number of requests served by D_{hi} does not always yield the best performance. Consider the case where D_{hi} is 2× faster than D_{lo}. With copious concurrency, when about two-thirds of the requests are directed to D_{hi}, splitting realizes the aggregate bandwidth of D_{hi} and D_{lo}. Increasing the split rate further only degrades performance. Thus, in modern hierarchies, instead of maximizing the hit or split rate, the key is to find the right proportion of requests that must be sent to each device.

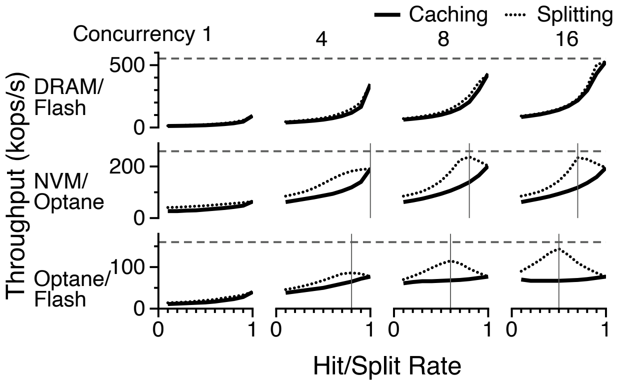


Figure 4: **Performance of Caching and Splitting.** *This figure shows the throughput of read-only workloads. Horizontal dotted lines represent the combined bandwidth of both devices (the maximum possible throughput).*

3.2 Evaluation with Optane DCPM and Optane SSD

Next, we demonstrate that the observations from our model hold for real storage stacks. We use one traditional hierarchy consisting of DRAM and a Flash SSD [14]. We also use two modern stacks: first, NVM (Optane DCPM 128GB) and an Optane 905P SSD; second, an Optane SSD and a Flash SSD. We use these hierarchies to cover a wide range of performance differences; meantime, DCPM and Optane SSD are the most popular emerging devices nowadays. While there could be many hierarchies (e.g., with different versions of these devices), we believe our hierarchies are adequate to validate our modeling and draw meaningful implications for our designs.

For these experiments, we have implemented a new benchmarking tool, called the *Hierarchical Flexible I/O Benchmark Suite (HFIO)*. HFIO contains a configurable hierarchy controller that implements caching and splitting. HFIO uses the LRU-replacement policy for caching. HFIO generates synthetic workloads with a variety of parameters (e.g., mix of reads and writes, locality, and the number of concurrent accesses). HFIO precisely controls the caching layer size and access locality to obtain a desired hit rate. We fix the block size to 32 KB and consider only random accesses. We run our experiments on an Intel Xeon Gold 5218 CPU at 2.3GHz (16 cores), running Ubuntu 18.04. All experiments ran long enough to fill the cache and deliver steady-state performance.

We begin by replicating the results from our model by running read-only workloads and measuring the throughput. Figure 4 shows the results on three hierarchies and workloads with different levels of concurrency. First, in the traditional hierarchy (DRAM+Flash SSD, the first row of Figure 4), as expected, both caching and splitting can achieve high performance. Caching and splitting perform similarly, exactly as our model predicted (Figure 3, 100:1 and 100:10 cases).

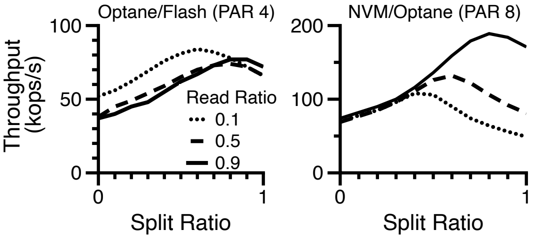
The second two rows of Figure 4 show that caching in new storage hierarchies (e.g., NVM+Optane, Optane+Flash) behaves much differently than in the traditional hierarchy. With low concurrency (1 or 4), the caching device (i.e., DCPM or Optane SSD) is not fully utilized and thus optimizing the hit/split rate still improves performance. However, for workloads with more concurrency, maximizing the hit/split rates does not lead to peak performance in either of the NVM+Optane or Optane+Flash hierarchies. In these situations, capacity devices such as Optane SSD provide substantial performance compared to their caching layers (e.g., DCPM). Splitting (with an optimal split rate) can thus deliver significantly greater performance than caching.

Our experiments with real devices reveal several complexities that the models do not: the optimal split rate depends upon several factors. From Figure 4, we can see that the optimal split rate varies significantly from one device to another and with the level of parallelism of the workload. Write ratios also influence the optimal split rate. As shown in Figure 5, for Optane+Flash, the optimal split rate for a read-heavy workload is 90%, while it is about 60% when the workload is write-heavy. This change occurs because the difference between the write performances of Optane and Flash is smaller than the difference between their read performances. We observe similar results for the NVM+Optane hierarchy.

Summary and Implications: Our performance characterization (modeling and evaluation) of caching provides important lessons for our design. Classic caching is no longer effective in modern hierarchies: it does not exploit the considerable performance that can be delivered by the capacity layer. With high hit rates and when the cache layer is under heavy load, some of the requests can be offloaded to the capacity device. Such high hit rates and heavy load are quite common in production caching systems. For instance, a recent study at Twitter showed that eight out of the ten Twemcache clusters have a miss ratio lower than 5% [98]. Studies have also shown that cache layers often experience heavy load (i.e., they are bandwidth saturated) [17, 56].

In the modern hierarchy, the capacity layer can offer substantial performance and should thus be exploited in such situations. An alternative solution is to increase the number of cache devices in the hierarchy; however, this approach can be prohibitively expensive as performance devices are more costly. In contrast, offloading requests to the capacity layer offers a more economic way to realize significant improvements. Such an offloading approach can deliver the aggregate performance of all devices by optimally splitting the requests to each device. For the offloading approach to work well, it is essential to dynamically adjust the split rate because the optimal rate varies widely in modern hierarchies depending upon factors such as write ratios and level of concurrency.

We note that classic tiering (which also aims to direct most requests to the performance layer) suffers from similar shortcomings as caching in modern hierarchies. In this work, we focus on improving caching for two main reasons. First, getting tiering to optimally split accesses is fundamentally hard. Migration or replication to match the current optimal split in tiering may hurt performance. In contrast, caching can readily bypass cache hits to capacity devices; copies of hot data are always available on both devices. Second, we believe there are many scenarios where caching may be the only suitable solution and tiering may not be appropriate. For instance, applications can only use DRAM as a cache when persistence is required and cannot tier in the DRAM+NVM hierarchy. We believe many systems use caching for such reasons and an approach that improves upon classic caching can be beneficial for many such systems.



(a) Optane SSD + Flash (b) NVM + Optane SSD

Figure 5: **Mixed Reads and Writes Workloads.** *The figure shows the performance of splitting with read-write workloads; PAR: workload parallelism/concurrency.*

4 Non-Hierarchical Caching

We present *non-hierarchical caching* (NHC), a caching framework that utilizes the performance of devices that would have been treated as only a capacity layer with classic caching. NHC has the following **goals**:

(i) Perform as well or better than classic caching. Classic caching optimizes the performance of a storage hierarchy by optimizing the performance from the higher-level device, D_{hi}; this performance is optimized by finding the working set that maximizes the hit rate. NHC should degenerate in the worst-case to classic caching and should be able to leverage any classic caching policy (e.g., eviction and write-allocation).

(ii) Require no special knowledge or configuration. NHC should not make more assumptions than classic caching. NHC should not require prior knowledge of the workload or detailed performance characteristics of the devices. NHC should be able to manage any storage hierarchy.

(iii) Be robust to dynamic workloads: Workloads change over time, in their amount of load and working set. NHC should adjust to dynamic changes.

The main idea of NHC (Figure 1) is to offload excess load to capacity devices when doing so improves the overall caching performance. NHC can be described in three steps. First, when warming up the system (or after a significant workload change), NHC leverages classic caching to identify the current working set and load that data into the D_{hi}; this ensures that NHC performs at least as well as classic caching. Second, after the hit rate has stabilized, NHC improves upon classic caching by sending excess load to the capacity device, D_{lo}. This excess load has two components: read hits that are not delivering additional performance on D_{hi} because D_{hi} is already at its maximum performance, and read misses that cause unnecessary data movement between the two devices. Classic caching moves data from D_{lo} to D_{hi} when a miss occurs to improve the hit rate. However, improving hit rate is not beneficial when D_{hi} is already delivering its maximum performance. Therefore, NHC decreases the amount of data admissions into D_{hi} . Using a feedback-based approach, NHC determines the excess load; it requires no knowledge of the device or the workload. Finally, if a workload change is observed, NHC returns to classic caching; if the workload never stabilizes, the algorithm degenerates to classic caching. NHC can leverage the same write-allocation policies as a classic cache (e.g., write-around or write-back).

4.1 Formal Definitions

To describe NHC, we introduce a few terms. We assume that the storage hierarchy is still composed of two devices, D_{hi} and D_{lo}. Caching performance is determined by how the workload is distributed across those two devices. We denote the total workload over a time period δ_t as a constant W, a finite set of accesses to data items. We use w to refer to the subset of W served by D_{hi}, and use its complement set $W - w$ for that served by D_{lo}. We model performance in the time period δ_t as $\mathbf{P}(W,w) = \mathbf{p}_{hi}(w) + \mathbf{p}_{lo}(W - w)$. We make the following assumptions about the devices:

Assumption 1: Performance of a device has an upper bound. The performance of a device cannot increase after it is fully utilized. L_{hi} and L_{lo} represent the maximum possible performance that can be delivered by each device for the current workload, W. We note w_0 as the smallest subset of w such that $\mathbf{p}_{hi}(w_0) = L_{hi}$.

Assumption 2: Increasing the workload on a device does not decrease performance. This implies $\mathbf{p}_{hi}(x)$ and $\mathbf{p}_{lo}(x)$ are monotonically increasing functions. Note that HDD performance can decrease with more random requests due to more seeks, but the assumption generally holds for high-performing devices such as DRAM, NVM, and SSDs.

Assumption 3: Before reaching upper limits, $\mathbf{p}_{hi}(x)$ has a larger gradient than $\mathbf{p}_{lo}(x)$. Based on our observations from real devices, the potential performance gain of using D_{hi} is greater than that of using D_{lo}.

We define classic caching as an approach that optimizes $\mathbf{P}(W,w)$ by maximizing only $\mathbf{p}_{hi}(w)$. Classic caching attempts to maximize $\mathbf{P}(W,w)$ by finding some working set w_{max} that maximizes the hit rate of D_{hi}.

The key insight of NHC is that, when $w_0 < w_{max}$, an opportunity exists to move some portion of the workload $w_{max} - w_0$ away from D_{hi} to D_{lo}. Since $\mathbf{p}_{hi}(w_0) = \mathbf{p}_{hi}(w_{max}) = L_{hi}$, removing $w_{max} - w_0$ from D_{hi} does not decrease the performance of D_{hi} below L_{hi} and now D_{lo} can deliver some amount of performance for $w_{max} - w_0$. Thus, NHC can always perform as well or better than classic caching.

4.2 Architecture

As shown in Figure 6, classic caching can be upgraded to NHC by adding decision points to its cache controller and a *non-hierarchical cache scheduler*. The classic cache controller serves reads and writes from a user/application to the storage devices (i.e., D_{hi} and D_{lo}) and controls the contents of D_{hi} based on its replacement policy (e.g., LRU). A new cache scheduler monitors performance and controls whether classic caching is performed and where cache read hits are served. The scheduler optimizes a target performance metric that can be supplied by the end-user (e.g., ops/sec) or use device-level measurements (e.g., request latency).

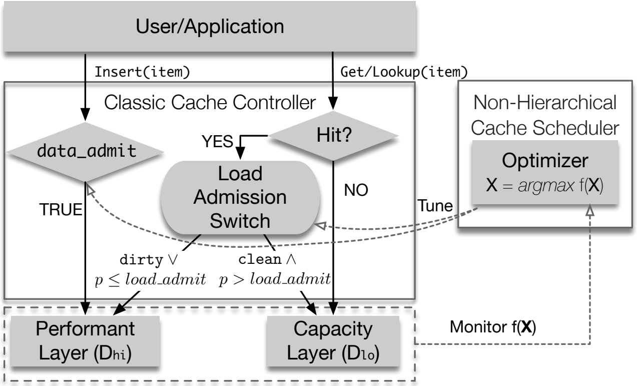


Figure 6: **Non-Hierarchical Caching Architecture.** *This figure shows the architecture of NHC. NHC adds decision points and a scheduler to classic caching. As before, NHC is transparent to users. Any classic caching implementation can be upgraded to be a NHC one. Note that decision points only tune cache read hits/misses.*

The NHC scheduler performs this control with a boolean `data_admit (da)` and a variable `load_admit (la)`. The `da` flag controls behavior when a **read miss** occurs on D_{hi}: when `da` is set, missed data items are allocated in D_{hi}, according to the cache replacement policy; when `da` is not set, the miss is handled by D_{lo} and not allocated in D_{hi}. Classic caching corresponds to the case where the `da` flag is true.

The `la` variable controls how **read hits** are handled and designates the percentage of read hits that should be sent to D_{hi}; when `la` is 0, all read hits are sent to D_{lo}. Specifically, for each read hit, a random number $R \in [0, 1.0]$ is generated; if $R <=$ `la`, the request is sent to D_{hi}; else, it is sent to D_{lo}. In classic caching, `la` is always 1.

The NHC framework works with any classic caching write-allocation policy (specified by users), which handles **write hits/misses**. NHC admits write misses into D_{hi} according to the policy; `da`, `la` do not control write hits/misses. With write-back, cache writes introduce dirty data in D_{hi} and data on D_{lo} can be out-of-date; in this case, NHC does not send dirty reads to D_{lo}.

4.3 Cache Scheduler Algorithm

The NHC scheduler adjusts the behavior of the controller to optimize a target performance metric. As shown in Algorithm 1, the scheduler has two states: increasing the amount of data cached on D_{hi} to maximize hit rate, or keeping the data cached constant, while adjusting the load sent to each device.

Algorithm 1: Non-hierarchical caching scheduler

```
cache: classic cache controller
step: the adjustment step size for load_admit
f(x): function that measures target performance metric when
  load_admit = x, the value is measured by setting
  load_admit = x for a time interval
1  while true do
       # State 1: Improve hit rate
2      data_admit = true,  load_admit = 1.0
3      while cache.hit_rate is not stable do
4          sleep_a_while()
5      data_admit = false,  start_hit_rate = cache.hit_rate
       # State 2: Adjust load_admit
6      while true do
7          ratio = load_admit
           # Measure f(ratio-step) and f(ratio+step)
8          max_f = Max(f(ratio-step), f(ratio), f(ratio+step))
           # Modify load_admit based on the slope
9          if f(ratio-step) == max_f then
10             load_admit = ratio - step
11         else if f(ratio+step) == max_f then
12             load_admit = ratio + step
13         else if f(ratio) == max_f then
14             load_admit = ratio
15             if load_admit == 1.0 then
16                 goto line 2 # Quit tuning if $w < w_0$
           # Check whether workload locality changes
17         if cache.hit_rate < (1-α)start_hit_rate then
18             goto line 2
```

State 1: Improve hit rate. The NHC scheduler begins by letting the cache controller perform classic caching with its default replacement policy (`da` is true and `la` is 1); during this process, the cache is warmed up and the hit rate improves as the working set is cached in D_{hi}. The NHC scheduler monitors the hit rate of D_{hi} and ends this phase when the hit rate is relatively stable; at this point, the performance delivered by D_{hi} for the workload w_{max} is near its peak.

State 2: Adjust load between devices. After D_{hi} contains the working set leading to a high hit rate and performance, the NHC scheduler explores if sending some requests to D_{lo} increases the performance of D_{lo}, while not decreasing the performance of D_{hi}, i.e., the algorithm moves from w_{max} toward w_0. In this state, `da` is set to false and feedback is used to tune `la` to maximize the target performance metric. Specifically, the scheduler (Lines 6–18) modifies `la`; in each iteration, performance is measured with `la +/- step` over a time interval (e.g., 5ms see §5). The value of `la` is adjusted in the direction indicated by the three data points. When the current value of `la` leads to the best performance, the scheduler sticks with the current value. The value of `la` is kept in the acceptable domain of [0, 1.0] with a negative penalty function. If the scheduler finds the optimal `la` is 1, it quits tuning and moves back to State 1; intuitively, this means NHC has moved the current workload w below w_0 and hence requires classic caching to improve the hit rate to further improve performance.

Since NHC relies on classic caching to achieve an acceptable hit rate, it restarts the optimization process when workload locality changes. The NHC scheduler monitors the cache hit rate at runtime; if the current hit rate drops, the scheduler re-enters State 1 to reconfigure the cache with the current working set. If the workload never stabilizes, NHC behaves like classic caching.

Target Performance Metrics: NHC can improve different aspects of performance. NHC can be configured to optimize metrics such as throughput, latency, tail latency, or any combination. The target metrics can also capture performance at various levels of the system (e.g., hardware, OS, or application). f is a function that measures the target metric.

Write Operations: NHC handle writes with the write-allocation policy (specified by users) in the classic cache controller. NHC does not adjust the write-allocation policy because it may be chosen for factors other than performance: endurance [37, 86], persistence, or consistency [59].

Adapting to Dynamic Workloads: NHC periodically measures the target metric (e.g., throughput) using f and optimizes it by adjusting load-admission ratios (in a way similar to gradient-descent). NHC only needs Δf to determine the optimal split of accesses. Since tuning involves only one parameter (load-admission ratio), it is cheap and converges fast. NHC can thus handle frequently-changing workloads with continual tuning.

Summary: Non-hierarchical caching optimizes classic caching to effectively use the performance of capacity devices. NHC improves on classic caching in two ways. First, NHC does not admit read misses into D_{hi} when the performance of D_{hi} is fully utilized. Second, when the performance of D_{hi} is at its peak, NHC delivers useful performance from the D_{lo} device by sending some of the requests that would have hit in D_{hi} to D_{lo} instead. By determining at run-time the appropriate load, NHC obtains useful performance from D_{lo} instead of using D_{lo} only to serve misses into D_{hi}.

5 Implementation

We implement non-hierarchical caching in two places: Orthus-CAS, a generic block-layer caching kernel module, and Orthus-KV, a user-level caching layer for a key-value store.

Open CAS [32] is caching software built by Intel that accelerates accesses to a slow backend block device by using a faster device as a cache. It supports different write-allocation policies such as write-around, write-through, and write-back, and uses an approximate LRU policy for eviction. Open CAS is a kernel module that we modify to leverage NHC. Orthus-CAS works with all policies supported in Open CAS.

We also implement NHC within a persistent block cache for Wisckey [64], an LSM-tree key-value store. LSM trees are a good match for Optane SSD, and have garnered significant industry interest [2,12,38]. Wisckey is derived from LevelDB; the primary difference is that Wisckey separates keys from values to reduce amplification. While keys remain in the LSM-tree, values are stored in a log and each key points to its corresponding value in the log. Separating keys and values also improves caching because it avoids invalidating values when compacting levels; this is similar to the approach in memcached [11] for spilling data to SSD. We integrate NHC with Wisckey's persistent block cache layer. The cache keeps hot blocks (both LSM-tree key and value blocks, 4KB in size) on the cache device using sharded-LRU. Eviction occurs in units of 64 blocks. We call this implementation Orthus-KV.

Detecting Hit-rate Stability: NHC considers the hit-rate to be stable (Algorithm 1, Lines 3-4) when it changes within 0.1% in the last 100-milliseconds. This simple heuristic works well as NHC does not require perfect hit-rate-stability detection. With intensive workloads, a higher hit-rate will only let NHC bypass more hits; with light workloads, NHC switches to classic caching quickly.

Target Performance Metrics: Our implementations support three target metrics: throughput, average latency, and tail (P99) latency, with throughput being the default. When optimizing throughput, we use the Linux block-layer statistics [10] to track device throughputs. When optimizing for latency, we track the end-to-end request latency of the caching system.

Tuning Parameters: NHC implementations must measure the target metrics and tune parameters periodically. The speed at which NHC adapts to workload changes depends on both the interval between target performance measurements and the step size. With a smaller interval, tuning converges faster. Though frequent tuning means more CPU overheads, our CPU overheads are negligible. We found the Linux block layer counters [10] are not accurate when the interval is smaller than 5 ms, so we use the smallest yet accurate interval of 5 ms. A large step size leads to faster convergence but may get sub-optimal results. NHC adjusts the load ratio by 2% in each step; we have found this gives a reasonable converging time with end results similar to smaller step sizes. We leave an adaptive step size for future work.

Implementation Overhead: We find that implementing NHC into existing caching layers is fairly straightforward and requires nominal developer effort. We added only 460 (not including cache mode registration code) and 228 LOC into Open CAS and Wisckey, respectively.

6 Evaluation

Our evaluation aims to answer the following questions:

- How does NHC in Orthus-CAS perform across hierarchies, write-allocation policies, and target metrics? (§6.1)
- How does NHC as implemented in Orthus-KV perform on static workloads? (§6.2)
- How does Orthus-KV handle dynamic workloads? How does it adapt to changes in load and data locality? (§6.3)
- How does NHC compare to previous work? (§6.4)

Setup. We use the following real devices: a SK Hynix DDR4 module (denoted as DRAM), an Intel Optane 128GB DCPM (NVM), an Intel Optane SSD 905P (Optane), and a Samsung 970 Flash SSD (Flash). We also use FlashSim [58] to simulate flash devices with different performance characteristics.

6.1 Orthus-CAS

We begin by evaluating NHC as implemented in Orthus-CAS running on microbenchmarks where the workloads do not change over time. The accesses are uniformly random and 64KB (the suggested page size for Open CAS). We use 1GB of the cache device and generate workloads with different hit ratios. We report the stable performance of classic caching; for NHC, we report when its tuning stabilizes. Unless otherwise noted, we use throughput as the target function.

6.1.1 Storage Hierarchies

We show the normalized throughput of Open CAS and Orthus-CAS for read-only workloads with different hierarchies, amounts of load, and hit ratios in Figure 7. We define Load-1.0 as the minimum read load to achieve the maximum read bandwidth of the cache device; we generate Load-0.5, 1.5, and 2.0 by scaling load-1.0. We investigate hierarchies that include DRAM, NVM, Optane SSD, and Flash. We also mimic hierarchies with two performance differences (50:10 and 50:25) using FlashSim; we configure FlashSim to simulate devices with maximum speeds of 50MB/s, 25MB/s, and 10MB/s. We make the following observations from the figure:

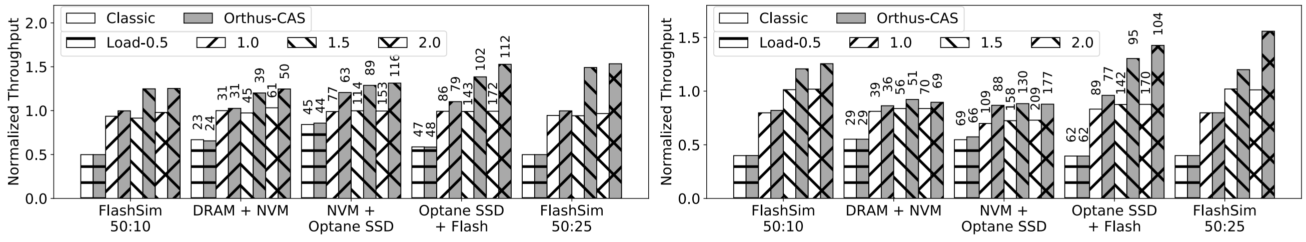
First, when load is light (e.g., Load-0.5), cache devices always outperform capacity devices. In this case, NHC does not bypass any load and behaves the same as classic caching.

Second, when the workload can fully utilize the cache device, Orthus-CAS improves performance. Intuitively, a higher hit ratio and load gives NHC more opportunities bypass requests and improve performance. Figure 7 confirms the intuition: with 95% hit ratio and Load-2.0, NHC obtains improvements of 21%, 32%, 54% for DRAM+NVM, NVM+Optane, and Optane+Flash, respectively. Such improvements are marginally reduced with an 80% hit ratio.

Third, among these hierarchies, Optane+Flash improves the most with Orthus-CAS since the performance difference between Optane and Flash is the smallest, followed by NVM+Optane and DRAM+NVM. Our results with FlashSim show how practitioners can predict the improvement of using NHC on their target hierarchies.

Finally, our measurements indicate that Orthus-CAS adapts to complex device characteristics. With an 80% hit ratio, classic caching does not achieve 1.0 normalized throughput on any real hierarchy because cache misses introduce additional writes to the cache device. NHC handles such complexities.

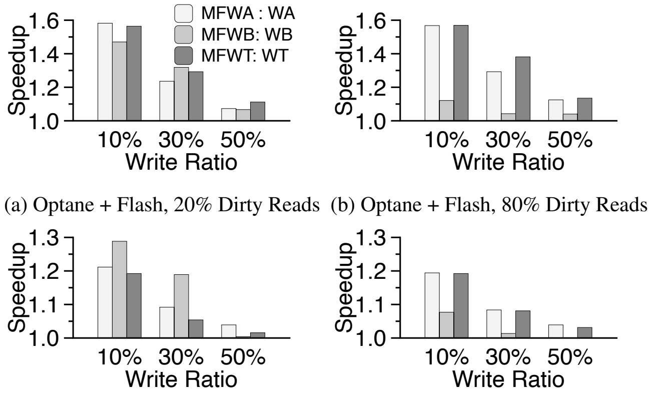
Latency Improvement. As shown in Figure 7, Orthus-CAS also improves average latency on all hierarchies. For instance, with Load-2.0, NHC reduces average latency by 19%, 25%, 39%, for DRAM+NVM, NVM+Optane, and Optane+Flash hierarchies, respectively.



(a) 95% Hit Rate

(b) 80% Hit Rate

Figure 7: **Orthus-CAS on Various Hierarchies.** *Read-only workloads; (a) and (b) show different cache hit rates. All throughputs are normalized to the maximum read bandwidth of the caching device. We show latency (μs) on top of each bar (not comparable across hierarchies).*



(a) Optane + Flash, 20% Dirty Reads (b) Optane + Flash, 80% Dirty Reads

(c) NVM + Optane, 20% Dirty Reads (d) NVM + Optane, 80% Dirty Reads

Figure 8: **Orthus-CAS with Different Write-allocation Policies.** *The figure shows Orthus-CAS overall throughput speedup (against Open CAS) with different write-allocation policies: WA, WB, and WT are write-around, write-back, and write-through. Workloads have a concurrency level of 16 and 95% hit rates.*

6.1.2 Write-Allocation Policies

Open CAS can use a variety of write-allocation policies (write-around, write-back, and write-through) and Figure 8 shows that NHC improves performance relative to classic caching with each policy. The experiments vary the storage hierarchy, the write ratio, and the **dirty-read ratio**. We control the dirty-read ratio by limiting the percentage of the working set that writes can touch (e.g., if writes go to 80% of the working set, then 80% of the reads will be dirty).

NHC improves reads irrespective of write ratios. When reads or writes overload the cache device, NHC bypasses read hits, improving performance (e.g., significantly so on NVM+OptaneSSD where NVM writes interfere with reads dramatically). As shown in Figure 8, the overall improvements are dependent upon a combination of the workload write and dirty-read ratios and the write-allocation policy. NHC offers more benefits when there are fewer writes. With write-back, NHC cannot offload reads of dirty items to the capacity device and thus performs much better with fewer dirty reads. Write-around and write-through maintain consistent copies and thus Orthus-CAS offers benefits independent of the dirty-read ratio.

Target Metric	**NVM + Optane**			**Optane + Flash**		
	Throughput GB/s	Avg. lat. *μs*	P99 lat. *μs*	Throughput GB/s	Avg. lat. *μs*	P99 lat. *μs*
Open CAS	6.7	77	115	2.3	227	269
Throughput	**8.0**	**64**	147	**3.9**	**132**	289
Avg. lat.	**8.0**	**64**	143	**3.9**	**132**	285
P99 lat.	6.9	75	**106**	3.3	155	**245**

Table 2: **Different Target Metrics.** *The figure shows Orthus-CAS performance using different target performance metrics. We use read-only workloads (concurrency level of 8, 95% hit ratio). The best result for each metric is in bold.*

6.1.3 Target Performance Metrics

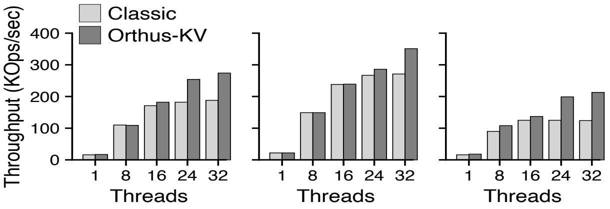
NHC can improve different performance metrics by using different measure functions f. Table 2 shows the performance of Open CAS and Orthus-CAS when using throughput, average latency, and tail (P99) latency as target metrics. Optimizing throughput or average latency yields similar improvements to both metrics on both hierarchies, but increases tail latency. This increase occurs because in each of these storage hierarchies, the performance device has much better tail latency than the capacity device; thus classic caching defaults to appropriate behavior. When NHC is configured with P99 latency as the target metric, Orthus-CAS has better tail latency than Open CAS and than it does with other targets.

6.2 Orthus-KV: Static Workloads

We use Orthus-KV, the NHC implementation in Wisckey, to show the benefits for real applications. Caching in Wisckey uses write-around, due to the LSM-tree's log-structured writes. In these experiments we focus on Optane+Flash since it is often used for key-value stores [12, 38]. We set the caching layer to 33 GB, 1/3 of the 100 GB dataset. We use `cgroup` to limit the OS page cache to 1 GB to focus on caching in the storage system instead of caching in main memory.

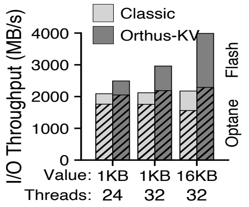
Our initial evaluation uses the YCSB workloads [35]. Most YCSB workloads are constant: their load does not change and they have a stable key popularity distribution (i.e., Zipfian). These workloads cover different read/write ratios (e.g., YCSB-C: 100% reads, YCSB-A: 50% reads and 50% updates), as well as various operations (e.g., YCSB-E involves 95% range queries and 5% inserting new keys, while Workload-F has 50% read-modify-writes). We evaluate YCSB-D as a dynamic workload (§6.3).

Gets: Figure 9 compares the throughput of Wisckey



(a) V:1KB theta:0.6 (b) V:1KB theta:0.8 (c) V:16KB theta:0.8

Figure 9: **Orthus-KV, YCSB-C Performance.** *YCSB-C workload has 100% reads. We use 16B keys and 1KB or 16KB values. Accesses follow a Zipfian distribution (theta).*



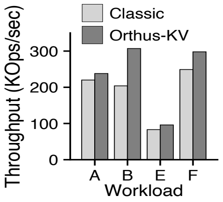


Figure 10: **Bandwidth Breakdown.** *Optane/Flash read bandwidth breakdown during YCSB-C.*

Figure 11: **Other YCSB Workloads.** *16B keys, 1KB values, 32 threads and theta = 0.6.*

and Orthus-KV for three YCSB-C workloads and different amounts of concurrency. Orthus-KV achieves equivalent or higher throughput than Wisckey for all workloads. Orthus-KV significantly improves throughput at high load levels: with 32 threads, 46%, 30%, and 71% higher throughput for the three workloads. When load is high enough to saturate Optane, the relative benefits of Orthus-KV depend on how much it can avoid unnecessary data movement and perform better load distribution. Figure 10 illustrates these two benefits with the read bandwidth delivered by each device. First, classic caching suffers from unnecessary data admissions into Optane: its effective Optane read bandwidth never reaches the peak (2.3GB/s). NHC avoids this wasteful data movement. Second, classic caching never delivers more than the maximum Optane bandwidth to the application. In contrast, NHC improves the performance out of the hierarchy by distributing some cache hits to the Flash SSD.

Updates, Inserts, and Range Queries: Figure 11 shows Orthus-KV handles a range of operations (gets, updates, inserts, and range queries) and always performs at least as well as Wisckey. NHC improves all YCSB workloads, with greater benefits with more get operations.

Latency Improvement: With throughput as its target, Orthus-KV reduces YCSB average latency by up to 42%. For YCSB-C (32 threads, 0.8 theta), Orthus-KV provides 30% and 38% lower latency for 1KB and 16KB values, respectively.

CPU Overhead: Orthus-KV increases CPU utilization slightly (0-2% for 24 threads) due to a few additional counters that track caching behavior and device performance over time.

6.3 Orthus-KV: Dynamic Workloads

We evaluate NHC for dynamic workloads using the same experimental setting as §6.2. We explore how Orthus-KV handles time-varying workloads and dynamic working sets.

6.3.1 Dynamic Load

We evaluate how well NHC handles load changes with the Facebook ZippyDB benchmark [31]. ZippyDB is a distributed key-value store built on RocksDB and used by Facebook. The ZippyDB benchmark generates key-value operations according to realistic trace statistics: 85% gets, 14% puts, 1% scans following a hot range-based model; the request rate models the diurnal load sent to ZippyDB servers. We note that the access patterns (e.g., read sizes) of the ZippyDB benchmark vary significantly as their value sizes range from bytes to MBs. We speed up the replay of Zippydb requests by 1000 to stress the storage system and to better evaluate NHC's reactions to changes in load.

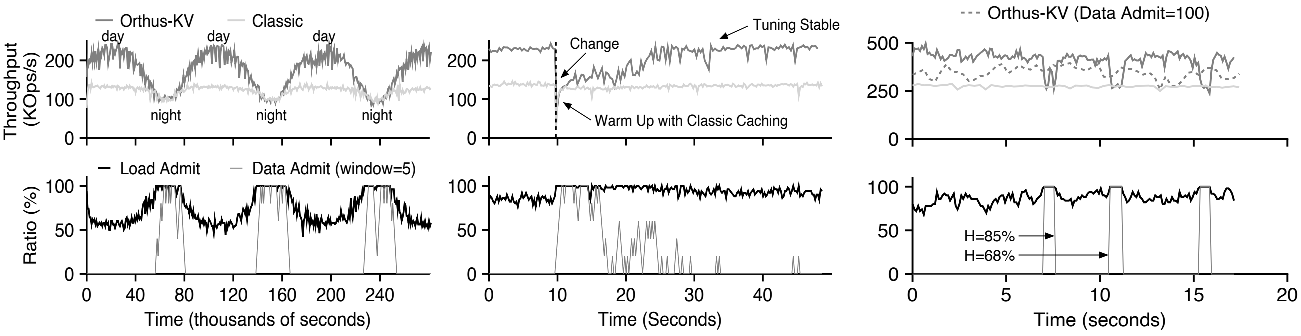
As shown in Figure 12a (top), Orthus-KV outperforms Wisckey during the day by up to 100%, but performs similarly when the load is lower at night. Figure 12a (bottom) shows how Orthus-KV adjusts data and load admit ratios. During the night, both are around 100%; Orthus-KV occasionally adjusts the load admit ratio when the hit rate is stable, but quickly returns to classic caching after finding no improvements. During the day, Orthus-KV keeps the data admit ratio at 0 and adjusts the load admit ratio to adapt to the dynamic load.

6.3.2 Dynamic Data Locality

We demonstrate that NHC reacts well to abrupt changes in the working set in Figure 12b. The experiments base on YCSB-C, beginning with one working set (Zipfian theta=0.8, hot spot at beginning of the key space), and then changing at time 10s (a hot spot at the end of the key space). The graph shows that when the working set changes (time=10s), Orthus-KV quickly detects the change in hit rate and switches to classic caching: the load and data admit ratios increase to 1.0. After the hit rate begins to stabilize (time=11s), Orthus-KV tunes the load admit ratio. Initially (11s-28s), because the hit rate is still not high enough,Orthus-KV often identifies 1.0 as the best load admit and returns to classic caching with data movement. Approximately 20s after the workload change, the hit rate stabilizes and Orthus-KV reaches steady-state performance that is 60% higher than classic caching.

Finally, we show that NHC can outperform classic caching even when the working set changes gradually. Figure 12c shows Orthus-KV's performance on YCSB-D (95% reads, 5% inserts), where locality shifts over time as reads are performed on recently-inserted values. Due to the locality changes and not admitting new data to the cache, the hit rate in Orthus-KV decreases over time, until NHC identifies that 1.0 is the best load admit rate. Then Orthus-KV returns to classic caching and raises the hit rate. Once the hit rate restabilizes, the cycle resumes with Orthus-KV adjusting the load admit rate.

We also explore the alternative approach of always performing data admission while tuning the load admit rate in Figure 12c. As shown, this alternative maintains a stable hit rate, avoiding abrupt phases of admitting new data; this



(a) ZippyDB Workloads (b) Sudden Change in Data Locality (c) Gradual Changes in Data Locality

Figure 12: **Orthus-KV with Dynamic Workloads.** *This figure shows the throughput of Orthus-KV and classic caching (top graphs), as well as load/data admit ratio over time in Orthus-KV (bottom graphs). Because data admit is 0 or 1, we show a fractional windowed sum of its value over 5 time steps for readability. In (a), we replay the ZippyDB benchmark on a single machine. The average value size is 16KB; the number of key ranges is 6. We use 32 threads for the maximum load and adjust the number of threads dynamically according to the diurnal load model. We speed up the two-day workload by 1000×. In (b), the workload is similar to YCSB-C 16KB value, 32 threads, but with two different working sets before and after 10s. In (c), we use YCSB-D with 16B keys, 1KB values, 32 threads. We also show throughput of a modified Orthus-KV that always performs data admit in (c).*

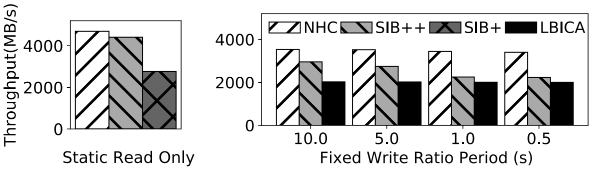
always performs better than classic caching but its peak performance does not reach that of the default Orthus-KV. Our results illustrate the interesting tradeoff between avoiding unnecessary data movement and maintaining a stable hit rate for dynamically changing workloads.

6.4 Comparisons with Prior Approaches

We now show that NHC significantly outperforms two other approaches that have been suggested for utilizing the performance of a capacity device: SIB [56] and LBICA [19]. SIB targets HDFS clusters with many SSDs and HDDs, in which case the aggregate HDD throughput is non-trivial: SIB uses SSDs as a write buffer (does not admit any read miss), and proposes using the HDDs for handling extra read traffic. LBICA determines when a performance layer is under “burst load” at which point it will not allocate new data to the performance layer; unlike NHC, LBICA does not redirect any read hits.

To compare NHC against SIB and LBICA, we have implemented these approaches in Open CAS. To make SIB suitable for general-purpose caching environments, we have improved it in two ways. First, SIB operates on a per-process granularity instead of per-request: the traffic from some processes is not allowed to use the SSD cache; we altered SIB so that it adjusts load per-request (SIB+). Second, we modified SIB so that it admits read misses into the cache (SIB++).

Using the experimental setup from §6.1 on Optane+Flash, we start with a read-only workload in Figure 13.a. SIB+ does not perform well because it does not admit read misses into Optane. SIB++ performs better, but suffers when the workload changes as in Figure 13.b; in these workloads, the amount of write traffic is changed every period, for periods between 10 and 0.5 seconds. SIB cannot handle dynamic workloads because SIB has two phases; in its training phase, SIB learns the maximum performance of the caching device for the current workload; in the inference phase, SIB judges whether the caching device is saturated and, if it is, bypasses some processes (requests). As we have shown, the maximum performance of modern devices depends on many workload parameters: read-write ratios, load, and access patterns. Thus, if the workload changes in any way, SIB must either relearn the target maximum performance or use an inaccurate target. In our experiments, as the duration of each phase decreases, the performance of SIB decreases dramatically. Unlike SIB, NHC uses a simple, continuous feedback-based tuning approach and thus converges rapidly and adapts to dynamic workloads well. Finally, LBICA performs poorly because it does not redirect any read hits to use the capacity device; it simply does not allocate more data to the performance device when it is overloaded.



(a) Static Workload (b) Dynamic Workloads

Figure 13: **NHC vs. SIB and LBICA.** *(a) uses a static read-only workload. (b) uses dynamic workloads; the write-ratio is fixed in each period (e.g., 10s), but changes (randomly between 0% to 50%) across periods. We use workloads with parallelism/concurrency 16, 95% hits, runtime 100s on Optane+Flash hierarchy.*

7 Related Work

Algorithms and Policies in Hybrid Storage Systems: Algorithms and policies for managing traditional hierarchy have been studied extensively [65, 67, 68, 70, 78, 89, 91, 96, 104]. Techniques have been introduced to optimize data allocation [3, 49, 80, 84, 89, 90], address translation [25, 82], identify hot data [4, 51, 53, 71, 73, 75, 79, 88, 91, 95] and perform data migration [26, 33, 40, 43, 70, 87, 91, 100]. Most previous work improves performance by focusing on workload access locality. In contrast, NHC improves by taking all devices and workloads into account.

Storage Optimization: A long line of pioneering work in storage management [21–23, 89, 91] shows how to trace workloads and optimize storage decisions for improved performance; NHC could fit into such a system, making short-term decisions to handle more dynamic workload changes, leaving longer-term optimization to a higher-level system.

Storage Aware Caching/Tiering: Our paper shares aspects with storage-aware caching/tiering [19,29,42,43,47,52, 56,59,60,72,93,99], which considers more factors than hit rate. For instance, Oh et al. [72] propose over-provisioning in Flash to avoid the influence of SSD garbage collection. Modern devices like NVDIMM and Optane SSD have distinctive characteristics compared to Flash. We study the implications of these important emerging devices to caching/tiering. BATMAN [34] shares a similar motivation to NHC: classic caching is not effective when the bandwidth of the capacity layer is a significant fraction of overall bandwidth. However, it investigates a much simpler hierarchy with fixed performance difference (4:1 between high-bandwidth memory and DRAM). Given the fixed difference, BATMAN splits cache accesses between HBM and DRAM statically. This approach would not work effectively on modern hierarchies where performance differences vary dynamically (e.g., depending upon the amount of writes or the level of parallelism in the workload). Wu et al. [93] study tiering on SSDs and HDDs and recognize a similar problem: SSDs (or faster devices) can be the throughput bottleneck. To mitigate the problem, they proposed to periodically migrate data from SSDs to HDDs when the SSD response time is higher than that of HDDs. This approach is limited in three aspects. First, due to its tiering nature, it cannot react to workload changes quickly, its migration traffic can be significant, and it requires extra metadata to track objects across devices. Second, similar to SIB approach, it estimates workload intensity in a period and then migrates data based on the estimation; it hence struggles with dynamic workloads. Third, it is tuned for a specific hierarchy (SSDs and hard drives). Unlike this approach, NHC focuses on improving caching, adapts its behavior during runtime, can react to complex and dynamic workloads, and works well on a range of modern devices.

Managing NVM-based Devices: Other related work integrates NVM-based devices into the memory-storage hierarchy [18, 27, 44, 45, 48, 63, 69, 83]. This work includes extensive measurements for both Optane SSD [92] and Optane DCPM [50, 97]. New file systems and databases were proposed to manage NVDIMM [76, 94] and low latency SSDs [61, 66, 101]. Many works have evaluated the potential benefits of caching and tiering on NVM. Kim et al. [55] provide a simulation-based measurement of NVM caching with performance numbers from a Micron all-PCM SSD prototype. [41] provides a I/O cache simulator that assists the analysis of caching workloads on new storage hierarchies. Strata [62] and Ziggurat [103] are file systems that tier data across a DRAM, NVM, and SSD hierarchy. Dulloor et al. [36], Arulraj et al. [24] and Zhang et al. [102] proposed NVM-aware data placement strategies for the new storage hierarchy. These strategies optimize data placements in a longer period (e.g., offline or periodically). NHC can work with them, providing further improvement by handling more dynamic workload changes. Finally, there have been many companies utilizing NVM/ Optane SSD as a caching layer [30, 38, 39]. Our paper is the first to analyze general caching and tiering on modern hierarchies through modeling and empirical evaluation. We are also the first to propose a generic solution (NHC) to realize the full performance benefits of such a hierarchy.

8 Conclusion

In this paper, we show how emerging storage devices have strong implications for caching in modern hierarchies. We introduced non-hierarchical caching, a new approach optimized to extract peak performance from modern devices. NHC is based upon a novel cache scheduling algorithm, which accounts for workload and device characteristics to make allocation and access decisions. Through experiments, we showed the benefits of NHC on a wide range of devices, cache configurations, and workloads. We believe NHC can serve as a better foundation to manage storage hierarchies.

Acknowledgments

We thank Song Jiang (our shepherd), the anonymous reviewers and the members of ADSL for their valuable input. This material was supported by funding from NSF CNS-1838733, CNS-1763810, VMware, Intel, Seagate, and Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or any other institutions.

References

[1] 3D XPoint. https://en.wikipedia.org/wiki/3D_XPoint.

[2] Accelerate Ceph Clusters with Intel Optane DC SSDs. https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/accelerate-ceph-clusters-with-optane-dc\-ssds-brief.pdf.

[3] Cache (computing). https://en.wikipedia.org/wiki/Cache_(computing).

[4] Cache replacement policies. https://en.wikipedia.org/wiki/Cache_replacement_policies.

[5] Caching and Tiering. https://storageswiss.com/2014/01/15/whats-the-difference-between\-tiering-and-caching/.

[6] Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[7] Intel Optane DIMM Pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html.

[8] Intel Optane SSD 905P. https://www.tomshardware.com/reviews/intel-optane-ssd-905p,5600-2.html.

[9] Intel SSD 520 Series. https://ark.intel.com/content/www/us/en/ark/products/series/66202/intel-ssd-520-series.html.

[10] Linux block layer statistics. https://www.kernel.org/doc/Documentation/block/stat.txt.

[11] Memcached Exstore. https://memcached.org/blog/nvm-caching/.

[12] Micron Heterogeneous-Memory Storage Engine. https://www.micron.com/products/advanced-solutions/heterogeneous-memory-storage-engine.

[13] Micron X100 NVMe SSD. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology/x100.

[14] Samsung 970 Pro. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/970pro/.

[15] Samsung 980 Pro Flash SSD. https://www.anandtech.com/show/15352/ces-2020\-samsung-980-pro-pcie-40-ssd-makes\-an-appearance.

[16] Samsung Z-NAND SSD. https://www.samsung.com/semiconductor/ssd/z-ssd/.

[17] SDC2020: Caching on PMEM: an Iterative Approach. https://www.youtube.com/watch?v=lTiw4ehHAP4, 2020.

[18] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-Mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 971–985. ACM, 2019.

[19] Saba Ahmadian, Reza Salkhordeh, and Hossein Asadi. Lbica: A load balancer for i/o cache architectures. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1196–1201. IEEE, 2019.

[20] Ameen Akel, Adrian M Caulfield, Todor I Mollov, Rajesh K Gupta, and Steven Swanson. Onyx: A prototype phase change memory storage array. *HotStorage*, 1:1, 2011.

[21] Guillermo A Alvarez, Elizabeth Borowsky, Susie Go, Theodore H Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems (TOCS)*, 19(4):483–518, 2001.

[22] Eric Anderson, Michael Hobbs, Kim Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[23] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, and Qian Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems (TOCS)*, 23(4):337–374, 2005.

[24] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. Multi-tier buffer management and storage system design for non-volatile memory. *arXiv preprint arXiv:1901.10938*, 2019.

[25] Shi Bai, Jie Yin, Gang Tan, Yu-Ping Wang, and Shi-Min Hu. Fdtl: a unified flash memory and hard disk translation layer. *IEEE Transactions on Consumer Electronics*, 57(4):1719–1727, 2011.

[26] Swapnil Bhatia, Elizabeth Varki, and Arif Merchant. Sequential prefetch cache sizing for maximal hit rate. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 89–98. IEEE, 2010.

[27] Timothy Bisson and Scott A Brandt. Flushing policies for nvcache enabled hard disks. In *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, pages 299–304. IEEE, 2007.

[28] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 281. Prentice Hall Upper Saddle River, 2003.

[29] Nathan C Burnett, John Bent, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Exploiting gray-box knowledge of buffer-cache management. In *USENIX Annual Technical Conference, General Track*, pages 29–44, 2002.

[30] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

[31] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[32] Open CAS. Open Cache Acceleration Software. `https://open-cas.github.io/`.

[33] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing belady's limitations: In search of flash cache offline optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 379–392, 2016.

[34] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram. In *Proceedings of the International Symposium on Memory Systems*, pages 268–280, 2017.

[35] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, pages 143–154, Indianapolis, IN, June 2010.

[36] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 15. ACM, 2016.

[37] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, 2019.

[38] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with *NVM* in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018.

[39] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using Non-volatile Memory for Storing Deep Learning Models. *arXiv preprint arXiv:1811.05922*, 2018.

[40] Ahmed Elnably, Hui Wang, Ajay Gulati, and Peter J Varman. Efficient qos for multi-tiered storage systems. In *HotStorage*, 2012.

[41] Tyler Estro, Pranav Bhandari, Avani Wildani, and Erez Zadok. Desperately seeking... optimal multi-tier cache configurations. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[42] Brian Forney, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2002.

[43] Jorge Guerra, Himabindu Pucha, Joseph S Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *FAST*, volume 11, pages 20–20, 2011.

[44] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform Storage Performance With 3D XPoint Technology. *Proceedings of the IEEE*, 105(9), 2017.

[45] Theodore R Haining and Darrell DE Long. Management policies for non-volatile write caches. In *1999 IEEE International Performance, Computing and Communications Conference (Cat. No. 99CH36305)*, pages 321–328. IEEE, 1999.

[46] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[47] David A Holland, Elaine Angelino, Gideon Wald, and Margo I Seltzer. Flash caching on the storage client. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 127–138, 2013.

[48] Morteza Hoseinzadeh. A survey on tiering and caching in high-performance storage systems. *arXiv preprint arXiv:1904.11560*, 2019.

[49] Ilias Iliadis, Jens Jelitto, Yusik Kim, Slavisa Sarafijanovic, and Vinodh Venkatesan. Exaplan: queueing-based data placement and provisioning for large tiered storage systems. In *2015 IEEE 23rd International*

Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pages 218–227. IEEE, 2015.

[50] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[51] Jaeheon Jeong and Michel Dubois. Cost-sensitive cache replacement algorithms. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 327–337. IEEE, 2003.

[52] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. Dulo: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, volume 4, pages 8–8, 2005.

[53] Shudong Jin and Azer Bestavros. Popularity-aware greedy dual-size web proxy caching algorithms. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 254–261. IEEE, 2000.

[54] Takayuki Kawahara. Scalable Spin-transfer Torque RAM Technology for Normally-off Computing. *IEEE Design & Test of Computers*, 28(1):52–63, 2010.

[55] Hyojun Kim, Sangeetha Seshadri, Clement L Dickey, and Lawrence 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 (FAST 14)*, pages 33–45, 2014.

[56] Jaehyung Kim, Hongchan Roh, and Sanghyun Park. Selective i/o bypass and load balancing method for write-through ssd caching in big data analytics. *IEEE Transactions on Computers*, 67(4):589–595, 2017.

[57] Youngjae Kim, Aayush Gupta, Bhuvan Urgaonkar, Piotr Berman, and Anand Sivasubramaniam. Hybrid-Store: A Cost-Efficient, High-Performance Storage System Combining SSDs and HDDs. MASCOTS '11, 2011.

[58] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A Simulator for Nand Flash-based Solid-State Drives. In *Proceedings of the First International Conference on Advances in System Simulation (SIMUL '09)*, Porto, Portugal, September 2009.

[59] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 45–58, 2013.

[60] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. Centaur: Host-side ssd caching for storage performance control. In *2015 IEEE International Conference on Autonomic Computing*, pages 51–60. IEEE, 2015.

[61] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast {NVM} storage with udepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, 2019.

[62] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[63] Chun-Hao Lai, Jishen Zhao, and Chia-Lin Yang. Leave the cache hierarchy operation as it is: A new persistent memory accelerating approach. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[64] Lanyue Lu and Thanumalayan Sankaranarayana Pillai and 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)*, pages 133–148, Santa Clara, California, February 2016.

[65] Donghee Lee, Jongmoo Choi, Jun-Hum Kim, Sam H. Noh, Sang Lyul Min, Yookum Cho, and Chong Sang Kim. On The Existence Of A Spectrum Of Policies That Subsumes The Least Recently Used (LRU) And Least Frequently Used (LFU) Policies. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, Atlanta, Georgia, May 1999.

[66] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[67] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[68] Michael Mesnier, Feng Chen, Tian Luo, and Jason B Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 57–70. ACM, 2011.

[69] Sparsh Mittal and Jeffrey S Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2015.

[70] David Montgomery. Extent migration scheduling for multi-tier storage architectures, November 5 2013. US Patent 8,578,107.

[71] Junpeng Niu, Jun Xu, and Lihua Xie. Hybrid storage systems: a survey of architectures and algorithms. *IEEE Access*, 6:13385–13406, 2018.

[72] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST*, volume 12, 2012.

[73] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[74] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, pages 297–306, Washington, DC, May 1993.

[75] Dongchul Park and David HC Du. Hot Data Identification for Flash-Based Storage Systems Using Multiple Bloom Filters. In *Proceedings of the 27th IEEE Symposium on Mass Storage Systems and Technologies (MSST '11)*, Denver, Colorado, May 2011.

[76] Sheng Qiu and AL Narasimha Reddy. Nvmfs: A hybrid file system for improving random write in nand-flash ssd. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5. IEEE, 2013.

[77] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[78] Benjamin Reed and Darrell DE Long. Analysis of caching algorithms for distributed file systems. *ACM SIGOPS Operating Systems Review*, 30(3):12–21, 1996.

[79] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '90)*, Boulder, Colorado, April 1990.

[80] Reza Salkhordeh, Hossein Asadi, and Shahriar Ebrahimi. Operating system level data tiering using online workload characterization. *The Journal of Supercomputing*, 71(4):1534–1562, 2015.

[81] Mohit Saxena, Michael M Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 267–280. ACM, 2012.

[82] Andre Schaefer and Matthias Gries. Adaptive address mapping with dynamic runtime memory mapping selection, 2012. US Patent 8,135,936.

[83] Priya Sehgal, Sourav Basu, Kiran Srinivasan, and Kaladhar Voruganti. An empirical study of file systems on nvm. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.

[84] Haixiang Shi, Rajesh Vellore Arumugam, Chuan Heng Foh, and Kyawt Kyawt Khaing. Optimal disk storage allocation for multitier storage system. *IEEE Transactions on magnetics*, 49(6):2603–2609, 2013.

[85] Abraham Silberschatz, Greg Gagne, and Peter B Galvin. *Operating system concepts*. Wiley, 2018.

[86] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *FAST*, volume 10, pages 101–114, 2010.

[87] Elizabeth Varki, Allen Hubbe, and Arif Merchant. Improve prefetch performance by splitting the cache replacement queue. In *IEEE International Conference on Advanced Infocomm Technology*, pages 98–108. Springer, 2012.

[88] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ml-based lecar. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[89] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, 2017.

[90] Hui Wang and Peter Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 229–242, 2014.

[91] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[92] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of intel optane ssd. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19). USENIX Association, Renton, WA*, 2019.

[93] Xiaojian Wu and AL Narasimha Reddy. A novel approach to manage a hybrid storage system. *JCM*, 7(7):473–483, 2012.

[94] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[95] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-all replacement for a multilevel cache. In *Fast*, volume 7, pages 25–25, 2007.

[96] Gala Yadgar, Michael Factor, and Assaf Schuster. Cooperative caching with return on investment. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13. IEEE, 2013.

[97] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. *arXiv preprint arXiv:1908.03583*, 2019.

[98] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, 2020.

[99] Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. Autotiering: automatic data placement manager in multi-tier all-flash datacenter. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2017.

[100] Gong Zhang, Lawrence Chiu, and Ling Liu. Adaptive data migration in multi-tiered storage based cloud environment. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 148–155. IEEE, 2010.

[101] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. Flashshare: punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 477–492, 2018.

[102] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal data placement for heterogeneous cache, memory, and storage systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, pages 1–27, 2020.

[103] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: a tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, 2019.

[104] Yuanyuan Zhou, James F. Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, pages 91–104, Boston, Massachusetts, June 2001.

A Community Cache with Complete Information

Mania Abdi[⋆], *Amin Mosayyebzadeh*[◇], *Mohammad Hossein Hajkazemi*[⋆], *Emine Ugur Kaynar*[◇],
Ata Turk[‡], *Larry Rudolph*[†], *Orran Krieger*[◇], *Peter Desnoyers*[⋆]
[‡]*State Street,* [†]*TwoSigma,* [◇]Boston University, [⋆]Northeastern University

Abstract

Kariz is a new architecture for caching data from datalakes accessed, potentially concurrently, by multiple analytic platforms. It integrates rich information from analytics platforms with global knowledge about demand and resource availability to enable sophisticated cache management and prefetching strategies that, for example, combine historical run time information with job dependency graphs (DAGs), information about the cache state and sharing across compute clusters. Our prototype supports multiple analytic frameworks (Pig/Hadoop and Spark), and we show that the required changes are modest. We have implemented three algorithms in Kariz for optimizing the caching of individual queries (one from the literature, and two novel to our platform) and three policies for optimizing across queries from, potentially, multiple different clusters. With an algorithm that fully exploits the rich information available from Kariz, we demonstrate major speedups (as much as 3×) for TPC-H and TPC-DS.

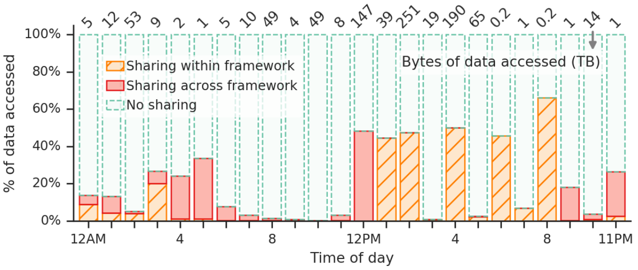


Figure 1: *The value at the top shows the total amount of data accesses in each hour. Green is the proportion of the accesses to data that was not accessed previously in the hour. Orange is the accesses that go to data previously accessed in the same hour by the same analytic framework. Red is accesses to data that was last accessed by another framework in that hour. This analysis suggests that caching can be effective both to capture repeated access to the same data from a framework, and access by different frameworks to the same data.*

1 Introduction

Large-scale data-flow oriented analytic frameworks, such as Spark [72], Hive [62], and Pig [56], are broadly used in many public and private cloud environments. Today, cloud deployments commonly use centralized "data lakes" [3, 9, 10, 42, 58] such as Amazon S3 [4], Azure Data Lake Store [11], and Ceph [67] that are used by all the frameworks running in the cloud. Although such dis-aggregation of storage offers many benefits, it also carries major performance costs [61].

Caching and prefetching, which move frequently-used datasets close to the analytic frameworks, are standard techniques for improving performance [20, 50]. Data-flow oriented analytical frameworks share a number of features that provide the opportunity to explore caching strategies that differ from prior work on CPU, page-based, and variable-sized (e.g. web) caching:

- they expose the input objects and inter-job dependency with Directed Acyclic Graphs (data-flow DAGs), where complex DAGs providing a detailed view into future I/O behavior;
- units of data access and computation are large, taking many seconds to access or run, allowing complex strategies not feasible in many other caching domains; and
- recurring jobs, where the same code runs on different data, are common [19, 24, 48], allowing accurate prediction of execution timing and characteristics [21, 40, 46, 66],

To illustrate these features, we consider confidential traces shared with us by an industrial partner recording 4 months of usage from a mid-sized (>100 nodes) cluster in production use which is running Hive/Hadoop, Spark, native MapReduce, and streaming jobs. Although the majority of jobs were small (1-4 node DAGs), 90% of input data was read by complex DAG-based jobs with 5 nodes or more, with some DAGs involving over 50 nodes. Individual DAG stages averaged 5 minutes with some stages taking as 6 hours. Over 90% of the jobs seen over the four months were jobs that repeated many times, and more than 90% of object reads were to objects repeatedly read. Clearly, there is an enormous opportunity to optimize performance for repeating I/O intensive jobs that provide complete visibility in future accesses and where minutes are available to compute strategies. In fact, a number of groups have started exploiting these characteristics to develop more sophisticated caching and prefetching strategies within analytics clusters [17, 20, 37, 44, 57, 69, 71].

Kariz is a platform designed to enable different strategies for caching and prefetching at the storage level. It collects DAG information from analytic platforms and the execution time that stages of the DAG take to execute. It also maintains global information about what data is cached and resource availability (e.g. storage bandwidth). This information is used by Kariz to accurately (§3.3) predict stage run-time. We have implemented a number of strategies, including the previously published strategy MRD [57] (Most Reference Distance) that exploit DAGs to maximize cache hit rate, and two new strategies, we call CP (Critical Path) and CMR (Cache for Minimizing Runtime). CP exploits the stage run-time prediction to cache data in order to reduce the critical path of stages in the DAGs. CMR relies on support in Kariz for partial file caching, prefetch partial data from each stage of the DAG to improve performance beyond a single critical path.

The focus in Kariz for optimizing the critical path is not primarily to improve the latency. Analytic platforms like Spark typically reserve resources until the entire DAG has completed. By optimizing the critical path, the resources reserved for a DAG can be freed earlier, improving throughput; potentially huge savings for complicated DAGs where the critical path takes much longer than most of the work.

A fundamental difference between Kariz and previous work [57,69,71,72] is that Kariz implements a *community cache*, where multiple clusters, potentially running different analytic platforms, can concurrently share a single instance of Kariz. A community cache is based on the idea that the near past of a community of clusters data accesses may be a good predictor of the near future data access pattern of new jobs from a cluster.

For example, within some financial services companies (e.g. the employers of two of the authors) different groups often create individual clusters with varied frameworks accessing the same shared data-lake, with reasons for this fragmentation including regulatory issues, security, organizational structure, or preference. When new datasets arrive (e.g. recent market data), many jobs are submitted independently by members of different groups, resulting in heavy access across most or all frameworks and clusters. Anecdotal reports (e.g. as described by the authors of Quiver [48]) indicate similar behavior by data scientists, who often use company-wide datasets for joins or training ML models. Finally, we see similar behavior in the confidential traces, e.g. in Figure 1 at 3AM the 692 jobs are distributed across frameworks (11% Spark, 20% Oozie, 16% MapReduce, 47% Hive/MapReduce); 36% of Spark jobs share objects (42% of total data) with Hive, while tables accessed by Oozie were a subset of Hive accesses.

In contrast to previous works [57,69,71], that focused on optimizing individual queries, Kariz also supports strategies for optimizing multiple concurrent queries coming from potentially different analytic platforms to the community cache. We have implemented two such strategies: 1) Shortest Job First (SJF) that focuses cache resources to accelerate the fastest predicted query to free resources as quickly as possible. 2) Cache for Minimizing Runtime of multiple DAGs (CMR-M), that additionally takes into account storage bandwidth and the sharing of data across multiple queries in selecting data to be cached and prefetched.

We have adapted two analytic platforms to exploit Kariz (PIG/Hadoop and SPARK) and found that while significant effort was required to understand the platform, the end changes were less than 100 LOC in each platform.

We experimentally evaluated our system on a 16-node bare-metal cluster. We use the characteristics of the confidential trace (including query submission rate, DAG structure, data accessed, data reuse) to run a mix of synthetic workloads from TPCH and TPC-DS. We demonstrate experimentally that the new algorithms enabled by the rich information collected by Kariz (and its support for partial caching) result in major performance advantages with our synthetic workload on both Pig/Hadoop (mean across all queries of 1.25x and maximum 2x) and Spark (mean 1.8x and 3x). Through simulation, we show that there are significant advantages to a community cache; e.g., an improvement of up to 1.5x over separate per-analytics platform caches.

Key contributions of this work are: 1) demonstrating the value of partial over than full file caching, 2) a high-accuracy run-time prediction based on the amount of cached state and available storage bandwidth, 3) demonstrate that it is possible, and in fact simple, to extract the information needed for optimizing performance from multiple analytics platforms, 4) showing that strategies can be developed and effective that target the direct improvement of predicted run-time by optimizing critical paths rather than implicit characteristics such as hit rate, and finally, 5) a new architecture that demonstrates that a community caching layer for analytics platforms is feasible and offers value.

2 Background and Related Work

We begin by describing the data analytics frameworks targeted by Kariz in further detail, providing an example to motivate our approach, and then briefly survey related work.

DAG-based frameworks: One way in which frameworks such as Spark [72], Hive [62], Impala [47] and Pig [56] differ from traditional large-scale applications is in their use of Directed Acyclic Graphs (DAGs) of operations (vertices) and their input/output dependencies (edges). Before a user query is executed, a query planner parses it, generating an unoptimized logical plan with resolved tables and columns. The optimizer performs optimizations such as predicate pushdown, prunes columns and partitions, and may even remove data from the logical plan. The planner transforms the optimized logical plan to a corresponding physical plan—e.g. the logical operation *TableScan* is transformed to *JsonScan* or *ParquetScan*, and byte ranges within input objects are assigned to each physical operator before the physical execution plan is sent to the execution engine. It is this physical execution plan which Kariz uses for intelligent prefetching and caching.

Definitions: As there are differences in terminology used by each framework, in this paper we use the following terms: A *task*, t, is the smallest unit of computation; tasks are scheduled by the lower-level framework scheduler (e.g. Yarn [65]). A stage, s, is a set of parallel tasks that execute the same code and are submitted for execution simultaneously (e.g. mappers); a higher-level scheduler (e.g. Spark's DAG-scheduler) decides when to submit a stage of tasks to the lower-level framework scheduler. Data-flow oriented frameworks have different terms for a set of stages linked by dependencies; we will follow common usage and use the term *DAG* for them. For each stage there is a set I_s of input objects and their byte ranges for that stage. Finally, some data-flow oriented frameworks (e.g. Pig [56]) divide *DAGs* into *stage-sets*, sets of stages in the DAG without inter-dependencies, which may run in parallel but must complete before the next stage starts.

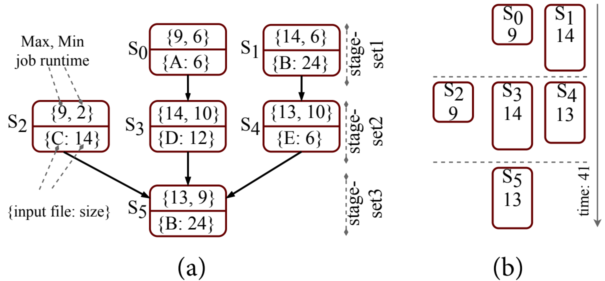


Figure 2: *(a) 6-stage DAG divided into 3 stage-sets. Each stage is labeled with run-time (max=no input in cache, min=all input in cache), input file name and size. (b) Execution schedule with no prefetching or caching for gang-scheduling.*

Disaggregated Storage: These frameworks typically default to using HDFS [61], which stores data locally on compute nodes in a cluster. Yet in modern practice, storage is often disaggregated from the systems performing analysis [42,52,54,58]. Data is typically stored on remote object stores such as S3 [4], Azure Blob Store [23], or on-premises equivalents such as Ceph [8], and accessed directly via e.g. the S3A connector [5]. This enables, for example, elastic analytics clusters [3] and serverless data analytics [59], but with a performance penalty, e.g. in 2018 the NetCo researchers [42] measured object storage speeds of less than 50MB/s per VM (or 4MB/core) on widely-used cloud services.

Motivational example: Kariz is a caching and prefetching system to accelerate computation in these frameworks; we provide a simple example to give intuition about our approach and how it contrasts with previous approaches. Figure 2 shows a DAG made up of 6 stages divided into three stages-sets (sets of stages in the DAG without inter-dependencies, which may run in parallel but must complete before the next stage starts), with the stages executing from the top to bottom. For each stage, the DAG specifies the input files and size; we have labeled each stage with its input file (A/B/...) and size, and run-time with (a) no data is in the cache, and (b) all inputs are cached or prefetched by Kariz. We assume Kariz can predict stage run-time from prior execution times, and that it can perform fine-grained *partial caching* with proportional speedup.

In stage-set 1 (top), we can see that there is no value in prefetching input B for S_0 (uncached duration 9) until we have addressed S_1 (duration 14), which determines the stage completion time. If we cache all of the input to S_1, however (see Figure 2a), reducing its run-time to 6, some of that cache space (and remote bandwidth) will be wasted, as S_0 will now determine stage-set completion time. Kariz instead caches "just enough" of each input to minimize stage-set run-time—e.g. caching 15 units of B will bring S_1 duration down to 9, and additional prefetching will be applied to both S_0 and S_1.

2.1 Related work

We focus on recent work on *informed* (rather than *history-based*) caching and prefetching for analytics frameworks, and omit the vast literature on disk/file [32,33,38,39,60], web [18], and CPU/memory caching [29,30,34,64].

Caching and prefetching rely on the existence of patterns and correlations in real workloads; Jockey [31] and Corral [41] show that the data access patterns of analytics frameworks are highly repetitive and predictable. In Ernest [66] we see that in real world deployments job run-times are predictable as well, based on factors such as input size and jobs DAG structure.

Unlike Kariz, MC^2 [70] and CD-LDS [27] are targeted to caching/prefetching (of files and memory respectively) for general applications, using OS and compiler hints rather than the job DAG available in our more specific scenario. Unlike Kariz which extracts the exact I/O access patterns from the analytic applications, Quiver [48] builds a deep learning specific cache layer and exploits the predictability of accesses in these applications for cache management. MRD [57] makes prefetch and eviction decisions based on graph distance, with the goal of maximizing cache hit rate; accesses at the next stage in the graph are prioritized over those farther in the future. LRC [71] does not prefetch, but makes eviction decisions based on *reference count*, i.e. the number of references to input in stages not yet executed. Dagon [69] ties caching with the DAG scheduler, using the stage scheduling priorities to evict/prefetch data. MemTune [68] manages RAM-based caching in Spark, evicting/prefetching data using only information from the currently runnable tasks.

Alluxio [2] (based on Tachyon [50]) and Apache Ignite [7] are widely used for caching in analytics frameworks; the caching component of Kariz is similar to these, although with extensions for partial caching of objects. A number of replacement and prefetching policies (as opposed to systems) specific to analysis frameworks have been developed, as well: PacMan [20] attempts to minimize run-time by considering MapReduce job wave widths; while NetCo [42] and MRD implement approximations to Belady's MIN algorithm based on predicted job execution order.

Kariz differs from these prior works in several ways: (1) it makes use of *partial caching*, which gives significant gains when the cache size is not large compared to input object sizes, and (2) it explicitly tries to minimize predicted DAG completion time, rather than e.g. cache hit rate.

3 Kariz design

Kariz is a cache management and prefetching system that controls admission/eviction to/from a storage cache, and calculates prefetching schedules for data-flow oriented frameworks. We show the Kariz architecture (§3.1), its partial caching (§3.2), runtime estimation (§3.3) and availability and scalability strategies(§3.4).

3.1 Architecture

As shown in Figure 3, Kariz implements a *community cache*, where multiple clusters, potentially running different analytic platforms, can concurrently share a single cache. Kariz interfaces with the frameworks to obtain data-flow DAGs, execution states, and scheduling events, and collects historical

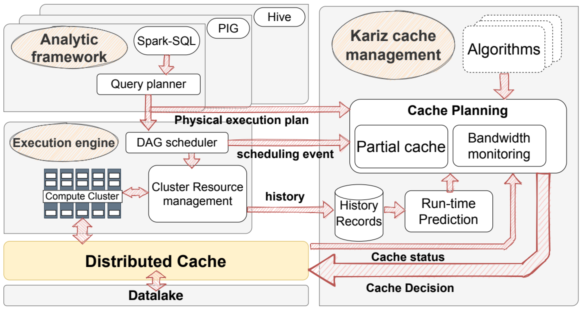


Figure 3: *Kariz architecture. The Kariz components extract DAGS, scheduler and run-time information from the analytic framework. The run-time predictor uses historical run-time information to predict stage execution time. The Kariz cache management algorithms generate caching/prefetching plans for DAGs by exploiting the predicted execution time, partial caching, and data sharing across DAGs. The cache planning makes caching, prefetching, and eviction decisions based on the generated plans to minimize the DAGs runtime while taking account of data sharing for efficiency.*

information (logs) to use in predicting the run-time of future jobs. The cache controller maintains information about what data is cached and estimates bandwidth available to the storage cluster. The run-time prediction component estimates stage completion time based on prior runtimes and current state. The algorithms make up the core of the system, where the task of a single-DAG planner is to come up with a plan for an individual DAG that is then combined into an overall plan by the multi-DAG planner, which issues cache, prefetch, and eviction commands to the cache controller and cache.

Interaction with analytics framework: The framework interface notifies Kariz of DAG submission, providing the physical query execution plan detailing input objects, sizes, formats, operation (e.g. map, filter, join), and parallel task count. Kariz is also notified when a stage-set (Pig) or stage (Spark) begins or finishes execution. In addition, Kariz needs job (DAG) history information (i.e. logs) for prediction; typically this is available through existing framework interfaces.

Storage bandwidth: Prefetching is constrained not only by cache capacity but also by the effective bandwidth from back-end storage, limited by the speed of the network or the storage system itself. Kariz schedules operations to fit within this bandwidth, which is currently configured based on measurements but could be estimated dynamically as well.

Planners: In Kariz, scheduling of cache capacity and storage bandwidth is performed in two stages. The single DAG planner examines individual DAGs and stages to determine *caching candidates*—sets of data from one or more objects which can be prefetched or retained in the cache to speed DAG execution. The multi-DAG planner, in turn, determines which of these caching candidates to put into effect within constraints of cache size and backend bandwidth, prioritizing completion time within a DAG and throughput across DAGs, taking account of data sharing for efficiency.

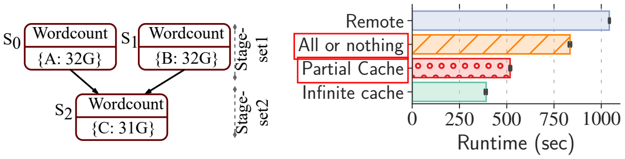


Figure 4: *Physical experiment to illustrate partial caching: DAG with 3 wordcount jobs with 32 GiB inputs and 63 GiB total cache capacity.*

Cache control: The cache controller provides an abstract cache interface to the Kariz planner. It is responsible for tracking currently cached data, and managing the prefetching, retention, and eviction processes; as a distributed cache scales [44] this component will be replicated. Its interface to the cache has methods to prefetch data on a fine-grained basis, "pin" it in cache or release it, and enumerate cache contents (e.g. on startup).

3.2 Partial caching

Hadoop and Spark are sensitive to *stragglers* [49], longer-running tasks which delay completion of a computational stage. Prior prefetching work [20, 42, 57, 71] assumes partial caching of stage inputs will lead to such stragglers, giving no benefit. This will occur when done naively, as some tasks will find their entire input in the cache, while others will fetch their full input from remote storage. We instead assume fine-grained control over prefetching and cache retention/eviction, allowing data to be cached in *strides* much smaller than the input to a single task.

We see the utility of partial caching for real workloads with limited caching in Figure 4. We define an artificial DAG with three wordcount Mapreduce stages, dependent on the other two; each with 32 GiB input. With all-or-nothing caching we can only cache the input to S_2, minimizing stage 2 runtime, but cannot speed up stage 1. With partial caching we still cache the entire input to S_2, yielding the highest runtime reduction per unit of caching, but can distribute the remaining cache across S_0 and S_1, reducing runtime closer to the fully-cached minimum.

Kariz architecture explicity supports column-oriented formats like Parquet [51] and Arrow [6]. It relies on physical query plans to identify object ranges, rather than entire objects; prefetch decisions would then be made within these ranges.

3.3 Run-time prediction

Recent studies from in-production clusters at Microsoft (e.g. Graphene [36], NetCo [42], and others [24, 28, 40, 43]) show that examined jobs are recurring—similar computations are repeatedly executed on different datasets. The same studies show that tasks extensively share common operands, and that most user-defined operations are not custom programs, but widely-used shared libraries (Cloudview Figure-(3) and Figure-4(a & d) [43]). Recent studies, such as Ernest [66], CherryPick [19], and Selecta [45] have shown good accuracy when predicting run-time for such recurring workloads, as a function of input file size, size of cluster and DAG structure [66].

In Kariz, we extend the runtime prediction model proposed by Ernest to incorporate caching and bandwidth differences between cache and remote storage. Similarly to Ernest, we assume that computation time scales linearly with the input size [66] and the communication patterns among stages of a DAG could be represented as sequential, aggregate, and shuffle operations. Unlike Ernest that assumes that builds a runtime prediction model for the entire DAG, Kariz predicts the performance of stages according to the operands executed on that stage and the communication pattern with the previous stage.

We predict stage time T given total input size S, with $f \cdot S$ in cache, bandwidth r_s and r_c to storage bandwidth and cache bandwidth, T tasks (e.g. mappers)[1] and N executors, fitting the following equation:

$$T = \theta_0 + \theta_1 \frac{(1-f)S}{r_s} + \theta_2 \frac{fS}{r_c} + \theta_3 \frac{T}{N} + \theta_4 \log(N) + \theta_5 N \qquad (1)$$

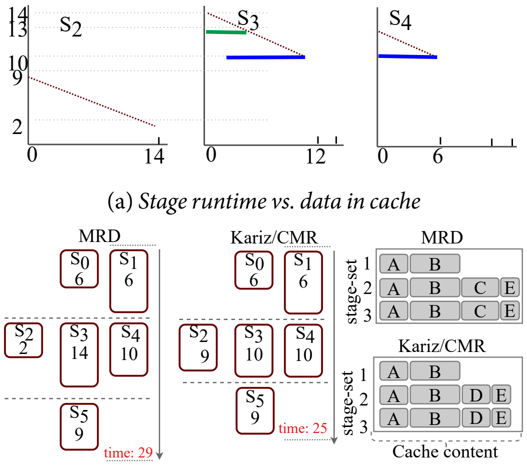
where terms represent fixed startup time (θ_0), data fetch from backend storage and cache (θ_1, θ_2); following Ernest we also incorporate terms for sequential (θ_3), aggregate (θ_4), and shuffle θ_5) cross-stage communication. We use Lasso regression [63] with non-negative coefficients and cross-validation to be resilient to overfitting when training on limited data.

At each scheduling event, Kariz identifies the available backend storage[2] and cache, iterates over the future stages to find longest and "slack" paths. Kariz does this by predicting the runtime of each stage in two cases—data in cache and data needing to be fetched from the remote and uses Bellman-Ford [22] with negative weights to identify the order of the longest paths. We discuss the accuracy of this model further in Section 6.3.

3.4 Availability and Scaling

Kariz takes a simple approach to availability, based on the principle that prefetching and sophisticated cache control are *optional*—DAG-based frameworks and associated caches (e.g. Alluxio) are widely used today with no prefetching or cross-DAG scheduling. If Kariz crashes, the cluster continues to operate, and the caches fall back to LRU after the current commands have completed; on restart Kariz can fetch all needed states (DAG queue, cache contents, execution history) from other components and resume.

Most computation in Kariz occurs in the cache controller, which is responsible for block-level caching and prefetching commands; this scales by adding additional controllers, each responsible for some set of caches. The central planning algorithm is not currently scalable, and the cluster size which can be controlled by a single Kariz instance is limited by its speed. In §6.8, we show that current unoptimized performance should scale to a cluster of thousands of nodes.



(a) *Stage runtime vs. data in cache*

(b) *Dag schedule and Cache status*

Figure 5: *(a) Job runtime vs. data in cache at job start for DAG stage 2 (S_2, S_3, S_4). Green represents caching needed to reduce S_3 runtime to that of S_4; blue is caching to reduce S_3, S_4 to their minimum runtime. Caching for S_2 will never reduce stage runtime, (b) With cache size 50, Kariz/CMR finishes before MRD (25 vs 29) despite a lower hit rate (83% vs 86%)*

4 Planners

We implement three planners in Kariz for scheduling caching and prefetching: MRD [57], Critical Path (CP) [17], and our new algorithm, Caching for Minimizing Runtime (CMR).

MRD (Most Reference Distance) is based on topological distance, i.e. the number of DAG stages between two accesses to a file, evicting data with the longest distance until future reference, and prefetching data with the shortest distance. CP, described in an earlier workshop paper, prioritizes prefetching and caching for jobs on the DAG critical path. We refer readers to the respective publications for a more detailed description.

CMR considers the analytic framework DAG scheduling schema and makes full use of the runtime estimation, partial caching, and bandwidth measurement features provided by Kariz. To minimize DAG runtime, it jointly schedules cache space and backend storage bandwidth. When multiple DAGs are active it divides cache space and prefetching opportunities across DAGs using a heuristic that attempts to maximize the throughput—i.e. prioritizing shared data which will speed up multiple DAGs.

4.1 CMR Overview

To explain the intuition behind CMR, we again use the 6-stage DAG from Algorithm 2, scheduled with gang scheduling, examining stage-set 2 (stages 2, 3, and 4) in more detail. In Figure 5a, we assume a graph of runtime vs. amount of input in the cache for these stages. For the sake of simplicity of the discussion, it

[1] This is predicted by the analytic framework during query planning

[2] Currently, Kariz splits bandwidth equally between all running DAGs in a cluster.

shows a piece-wise linear function for stage time as a function of prefetched/retained data. In reality, we use runtime prediction (Section 3.3) to predict the required cache size to achieve certain improvement. In Figure 5a, we see S_3 runtime decreases from 14 to 10 as its 12 units of data are cached; S_4 from 13 to 10 with 6 units of caching, and S_2 from 9 to 2 with 14 units of caching.

In making prefetch/retention decisions CMR examines *candidate caching sets*—sets of input objects and their ranges which speed up one or more stages in a stage-set. The first candidate here, shown as a green horizontal bar, represents the fraction of input to S_3 needed to reduce its runtime to that of S_4. The next candidate, shown in blue, corresponds to the remaining input to S_3 and S_4, reducing their runtime to a minimum. Note that the input to S_2 is not part of any candidate set, as S_2 will always complete before S_3 and S_4 and never affect stage-set completion time.

The core of CMR consists of enumerating these candidate sets and pick them in decreasing order of runtime improvement and prefetching or retaining all sets which fit within cache size and bandwidth constraints. We see CMR compared to MRD in Figure 5b, with a cache size of 50. Although MRD achieves a higher hit rate (86%) than CMR (83%), CMR's achieves a lower runtime (25 vs 29) by ignoring stages like S_2 with "slack" in their schedule and focusing on only ones determining stage-set runtimes.

In our simplified example, there is no need to compare caching candidates against each other—once we decide not to cache input to S_2, there is enough cache space for all remaining input. With fewer resources, however, we must choose between e.g. retaining data for one future stage vs. prefetching for a different one.

We do this based on *marginal utility*, i.e. the ratio of completion time saved by caching a candidate set to its size. This is similar to the fractional knapsack problem, i.e. achieving maximum reduction of run-time given a fixed cache capacity, hence the use of cost:benefit in comparisons. This allows comparing candidates across DAG stages, for example, to determine whether cache space and storage system bandwidth in stage-set 1 would be better spent prefetching for stages in stage-set 2 or stage-set 3.

4.2 CMR

The CMR planner runs upon receiving the stage-set scheduling events from the analytic framework, identifying the prefetching and cache pinning/unpinning operations to be executed *during* that stage for execution in *following* stages; prefetching is scheduled so that it will complete by the beginning of the stage in which the data will be used. As with prior work, we assume the existence of a "stage 0" before the DAG begins execution; in practice, this would correspond to the last stage of the previously-executed DAG. We describe CMR operation in the case of a single DAG in two parts: enumeration of caching candidates, in Algorithm 1, and candidate selection and execution, in Algorithm 2.

Algorithm 1 Caching candidate set enumeration

```
Input:
    $T_1, T_2, \ldots$        no-cache job completion times, longest first
    $\alpha_1, \alpha_2, \ldots$        per-job time improvement per unit cached
    $I_1, I_2, \ldots$        job inputs

Output:
    $c_1, c_2, \ldots$        caching candidates
    $c_i$ specifies data to be cached from $I_1 \ldots I_i$, and has value
    (i.e. time saved) = $T_i - T_{i+1}$

procedure CANDIDATES(stage $i$)
    $T_{min} = T_1 - \alpha_1 \cdot |I_1|$
    $t = (T_1 - T_2),\ c_1 = \{I_1 : \frac{t}{\alpha_1}\}$
    $t = (T_2 - T_3),\ c_2 = \{I_1 : \frac{t}{\alpha_1}, I_2 : \frac{t}{\alpha_2}\}$
    etc. while $T > T_{min}$
end procedure
```

The candidate enumeration algorithm in Algorithm 1 examines stages from longest to shortest within a stage-set, enumerating candidates in decreasing order of benefit (completion time saved) to cost (size). The first candidate will be from the input to the longest stage, of sufficient size to reduce its runtime to that of the next-longest—i.e. the green segment from Figure 5a. The second candidate in Figure 5a corresponds to the blue segments, reducing the runtime of S_3 and S_4. Candidate enumeration stops when no more candidates can be enumerated, e.g. in this case where S_3 and S_4 runtimes are reduced to their minimum.

Candidate selection is performed by Algorithm 2; we describe this first for the case of a single active DAG, before discussing its operation across multiple DAGs.

After enumerating caching candidates for all future stage-sets in decreasing benefit:cost order, we compute the "slack" back-end storage bandwidth available in each stage-set based on the current estimated stage-set completion time and measured storage access rate. Candidates which are "too early" are eliminated; these are ones that may be safely deferred to a later stage-set and still complete by the time of the stage-set in which they are needed.

We then consider the remaining candidates—if a candidate "fits" into the remaining cache space and bandwidth, we schedule the prefetching operation (if needed) and "pin" the candidate in cache until the end of the stage in which it is needed. In the next step, CMR updates available cache space and slack bandwidth (prefetch bandwidth), as well as adjusting stage completion time estimations to account for the speedup. If we run out of prefetch bandwidth before cache space (omitted for clarity in Algorithm 2), we continue examining in-cache candidates until we run out of cache space.

This strategy not only calculates a set of data to prefetch but implicitly calculates evictions as well. Data currently in the cache which is valuable for reducing the runtime of a later DAG stage will be part of one of the selected candidate sets, and will be pinned through the end of its scheduled use; the remaining cache contents are unpinned and may be evicted

Algorithm 2 Cache candidate selection: schedule prefetching/pinning for later stages

```
Initial Conditions:
    $t_0 = 0, t_i = t_{i-1} + max(T_{i,*})$            ▷ Estimated stage completion times
    $fetch_i = sum(|I_{i,*}|)$                        ▷ committed bandwidth by stage

Input:
    candidate lists for each active DAG

Output:
    prefetch, pin, and unpin operations

procedure PLAN($lists$)
    while slack$_{bw}$ and cache available do
        $lists \leftarrow sort\left( \frac{T_{list}}{\sum_{b \in frags(list)} \left(\frac{1}{n_b}\right)} \textbf{ for } list \textbf{ in } lists \right)$    ▷
$n_b : N_{dags}$ shared b.
        link $l_1$ and $l_2$ if $l_1$ share blocks with $l_2$ for each $l_1$ and $l_2$ in $lists$
        slack$_{bw}(j) \leftarrow r \cdot (t_j - t_{j-1})$          ▷ bandwidth slack in stage $j$
        $c = best(head(list)$ for $list$ in $lists)$        ▷ best candidate
        skip c if stage(c)>now and c fits in slack$_{bw}$(future)
        $s \leftarrow stage(c)$
        $fetch_s \leftarrow fetch_s - |c|$                  ▷ $|c|$ no longer demand-fetched
        if $c$ not in cache then
            $fetch_{now} \leftarrow fetch_{now} + |c|$
            prefetch($c$)
        end if
        adjust $t_s$,... for $c$ speedup
        pin $c$ until end of $s$
        cache used += $|c|$
    end while
end procedure
```

(e.g. in LRU order) if necessary to make room for new data.

Event Scheduling: in most analytic frameworks that implement the event-based scheduling schema, e.g. Spark, the root stages of the DAG (those with no-dependency) are responsible for fetching DAG input data. In this case, CMR, for each root stage, predicts the longest path to the leaf stages of the DAG (those with that produce output). Then, it sorts these paths according to the predicted run-time and recursively identifies the cache candidates for them.

4.3 Multi-DAG Scheduling

Next, we describe the algorithm that prefetches for multiple DAGs simultaneously, *Cache for Minimizing Runtime of Multiple DAGs* (CMR-M). It attempts to maximize throughput by minimizing the time to completion of sequences of DAGs.

CMR-M assumes the use of *static partitioning* (default setup) in Spark and Pig, where resources are allocated to a DAG for the entire period of execution [14].

We enumerate caching candidates using Algorithm 1, then we choose to execute candidates in Algorithm 2, as before; selecting the "best" cache candidates from competing per-DAG lists via a heuristic score for data sharing, *sharing-aware weight*. The *sharing-aware weight* is calculated per cache candidate and gives preference to candidates that are shared by other running DAGs, as the throughput increase from prefetching will be higher than that indicated by the single-DAG value.

The *sharing-aware weight* is similar to the resident set size (RSS) calculation [35] from 'ps': data shared by multiple DAGs is split equally among them before calculating cost. We calculate the *sharing-aware weight* on a block-per-block basis, counting the number of DAGs n_b sharing any block b (Equation 2). We then calculate a "unique file size", U_c, counting each block in C shared between n_b DAGs as having size $\frac{1}{n_b}$ (line 12 – 14 in Algorithm 1). We then re-compute the marginal utility using this weight and use this utility to compare candidates across DAGs. Finally, we find the "partners" to the selected candidate, i.e. those sharing blocks with it, and select those for prefetching/caching as well.

$$U_s = \sum_{b \in frags(C)} \left(\frac{1}{n_b} \right) \tag{2}$$

5 Implementation

The Kariz implementation combines a Kariz service with a our previously developed [44] caching layer embedded within the Ceph Rados Gateway (RGW [8]). We modified this caching layer for Kariz by integrating fine-grained prefetching and pinning (~100 C++ LOC). We have also modified both Pig/Hadoop [56] (~100 Java LOC) and Spark [72] (30 Scala LOC) to work with Kariz. We discuss each of these components in turn.

5.1 Kariz Service

Our Kariz prototype is about 5000 lines of Python[3], including the runtime predictor, the MRD, CP and CMR DAG planners, and the CMR-M multi-DAG planner. We use the `sklearn` package for runtime prediction, and `graph-tool` for graph traversal. The Kariz service also includes a cache coordinator that translates high-level operations from the planner into individual block operations on the cache.

Interfaces between the analytic frameworks and Kariz are listed in Table 1. The *newDAG*, *stageStart*, and *completeDAG* notifications from the framework trigger DAG planning activities, and carry information (e.g. annotated DAGs) needed for planning. The *prefetch*, *pin*, and *unpin* requests to the cache controller, in turn, translates high-level requests from the planner (specifying object and stride) into requests to the cache to fetch, pin/unpin, or evict individual blocks.

5.2 Caching layer

We build Kariz by extending the multi-tenant cooperative caching architecture recently added to RGW [44] This RGW cache layer expands to multiple clusters and allows different frameworks such as Hadoop MapReduce [26] and Apache Spark [72] to cache and share their inputs. Our extension to support Kariz involved around 100 C++ LOC. This involved

[3] http://github.com/maniaabdi/Kariz

Table 1: Interface from Analytics framework (e.g. Pig or Spark) to Kariz and between Kariz and the Cache

	API	Description
→ Kariz	`newDAG(ID, DAG)`	new DAG started
	`stageStart(ID, stage)`	Jobs in `stage` scheduled for execution
	`completeDAG(ID)`	DAG completed.
→ Cache	`prefetch(blocks)`	Asynchronously fetch blocks into cache
	`pin(blocks)`	Lock blocks in cache.
	`unpin(blocks)`	Release blocks to be replaced as space is needed

Table 2: Hardware configuration

	Compute Server	Cache Server
CPU	1x Intel E5-2650	2x Intel E5-2699v3
RAM	128 GB	128 GB
Disk	1x 500 GB HDDs 5400 RPM	2x Intel P3600 1.6 TB NVMe SSDs (RAID0)
Network	10Gb/s	40Gb/s

Table 3: Software configuration

	Hadoop	Pig	Spark
Version	2.8.4	0.17.0	2.4.5

adding the operations to prefetch pin and unpin lists of 4MB chunks of datasets, as depicted in Table 1. The small changes required is evidence that we will be able to integrate Kariz with other caching services. Kariz could integrate with other distributed caching systems such as Alluxio [2, 50] and coordinates their caches. In the case of Alluxio minor modifications are needed to support partial prefetching.

5.3 Analytical frameworks modifications

To exploit Kariz, an analytic framework must provide some interface that Kariz can use to extract run time interface and notify Kariz of new DAGs, the start of stages, and DAG completion using the interface in Table 1. We have found it relatively easy to develop adaptors for two frameworks, Pig [56] and Spark [72]; suggesting that framework developers will find it easy to add the required functionality to take advantage of Kariz.

Pig modifications: Modifications to Pig are 100 Java LoC in the following functions: (1) *compile()* in MapReduceLauncher.java to extract the DAG, annotate it, and invoke *newDAG*. (2) *launchPig()* in MapReduceLauncher.java to extract the stage and invoke *stageStart*. (3) *dumpStats()* in MRPigStatsUtil.java to invoke *completeDAG*; Kariz then request detailed statistics from Hadoop history server.

Spark modifications: Modifications to Spark are 50 Scala LoC in the following functions: (1) the *constructor* in SQLExecutionRDD.scala, and *toRdd()* in QueryExecution.scala to annotate the RDD DAG, (2) *runJob()* in SparkContext.scala to extract DAG and invoke *newDAG*, where the ID is based on a UUID and spark application ID, (3) *submitStage()* in DagScheduler.scala to extract the stage and invoke *stageStart* (4) When the SparkContext shuts down, it invokes *completeDAG*; Kariz then request detailed statistics from Spark history server.

6 Evaluation

We use a combination of experimental evaluation on our prototype and simulation to evaluate Kariz. After describing the experimental infrastructure and simulator (§6.1), we **experimentally** demonstrate the value of partial caching (§6.2), examine the accuracy of our run-time prediction (§6.3), and then show results for the different DAG planners with both PIG and Spark (§6.4). The remainder of the evaluation uses **simulation** to evaluate the single DAG planners for a larger set of queries (§6.5), explore the multi-DAG planners for both queries from separate PIG and Spark clusters, and when Kariz is simultaneously used by both PIG and Spark clusters (§6.6) and finally perform sensitivity (§6.7) and scalability (§6.8) analysis.

6.1 Setup

Infrastructure: The physical experiments with Pig/Hadoop and Spark are run on a 16 node cluster with the hardware and software configuration in Table 2 and Table 3. We provisioned the compute nodes via diskless provisioning [53] and use the local disks of the compute nodes to deploy local HDFS. We use the NVMe SSDs of the cache servers to build the cache layer.

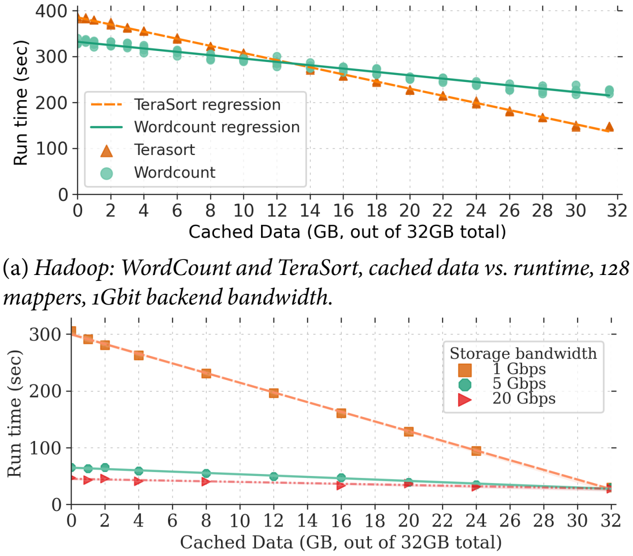
Simulator: We implement a simulated execution framework and cache, allowing additional experiments not possible on the physical cluster[4]. Execution time for each job to be simulated was determined by the run-time prediction model trained for each operation in Pig/Hadoop and Spark with different cache sizes. A random term was added to the runtime, with standard deviation taken from measured run-time. Additional simulator logic mimics the Kariz extensions to the framework scheduler, allowing the same Kariz code to be used in physical experiments and simulations.

6.2 Partial Caching

We evaluate a key premise of Kariz: that straggler-resistant partial caching can reduce runtimes. In Figure 6a, we see experimental results for `Wordcount` and `TeraSort` on a 16-node Hadoop cluster with 128 mappers and 32 GiB input files. In Figure 6b, we see the `Wordcount` benchmark on a 16-node Spark cluster with 64 GiB input files, and vary bandwidth to remote storage.

With both Hadoop and Spark we see a linear relationship between run time and cached data. While these are trivial applications, given that the platform partitions the data across the mappers, we believe this is good evidence that partial caching can be effective. In contrast, random choice of 4 MiB blocks was found to produce little or no speedup when less than 60% of the input was cached, and (as expected) caching a strict prefix of the file produced no improvement until the entire file was in cache.

[4] As well as timely reproduction after algorithm changes.



(a) *Hadoop: WordCount and TeraSort, cached data vs. runtime, 128 mappers, 1Gbit backend bandwidth.*

(b) *Spark: cached data vs. runtime for WordCount and different backend bandwidth on 32 GiB input, 128 partitions.*

Figure 6: *Partial caching*

We also see that the slope varies with storage bandwidth and application, motivating our design to build per-operation runtime prediction model that incorporates storage bandwidth.

6.3 Runtime prediction

Training model: to train the runtime prediction models, we ran Lasso regression with $\alpha = 0.001$ on each category. On our test cluster, we run all 44 queries from Pig-TPCH [13] and Spark-TPCH [16], 13 times each, with different configuration: (1) input datasets randomly selected from 8GB to 80GB, (2) number of cached blocks randomly selected from 0% to 100%, (3) the bandwidth to backend storage restricted randomly from 1Gbps to 40GBps, and (4) the number of executors per query was configured randomly from the 2, 4, 8, 16. In total, we captured statistics for 286 queries for Pig/Hadoop and 286 queries for Spark. We use the 80%/20% split to train and test each model. Table 4 shows the average run time of the test set, the root mean square error (RMSE) of the runtime prediction model per operation and average absolute error.

Prediction accuracy: The caching and prefetching planners depend on the Kariz runtime predictor being accurate enough to predict the correct paths (longest, the 2^{nd}-longest, etc.) to cache for. We ran the 22 queries from the Spark TPCH benchmark, with random input sizes, on a 16-node cluster with no caching. We predict the run-time using the trained models: for 20 queries out of 22 the order for 1^{st}, the 2^{nd}, and 3^{rd} longest path were identified correctly. For query Q11, it mis-predicts the 1^{st} longest path, and for query Q21, the order of the 2^{nd} and 3^{nd} longest paths were reversed. In these two cases, the errors were in paths differing by less than 3s; in almost all cases either both or neither would be cached, and the misprediction impact would be minor.

In Figure 7, we see the ratio of the actual to predicted longest path. The maximum error was 27% and on average 7%. We annotate each bar with the bandwidth, the input size, and the actual runtime of the longest path for that query. For Q6, where we had the maximum relative error, the actual runtime was short.

6.4 Experimental evaluation

We compare CMR with two DAG-informed policies: (1) MRD [57], which caches and prefetches in breadth-first order to increase hit ratio, and (2) our CP [17] which caches and prefetches for jobs on the DAG critical path.

Workloads: We use the characteristics of the confidential trace (including query submission rate, DAG structure, data accessed, data reuse) to construct a mix of synthetic workload using standard analytic benchmarks (TPC-H and TPC-DS) because of the lack of public workloads. The synthetic benchmark represents an hour of data processing. We scale down the size of our cluster and the DAG submission rate by a factor of 10; in the confidential traces, the job submission rate follows the Poisson distribution with distribution parameter(λ) of 0.2 (on average

Table 4: Accuracy of runtime prediction per operation

(a) Pig operations

Operation	$\overline{runtime}$ (s)	*RMSE* (s)	$\overline{Absolute}$ (s)
Co-group	330	63.27	61.14
Map only	5.8	0.33	0.26
Groupby	14.6	1.8	1.6
Combiner	23	8.3	2.8
Hash join	99.3	37	25
Replicated join	251	38.5	55
Order by	10.6	0.39	0.49
Sampler	10.74	0.58	0.54

(b) Spark operations

Operation	$\overline{runtime}$ (s)	*RMSE* (s)	$\overline{Absolute}$ (s)
Hash aggregate (HA)	1.3	0.66	0.52
Scan	12	3.3	2
Scan, Filter	20.2	6.28	3.27
Scan, Filter & HA	30.6	5.2	3.97
Filter & HA	2	0.66	0.63
Sort merge join	3.51	1.5	0.94
Sort merge join & HA	3.53	0.75	0.53

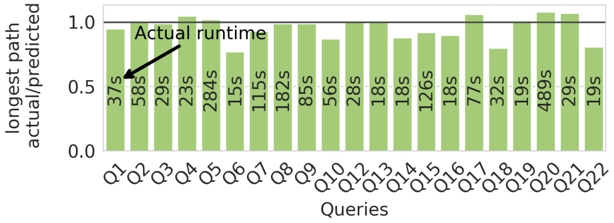


Figure 7: *Ratio of actual to predicted longest path on different Spark queries. See text for Q11.*

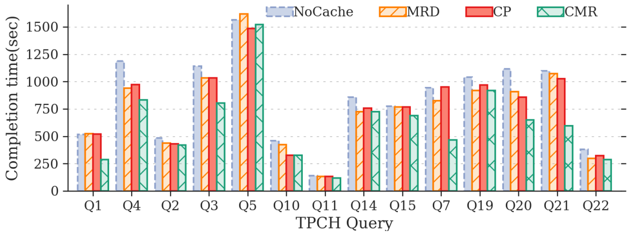


Figure 8: *Pig-MapReduce performance for selected TPC-H queries - small DAGs (1,4), long sequential (2,3,5), tree-like (10,11,14), aggregate (15), large/complex (7,19-22). 64 GiB data set, 40 GiB cache, cold start.*

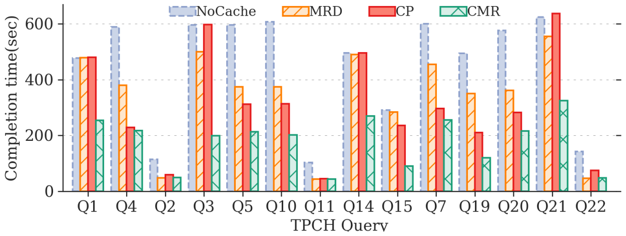


Figure 9: *TPC-H query performance using Spark contrasting CMR, CP, MRD and no caching.*

702 queries were submitted per hour). Thus, we use Poisson distribution with λ of 0.02 to generate query submission events (result in 67 queries). Then, at each event, a query is drawn from a pool of 22 TPC-H [13], and 19 TPC-DS [12] queries translated into Pig Latin [56]. To select queries from the pool, we categorize queries according to their DAG structure for a sampled hour. We map each query in the pool to a category with the maximum DAG similarity (maximum common subgraph [55]). To be consistent, for Spark, we use the same set of queries from Spark TPCH [16] and Spark TPC-DS benchmarks [15].

For the sampled hour, 30% of the objects were accessed at least twice. This is similar to the ZipF distribution with $a = 1.125$. Accordingly, we assign each query a dataset selected from this distribution, giving a trace of 496 accesses over 357 unique input objects, with 78 accesses to the most used input object, consistent with our evaluated traces. Then, we associate the input sizes to the datasets by randomly choosing from 4 GB, to 256 GB. Finally, we use the standard TPC-H [13] and TPC-DS [12] input generators to create CSV datasets.

The cache size is set to 40GB and the dataset size is 64GB. The RGW cache network to datalake is throttled to 10Gbps. For each query, we assume 5 seconds queue time, the minimum observed queue time in the evaluated traces, before submitting the first stage for execution to the execution engine. We take advantage of the queue time to start prefetching for the DAG and we clear the cache before each run. The reported numbers are the average of three runs.

Performance evaluation: Figure 8 and Figure 9 shows runtime of selected TPC-H queries in, respectively, Pig-Latin and Spark comparing CMR, CP, MRD, and no caching.

Our evaluations show that relative to the no caching case, CMR can improve query performance by up to 2 times for Pig-MapReduce and up to 3.2x on Spark, with mean speedups of 1.3x and 1.8x respectively. Running on the Pig-MapReduce framework and comparing to MRD and CP, CMR can improve the runtime by up to 2x and 1.8x and in average by 1.3x and 1.3x respectively. Our experiments using LRU shows similar behavior to no-prefetching (gray bar).

With the Spark framework, CMR can improve the runtime compared to MRD and CP by up to 3.1x and 3x and in average by 1.8x and 1.7x respectively. Spark shows more sensitivity to the backend storage bandwidth than MapReduce, Since MapReduce imposes extra overheads such as JVM start up([45, 46]). This results in sharper speed up slope for CMR on the Spark framework than the Pig-MapReduce framework and better cache space utilization compared to MRD and CP.

Q1 and Q4 represent small-sequential queries; with reads only at the begining of the job in the graph. Due to the partial caching strategy implemented by CMR, it has better performance compared to MRD, CP on both Pig/MapReduce and Spark frameworks. The table shows the average speedup:

Q1 and Q4	MRD	CP	no caching
Pig/MapReduce	1.5x	1.5x	1.6x
Spark	1.8x	1.4x	2.2x

For Q2, Q3, Q5, Q10, Q11, Q14, and Q15, the structure of DAGs generated by Pig and Spark is different, which leads to different caching decisions. On Pig-MapReduce, Q2, Q3, and Q5 are long sequential queries with small reads in the first stages and large reads in the following one. Here, the CMR ranking mechanism makes it possible to prioritize prefetching plans that have more effect on the sequential DAGs. Q10, Q11, and Q14 are long sequential graphs and Q15 is a large aggregated graph. For these, the combinations of stage oriented decisions and partial caching leads to performance improvement.

Q7 and Q19 to Q22 have large complicated DAGs on both Pig-MapReduce and Spark. The excellent relative performance of CMR over the other options for these queries (see table below) is encouraging, as our analysis of real-world traces in §1 showed over 90% of data read by complex queries like these.

Q7 and Q19-Q22		Pig/MapReduce	Spark
Average	MRD	1.4x	1.8x
	CP	1.5x	1.5x
	no caching	1.6x	2.8x
Maximum	MRD	1.8x	2.9x
	CP	2x	2x
	no caching	2x	3.2x

6.5 Simulated evaluation - Single DAGs

We evaluate CMR across the synthetic workload (67 TPC-H and TPC-DS queries) using our simulated cache and Pig framework. In Figure 10 we see CMR, CP, and MRD performance

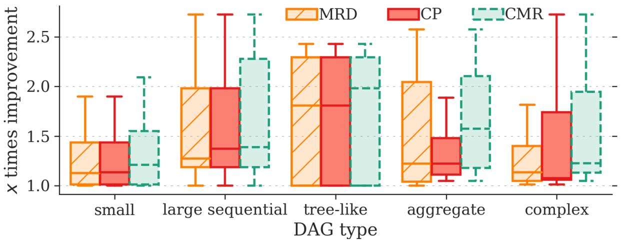


Figure 10: *Runtime by DAG class for 67 queries; 128 GiB cache, 10 Gb/s backend bandwidth.*

relative to no cache, with a simulated cache size of 128 *GiB*, grouped by query type.

Median (center line) and 75th-percentile (top of the box) performance are seen to be higher than MRD or CP in all cases. CMR performance is much higher (1.6x vs 1.2x for MRD and CP) for aggregation queries, although the top quartile of queries achieved relatively similar speedups for CMR and MRD. For complex queries CMR outperformed MRD by large amounts, with a 75th-percentile speedup higher than the maximum MRD speedup.

6.6 CMR Performance on Multiple DAGs

Kariz is a *community cache* that considers data sharing between DAGs running on one or several analytic frameworks. We simulate Kariz to measure the performance under two scenarios:

Multiple DAGs in a single analytic cluster: We compare the performance of three multi-DAG strategies: CMR-M, shortest-job-first (SJF) (prefetches/caches for DAGs with the shortest remaining runtime), and Isolated (static isolated cache partitioning for each DAG) [25]. In all three cases, CMR is used to manage within-DAG caching/prefetching decisions.

We simulate 1 TB of cache, 25 Gbit/s network bandwidth, and 100Gbit/s cache bandwidth. We generate a workload that consists of 200 randomly-chosen Spark TPC-H [16] queries with a dataset size of 164GB in a cluster that can handle 10 simultaneous queries. For the Isolated strategy, we allocate 128 GiB of cache space to each query. To produce different sharing patterns, we generate 6 traces of 200 datasets generated by changing the ZipF distribution parameter (a: 1.001-2.4)– e.g. a = 1.001 giving a trace with 192 unique dataset accesses. Finally, we map each dataset in every trace to one query.

In Figure 11, we see the end-to-end runtime of all 200 TPC-H Spark queries with 6 traces, when we increase the reuse/sharing of datasets within the trace. As seen, CMR-M outperforms both isolated cache and shortest job first by up to 1.51× ($a = 1.23$) and 1.14× ($a = 1.38$), respectively. As depicted, the SJF policy can degrade performance compared to isolated cache. The reason is SJF favors DAGs with smaller predicted runtime. This results in DAGs with longer runtime deprived of the cache and therefore to read most of their data from the backend. For the Spark cluster, the runtime for the base case (all data remote) is 13000 seconds; for a = 0.001, i.e. almost no data sharing, the performance gain over that comes from data prefetching.

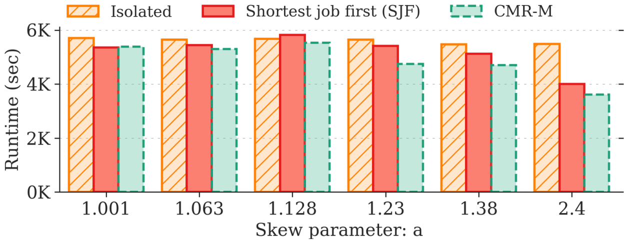


Figure 11: *Performance of different caching strategies when different level of data sharing exist within the cluster. (simulated)*

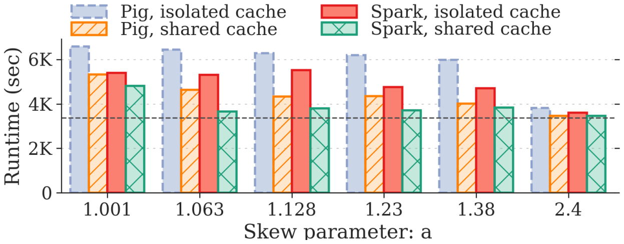


Figure 12: *Performance of different frameworks with shared cache vs isolated cache per framework. Each cache has 1 TB cache, 25 Gb/s storage-bandwidth, and 100 Gb/s cache bandwidth. (simulated)*

Two analytic clusters: We compare the performance of Kariz with multiple analytic clusters (Pig/MapReduce and Spark) sharing a cache vs the case where the cache is statically partitioned, using CMR-M with the CMR single-DAG planner. We use the same 200 Spark TPC-H queries, combined with 100 Pig TPC-H queries randomly selected from the same distribution, generating 6 traces with 300 datasets each as described above.

As shown in Figure 12, by increasing the data sharing across analytic clusters, both clusters have seen runtime improvement vs the statically-partitioned case. Pig cluster runtime improves by up to 1.5× ($a = 1.38$), with a mean of 1.3×. The Spark cluster benefits from both prefetching and caching, improving on average by 1.27× and up to 1.52× ($a = 1.128$). The dashed horizontal line in the Figure 12 shows the extreme case when one dataset is shared by all the queries in both clusters.

6.7 Sensitivity Analysis

We analyze sensitivity to cache size and prediction errors.

Cache size: Figure 13 shows the average speed up of all DAGs from the mixed workloads (§6.1) as we vary the cache size from 16GB to 400GB (the size of the dataset) with the network bandwidth to the backend set to 10Gbps. CMR achieves substantially higher performance compared to MRD and CP until the entire data set fits in the cache. For example, when the cache size is 64GB, CMR outperforms MRD and CP by up to 51% and on average 10% and 8% respectively.

Impact of runtime mis-prediction: To see the effect of runtime misprediction we introduce a multiplicative error factor R_{error}. We simulate 27 queries with a total of 310 jobs from the

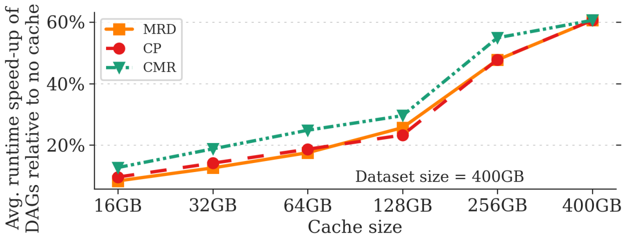


Figure 13: *Mean runtime across all queries vs. cache size, normalized to uncached runtime; 10 Gb/s bandwidth. (simulated)*

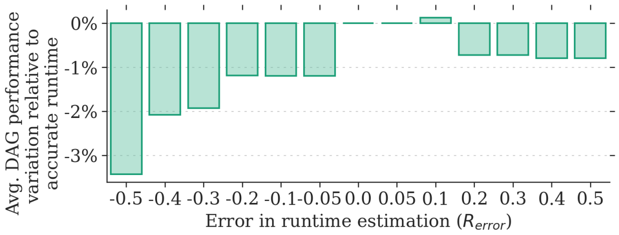


Figure 14: *Sensitivity to misprediction error: introduced error vs. performance degradation - 30% of predications adjusted by factor of* $(1+R_{error})$. *(simulated)*

mixed workloads in single-DAG mode, with 60% of jobs recurring, and randomly pick 130 jobs (~42%) to adjust by R_{error}.

In Figure 14 we see normalized change in runtime, relative to no mis-estimation, for values of $R_e rror$ between 0.5 and 1.5. CMR performance drops when the runtime is mispredicted, especially in the negative direction, but when only a fraction of jobs are mispredicted the effect is small.

6.8 Scalability

To analyze CMR scalability, we run (in simulation) a pool of 67 queries(DAGs) from TPC-H and TPC-DS benchmarks. Using timing from the Alibaba traces [1] (80% of DAG stages complete in less than one minute) we assign a random execution time between 1s to 60s to each stage. We submit DAGs at a rate of 90 per minute and measure the execution time for CMR planning.

Figure 15 shows CMR planner runtime vs a number of currently executing DAGs. Execution time (in unoptimized Python) is seen to be under six seconds in all cases, with up to 160 concurrent DAGs. Based on Alibaba statistics this would allow scaling a single controller to a cluster of 1500 to 2000 servers, with minimal delay in issuing prefetch commands.

7 Conclusion

Kariz is a cache management system for analytic frameworks that makes possible cache algorithms informed by DAGs, historical run time information, current cache state, and storage bandwidth. We have implemented multiple algorithms using Kariz, including a new CMR algorithm that achieves dramatic performance improvements by exploiting all this information.

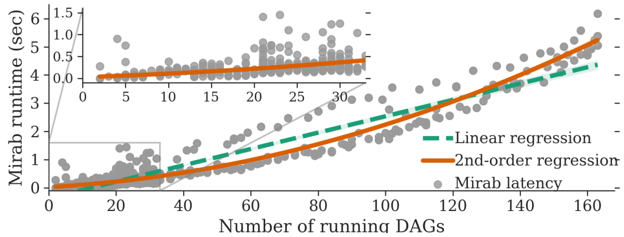


Figure 15: *CMR-M scaling - planner runtime vs running DAGs. (simulated)*

We demonstrate that new analytics frameworks (100 LOC for PIG/Hadoop, and 30 LOC for SPARK) and cache systems (100 LOC for the cache we used) can easily be integrated. Our work is the first to: 1) support multiple concurrent DAGs, 2) employ more than one of bandwidth, runtime prediction, and DAGs, 3) explore a cooperative caching model, and 4) employ straggler resistant partial caching.

Acknowledgment

We thank our shepherd, Nitin Agrawal, and our anonymous FAST reviewers for their valuable feedback and suggestions. We thank Shankar Pasupathy, Art Harkin, Peter Macko, and Xing Lin of NetApp, Matt Benjamin and Ali Maredia of Red Hat for their support and contribution. We would like to acknowledge the support of our commercial partners in Mass Open Cloud and Open Infra Labs, which include Red Hat, Two Sigma, Intel, IBM, Brocade, Cisco, and Lenovo. Partial support for this work was provided by the National Science Foundation award CNS-1910327, CNS-1414119, and a NetApp faculty fellowship.

References

[1] Alibaba trace data. `http://github.com/alibaba/clusterdata`, 2019.

[2] Alluxio. `http://www.alluxio.org`, 2019.

[3] Amazon EMR. `http://aws.amazon.com/emr/`, 2019.

[4] Amazon S3. `http://aws.amazon.com/s3/`, 2018.

[5] Anatomy of the S3A filesystem client. `http://redhat.com/en/blog/anatomy-s3a-filesystem-client`, 2018.

[6] Apache Arrow. `http://arrow.apache.org/`, 2019.

[7] Apache Ignite. `http://ignite.apache.org/`, 2019.

[8] Ceph Object Gateway. `http://docs.ceph.com/docs/master/radosgw/`, 2019.

[9] Dave Wells. The Future of the Data Warehouse. `http://eckerson.com`, 2017.

[10] Microsoft Azure HDInsight. `http://azure.microsoft.com/services/hdinsight/`, 2019.

[11] Microsoft Datalake. `http://azure.microsoft.com/en-us/solutions/data-lake`, 2019.

[12] Pig TPC-DS queries. `http://github.com/ssavvides/tpcds-pig`, 2019.

[13] Pig TPC-H queries. `http://github.com/ssavvides/tpch-pig`, 2019.

[14] Scheduling Spark cluster. `http://spark.apache.org/docs/latest/job-scheduling`, 2019.

[15] Spark TPC-DS queries. `http://github.com/databricks/spark-sql-perf`, 2019.

[16] Spark TPC-H queries. `http://github.com/ssavvides/tpch-spark`, 2019.

[17] Mania Abdi, Amin Mosayyebzadeh, Mohammad H. Hajkazemi, Ata Turk, Orran Krieger, and Peter Desnoyers. Caching in the Multiverse. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[18] Waleed Ali, Siti Mariyam Shamsuddin, and Abdul Samad Ismail. A Survey of Web Caching and Prefetching. *International Journal of Advances in Soft Computing and its Applications*, 3, 03 2011.

[19] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, March 2017. USENIX Association.

[20] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 267–280, San Jose, CA, 2012. USENIX.

[21] Danilo Ardagna, Enrico Barbierato, Athanasia Evangelinou, Eugenio Gianniti, Marco Gribaudo, Túlio B. M. Pinto, Anna Guimarães, Ana Paula Couto da Silva, and Jussara M. Almeida. Performance Prediction of Cloud-Based Big Data Applications. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 192–199, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[23] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, Cascais, Portugal, October 2011. Association for Computing Machinery.

[24] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. Unearthing inter-job dependencies for better cluster scheduling. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1205–1223. USENIX Association, November 2020.

[25] Jon Crowcroft and Philippe Oechslin. Differentiated End-to-end Internet Services Using a Weighted Proportional Fair Sharing TCP. *SIGCOMM Comput. Commun. Rev.*, 28(3):53–69, July 1998.

[26] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[27] Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 7–17, Feb 2009.

[28] Iman Elghandour and Ashraf Aboulnaga. ReStore: Reusing Results of MapReduce Jobs. *Proc. VLDB Endow.*, 5(6):586–597, February 2012.

[29] Hajar Falahati, Mania Abdi, Amirali Baniasadi, and Shahin Hessabi. ISP: Using idle SMs in hardware-based prefetching. In *The 17th CSI International Symposium on Computer Architecture Digital Systems (CADS 2013)*, pages 3–8, Oct 2013.

[30] Hajar Falahati, Shahin Hessabi, Mania Abdi, and Amirali Baniasadi. Power-efficient prefetching on GPGPUs. *The Journal of Supercomputing*, 71(8):2808–2829, Aug 2015.

[31] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 99–112, New York, NY, USA, 2012. Association for Computing Machinery.

[32] Seyedeh G. Ghaemi, Iman Ahmadpour, Mehdi Ardebili, and Hamed Farbeh. SMARTag: Error Correction in Cache Tag Array by Exploiting Address Locality. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1658–1663, 2018.

[33] Seyedeh G. Ghaemi, Iman Ahmadpour, Mehdi Ardebili, and Hamed Farbeh. Sleepy-LRU: extending the lifetime of non-volatile caches by reducing activity of age bits. *The Journal of Supercomputing*, 75(7):3945–3974, 2019.

[34] Seyedeh G. Ghaemi, Amir M. H. Monazzah, Hamed Farbeh, and Seyed G. Miremadi. LATED: Lifetime-Aware Tag for Enduring Design. In *2015 11th European Dependable Computing Conference (EDCC)*, pages 97–107, 2015.

[35] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[36] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, Savannah, GA, November 2016. USENIX Association.

[37] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters, booktitle = Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. OSDI'10, pages 75–88, Berkeley, CA, USA, 2010. USENIX Association.

[38] Mohammad H. Hajkazemi, Mania Abdi, and Peter Desnoyers. Minimizing Read Seeks for SMR Disk. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 146–155, 2018.

[39] Mohammad H. Hajkazemi, Mania Abdi, and Peter Desnoyers. uCache: a mutable cache for SMR translation layer. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8, 2020.

[40] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Bridging the Tenant-provider Gap in Cloud Services. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 10:1–10:14, New York, NY, USA, 2012. ACM.

[41] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. *SIGCOMM Comput. Commun. Rev.*, 45(4):407–420, August 2015.

[42] Virajith Jalaparti, Chris Douglas, Mainak Ghosh, Ashvin Agrawal, Avrilia Floratou, Srikanth Kandula, Ishai Menache, Joseph Seffi Naor, and Sriram Rao. Netco: Cache and I/O Management for Analytics over Disaggregated Stores. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 186–198, New York, NY, USA, 2018. ACM.

[43] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. Computation Reuse in Analytics Job Service at Microsoft. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 191–203, New York, NY, USA, 2018. ACM.

[44] Emine Ugur Kaynar, Mania Abdi, Mohammad H. Hajkazemi, Ata Turk, Raja R. Sambasivan, David Cohen, Larry Rudolph, Peter Desnoyers, and Orran Krieger. D3N: A multi-layer cache for the rest of us. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 327–338, 2019.

[45] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 759–773, Boston, MA, July 2018. USENIX Association.

[46] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[47] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[48] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296, Santa Clara, CA, February 2020. USENIX Association.

[49] Umesh Kumar and Jitendar Kumar. A Comprehensive Review of Straggler Handling Algorithms for MapReduce Framework. *International Journal of Grid and Distributed Computing*, 7(4):139–148, August 2014.

[50] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[51] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[52] Apoorve Mohan, Shripad Nadgowda, Bhautik Pipaliya, Sona Varma, Sahil Suneja, Canturk Isci, Gene Cooperman, Peter Desnoyers, Orran Krieger, and Ata Turk. Towards Non-Intrusive Software Introspection and Beyond. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pages 173–184, 2020.

[53] Apoorve Mohan, Ata Turk, Ravi S. Gudimetla, Sahil Tikale, Jason Hennesey, Emine Ugur Kaynar, Gene Cooperman, Peter Desnoyers, and Orran Krieger. M2: Malleable Metal as a Service. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 61–71, 2018.

[54] Amin Mosayyebzadeh, Apoorve Mohan, Sahil Tikale, Mania Abdi, Nabil Schear, Trammell Hudson, Charles Munson, Larry Rudolph, Gene Cooperman, Peter Desnoyers, and Orran Krieger. Supporting Security Sensitive Tenants in a Bare-Metal Cloud. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 587–602, Renton, WA, July 2019. USENIX Association.

[55] Siegfried Nijssen and Joost N. Kok. The Gaston Tool for Frequent Subgraph Mining. *Electronic Notes in Theoretical Computer Science*, 127(1):77 – 87, 2005. Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004).

[56] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[57] Tiago B. G. Perez, Xiaobo Zhou, and Dazhao Cheng. Reference-distance Eviction and Prefetching for Cache Management in Spark. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 88:1–88:10, New York, NY, USA, 2018. ACM.

[58] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 51–63, New York, NY, USA, 2017. ACM.

[59] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless Data Analytics in the IBM Cloud. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, pages 1–8, New York, NY, USA, 2018. Association for Computing Machinery.

[60] Elizabeth Shriver, Christopher Small, and Keith A Smith. Why does file system prefetching work? 1999.

[61] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert 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] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 996–1005, March 2010.

[63] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[64] Steven P. Vanderwiel and David J. Lilja. Data Prefetch Mechanisms. *ACM Comput. Surv.*, 32(2):174–199, June 2000.

[65] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[66] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, March 2016. USENIX Association.

[67] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[68] Luna Xu, Min Li, Li Zhang, Ali R. Butt, Yandong Wang, and Zane Zhenhua Hu. MEMTUNE: Dynamic Memory Management for In-Memory Data Analytic Platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 383–392, May 2016.

[69] Yinggen Xu, Liu Liu, and Zhijun Ding. DAG-Aware Joint Task Scheduling and Cache Management in Spark Clusters. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 378–387, May 2020.

[70] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Management of Multilevel, Multiclient Cache Hierarchies with Application Hints. *ACM Trans. Comput. Syst.*, 29(2):5:1–5:51, May 2011.

[71] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. LRC: Dependency-aware cache management for data analytics clusters. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

[72] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion 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.

Learning Cache Replacement with CACHEUS

Liana V. Rodriguez[†*] Farzana Yusuf[†*] Steven Lyons[†] Eysler Paz[†]
Raju Rangaswami[†] Jason Liu[†] Ming Zhao[‡] Giri Narasimhan[†]
[†] *Florida International University* [‡] *Arizona State University*

Abstract

Recent advances in machine learning open up new and attractive approaches for solving classic problems in computing systems. For storage systems, cache replacement is one such problem because of its enormous impact on performance. We classify workloads as a composition of four workload primitive types — *LFU-friendly, LRU-friendly, scan*, and *churn*. We then design and evaluate CACHEUS, a new class of fully adaptive, machine-learned caching algorithms that utilize a combination of experts designed to address these workload primitive types. The experts used by CACHEUS include the state-of-the-art ARC, LIRS and LFU, and two new ones – SR-LRU, a scan-resistant version of LRU, and CR-LFU, a churn-resistant version of LFU. We evaluate CACHEUS using 17,766 simulation experiments on a collection of 329 workloads run against 6 different cache configurations. Paired t-test analysis demonstrates that CACHEUS using the newly proposed lightweight experts, SR-LRU and CR-LFU, is the most consistently performing caching algorithm across a range of workloads and cache sizes. Furthermore, CACHEUS enables augmenting state-of-the-art algorithms (e.g., LIRS, ARC) by combining it with a complementary cache replacement algorithm (e.g., LFU) to better handle a wider variety of workload primitive types.

1 Introduction

Cache replacement algorithms have evolved over time with each algorithm attempting to address some shortcomings of previous algorithms. However, despite the many advances, state-of-the-art caching algorithms continue to leave room for improvement. First, as demonstrated abundantly in the literature, caching algorithms that do well for certain workloads do not perform well for others [23, 13, 20, 12, 29, 34]. The production storage workloads of today are significantly diverse in their characteristic features and these features can vary over time even within a single workload. Second, as demonstrated recently [34], caching algorithms that do well for certain cache sizes do not necessarily perform well for other cache sizes. Indeed, the workload-induced dynamic cache state, the cache-relevant workload features, and thereby the most effective strategies, can all vary as cache size changes.

The ML-based LeCaR algorithm demonstrated that having access to two simple policies, LRU and LFU was sufficient to outperform ARC across specific production-class workloads. LeCaR used *regret minimization* [22, 21], a machine learning technique that allowed the dynamic selection of one of these policies upon a *cache miss*. We review LeCaR both analytically and empirically to demonstrate that while LeCaR took a valuable first step, it had significant limitations. As a result, LeCaR underperforms state-of-the-art algorithms such as ARC, LIRS, and DLIRS for many production workloads.

As our first contribution, we identify the cache-relevant features that inform *workload primitive types*. In particular, we identify four workload primitive types: *LRU-friendly*, *LFU-friendly*, *scan*, and *churn*. The workload primitive types vary across workloads, within a single workload over time, and as cache size changes. Our second contribution, CACHEUS, is inspired by LeCaR but overcomes an important shortcoming by being completely *adaptive*, with the elimination of all statically chosen hyper-parameters, thus ensuring high flexibility. Our third contribution is the design of two lightweight experts, CR-LFU and SR-LRU; put together, these address a broad range of workload primitive types. CR-LFU infuses LFU with *churn resistance* and SR-LRU infuses LRU with *scan resistance*. CACHEUS when using the proposed two experts is able to perform competitively or better for a significant majority of the (workload, cache-size) combinations when compared with the state-of-the-art.

We evaluate CACHEUS using 17,766 simulation experiments on a workload collection comprising of over 329 individual single-day workloads sourced from 5 different production storage I/O datasets. For each workload, we evaluate against 6 different cache configurations that are sized relative to the individual workload's *footprint*, the set of unique data accessed. We perform *paired t-tests* analysis comparing CACHEUS against individual algorithms across 30 different (workload, cache-size) combinations. CACHEUS using SR-LRU and CR-LFU as experts is the most consistently performing algorithm with 87% of the workload-cache combinations being the best or indistinguishable from the best performing algorithm, and distinctly different than the best performing algorithm for the remaining 13%. For the 13%

[*] The first two authors contributed equally to this work.

Dataset	# Traces	Footprint	Requests	Details
FIU [33, 16]	184	398MB	314563	End user home directories; Webpage and web-based email servers; Online course management system
MSR [33, 24]	22	467MB	4126937	User home and Project directories; Hardware monitoring; Source control; Web staging; Terminal, Web/SQL, Media, Test web servers; Firewall/web proxy
CloudPhysics [35]	99	458MB	2470326	VMware VMs from cloud enterprise
CloudVPS [2]	18	3.7GB	3400025	VMs from cloud provider
CloudCache [2]	6	6.2GB	3867313	Online course website; CS department web server

Table 1: Descriptions for the 5 datasets used (average *footprint* and requests). *Each trace has a 1 day duration.*

cases where an algorithm other than CACHEUS is found to be distinctly better, no single algorithm is found to be consistently the best, indicating that CACHEUS is a good default choice. Finally, when using state-of-the-art algorithms such ARC and LFU, we show that the CACHEUS framework provides a simple way to enable access to an additional expert with complementary expertise such as LFU. These CACHEUS variants achieve at least competitive performance when compared against the original algorithms and other competitors.

2 Motivation

2.1 Understanding Workloads

Caching algorithms in the past have optimized for specific workload properties. As today's workloads continue to increase in complexity, even state-of-the-art algorithms demonstrate inconsistent performance. To understand the production storage workloads of today, we analyzed over 329 production storage traces sourced from 5 different production collections (see Table 1).

2.1.1 Workload Primitive Types

Based on our analysis of production storage workloads, we define the following set of workload primitive types.

- **LRU-friendly** defined by an access sequence that is best handled by the *least recently used (LRU)* caching algorithm.
- **LFU-friendly** defined by an access sequence that is best handled by the *least frequently used (LFU)* caching algorithm.
- **Scan** defined by an access sequence where a subset of stored items are accessed exactly once.
- **Churn** defined by repeated accesses to a subset of stored items with each item being accessed with equal probability.

Figure 1 shows an example of how the workload primitive types manifest in a production trace from the FIU collection. As one may notice, the primitive types are not all exclusive — for instance, a workload that's *LRU-friendly* may also manifest the *churn* type. Our goal was identifying workload primitive types that would directly inform specific, yet distinct, caching decisions.

We found that most of the workloads that we examined contained at least one occurrence of each of the workload primitive types. However, these workloads were not all the same in their composition. For instance, the MSR collection contains all the primitive types with one of the workloads *(proj3)* mostly comprising a single long scan. A summary of our findings are presented in Table 2.

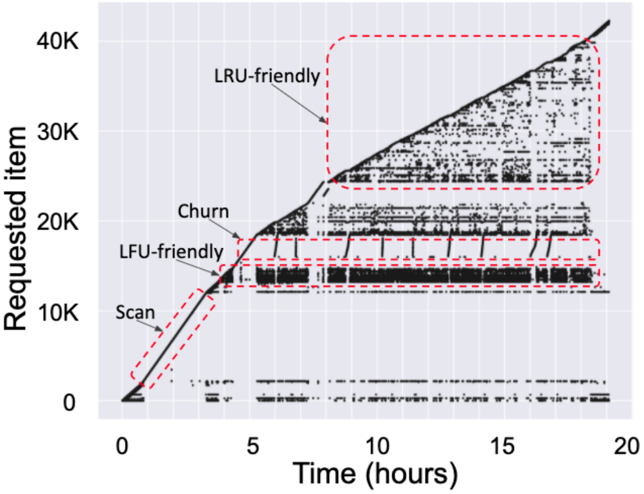


Figure 1: Access pattern for the *topgun (day* 16*)* workload from the FIU trace collection. *Dashed lines highlight manifestation of workload primitive types.*

2.1.2 Composing Workloads

Modern storage workloads are typically a composition of the above workload primitive types. Furthermore, as the cache size changes, a single workload's primitive type may vary. For instance, an *LRU-friendly* type workload at cache size C_1 may transform into a *Churn* type at a cache size $C_2 < C_1$. This can occur when items in the workload's LRU-friendly working set start getting removed from the cache prior to being reused. Figure 2 illustrates this phenomenon by comparing the performance of LRU against the churn-friendly CR-LFU algorithm proposed in this paper. Finally, storage working sets are telescoping in nature with larger subsets

Dataset	Churn	Scan	LRU	LFU
FIU [33, 16]	✓	✓	✓	✓
MSR [33, 24]	✓	✓	✓	✓
CloudPhysics [35]	✓	✓	✓	✓
CloudVPS [2]	✓	✓	✓	✓
CloudCache [2]	✓	✗	✓	✓

Table 2: Workload Primitive Types identified using algorithms that optimize for each primitive type.

Algorithm	Churn	Scan	LRU	LFU
ARC	✗	✓	✓	✗
LIRS*	✗	✓	✗	✗
LeCaR*	✓	✗	✓	✓
DLIRS	✗	✓	✓	✗

Table 3: Caching algorithms handling of workload primitive types. *Parametric algorithms are noted using an *.*

of items accessed at a lower frequency often each entirely subsuming one or more smaller subsets of items accessed at a higher frequency [17, 27]. The LeCaR [34] algorithm was the first to demonstrate an ability to adapt its behavior based on the available cache size, independent of the ability to adapt to the dynamics of the workload.

2.2 Caching Algorithms

Adaptive Replacement Cache (ARC): ARC [23] is an adaptive caching algorithm that is designed to recognize both recency and frequency of access. ARC divides the cache into two LRU lists, T_1 and T_2. T_1 holds items accessed once while T_2 keeps items accessed more than once since admission. Since ARC uses an LRU list for T_2, it is unable to capture the full frequency distribution of the workload and perform well for LFU-friendly workloads. For a *scan* workload, new items go through T_1 protecting frequent items previously inserted into T_2. However, for *churn* workloads, ARC's inability to distinguish between items that are equally important leads to continuous cache replacement [29].

Low Interference Recency Set (LIRS): LIRS [13] is a state-of-the-art caching algorithm based on reuse distance. LIRS handles *scan* workloads well by routing one-time accesses via its short filtering list Q. However, LIRS's ability to adapt is compromised because of its use of a fixed-length Q. In particular, if reuse distances exceed the 1% length, LIRS is unable to recognize reuse quickly enough for items with low overall reuse. And, similar to ARC, LIRS does not have access to the full frequency distribution of accessed items which limits its effectiveness for *LFU-friendly* workloads.

Dynamic LIRS (DLIRS): DLIRS [20] is a recently proposed caching policy that incorporates adaptation in LIRS. DLIRS dynamically adjusts the cache partitions assigned to high and low reuse-distance items. Although this strategy achieves performance comparable to ARC for some cache size configurations with *LRU-friendly* workloads while maintaining LIRS's behavior for scans, we found its performance inconsistent across the workloads we tested against. Finally, it inherits the LFU-unfriendliness of LIRS.

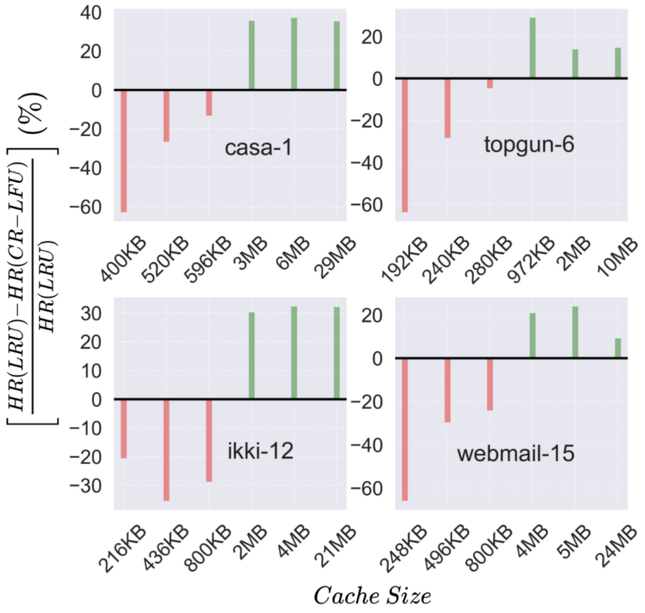


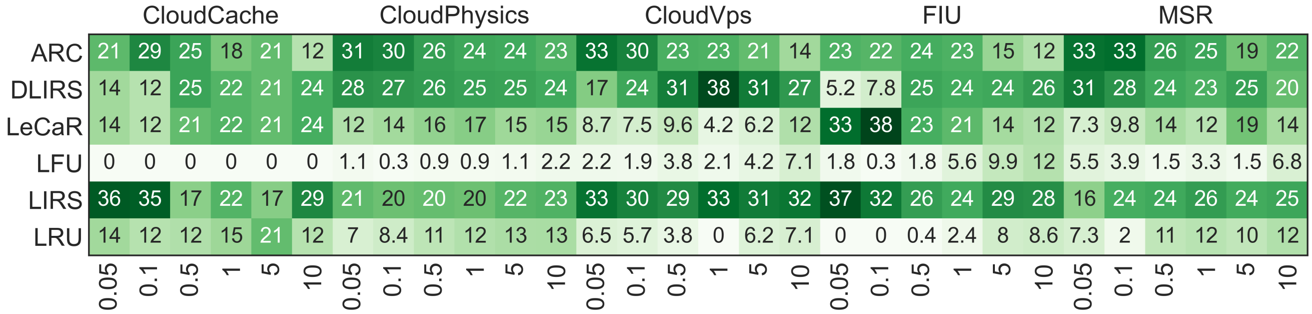
Figure 2: Relative difference in hit-rate (HR) of LRU and CR-LFU for *casa, topgun, ikki, and webmail* workloads from the FIU trace collection.

Learning Cache Replacement (LeCaR): LeCaR [34] is a machine learning-based caching algorithm that uses reinforcement learning and regret minimization to control its dynamic use of two cache replacement policies, LRU and LFU. LeCaR was shown to outperform ARC for small cache sizes for real-world workloads [34]. However, LeCaR has drawbacks relating to adaptiveness, overhead, and churn-friendliness. In Section 3, we discuss these limitations further.

In Table 3, we compare the current state-of-the-art algorithms in terms of their ability to handle various workload primitive types.

2.3 Need for a New Approach

Each of the state-of-the-art caching algorithms address a subset of workload primitive types. We conducted an empirical study using over 329 storage I/O traces from 5 different production systems, across 6 different workload-specific cache configurations — from 0.05% to 10% of the workload footprint. To understand relative performance across such a large collection of experiments, we ranked algorithms based on their achieved hit-rates for individual workloads. The best-performing algorithm received the rank of 1 as well as any other algorithm that achieved a hit-rate within a 5% *relative* margin. For example, if the best-performing algorithm achieves a hit-rate of 40%, any other algorithm that achieves a hit-rate within the range 38% to 40% is also ranked as 1, but anything lower than 38% is ranked 2 or higher. Next,



	CloudCache						CloudPhysics						CloudVps						FIU						MSR					
	0.05	0.1	0.5	1	5	10	0.05	0.1	0.5	1	5	10	0.05	0.1	0.5	1	5	10	0.05	0.1	0.5	1	5	10	0.05	0.1	0.5	1	5	10
ARC	21	29	25	18	21	12	31	30	26	24	24	23	33	30	23	23	21	14	23	22	24	23	15	12	33	33	26	25	19	22
DLIRS	14	12	25	22	21	24	28	27	26	25	25	24	17	24	31	38	31	27	5.2	7.8	25	24	24	26	31	28	24	23	25	20
LeCaR	14	12	21	22	21	24	12	14	16	17	15	15	8.7	7.5	9.6	4.2	6.2	12	33	38	23	21	14	12	7.3	9.8	14	12	19	14
LFU	0	0	0	0	0	0	1.1	0.3	0.9	0.9	1.1	2.2	2.2	1.9	3.8	2.1	4.2	7.1	1.8	0.3	1.8	5.6	9.9	12	5.5	3.9	1.5	3.3	1.5	6.8
LIRS	36	35	17	22	17	29	21	20	20	20	22	23	33	30	29	33	31	32	37	32	26	24	29	28	16	24	24	26	24	25
LRU	14	12	12	15	21	12	7	8.4	11	12	13	13	6.5	5.7	3.8	0	6.2	7.1	0	0	0.4	2.4	8	8.6	7.3	2	11	12	10	12

Figure 3: An analysis of the ranked performance of state-of-the-art caching algorithms. *The X-axis indicates cache size as a % of workload footprint. A darker cell indicates that an algorithm's performance across all workloads of the dataset was better. The number within each cell denotes the percentage of workloads for which an algorithm was ranked 1. For example, ARC has the highest hit-rate in 34% of the workloads for MSR at the 0.05% cache size.*

we computed the percentage of workloads within each set for which a given algorithm was assigned a rank of 1. We present this information in Figure 3.

Of the state-of-the-art caching algorithms, we observe that no algorithm is a clear winner. For instance, while LIRS achieves the best performance for CloudCache workloads at cache sizes of 0.05% and 0.1%, ARC outperforms the rest of the competitors for a majority of the MSR workloads, and LeCaR is the best for FIU workloads at a cache size of 0.1%. New caching algorithms that perform competitively across a wide range of workloads and cache configurations would be valuable.

3 CACHEUS

Given the distinct characteristics and dynamic manifestation of workload primitive types within a workload over time, caching algorithms need to be both nimble and adaptive. Online reinforcement learning is valuable because of its inherent ability to adapt to the unknown dynamics of the system being learned. CACHEUS uses online reinforcement learning with regret minimization to build a caching algorithm that attempts to optimize for dynamically manifesting workload primitive types. Since CACHEUS' design draws heavily from LeCaR, we review it briefly first, conduct an investigative study of LeCaR, and finally discuss the CACHEUS algorithm.

3.1 LeCaR: A Review

LeCaR demonstrated the feasibility of building a caching system that uses reinforcement learning and regret minimization. LeCaR learns the optimal eviction policy dynamically, choosing from exactly two basic experts, LRU and LFU. On each eviction, an expert is chosen randomly with probabilities proportional to the weights w_{LRU} and w_{LFU}. LeCaR dynamically learns these weights by assigning penalties for wrongful evictions.

To control online learning, LeCaR uses a *learning rate* parameter to set the magnitude of the change when the algorithm makes a poor decision. Larger learning rates allow quicker learning, but need larger corrections when the learning is flawed. LeCaR uses a *discount rate* parameter to decide how quickly to stop learning.

3.2 Running Diagnostics on LeCaR

In over 17,766 distinct caching simulations that we ran against LeCaR using 329 workloads, we found that experts other than LRU and LFU produced outcomes that were significantly better for a non-trivial number of workloads. In particular we found that LRU and LFU were unable to address the *scan* and *churn* workload primitive types. This motivates further exploration of the choice of experts for learning cache replacement within the regret minimization framework.

A second challenge when using LeCaR in practice is the manual configuration necessary for its two internal parameters — learning rate and discount rate. These parameters were fixed after experimenting with many workloads in LeCaR [34]. From the above empirical evaluation, we found that eliminating the discount rate altogether did not affect LeCaR's performance appreciably. Furthermore, different static values of the learning rate were found to be optimal for different workloads (see Figure 4). In addition, we observed across almost all workloads that not only do workload characteristics change substantially over time, the velocity and magnitude of these changes also varied significantly over time. To accommodate this dynamism, different values for the learning rate were found to be optimal at different points in time.

3.3 Formalizing CACHEUS(A,B)

CACHEUS starts off by simplifying and adapting LeCaR. First, for reasons discussed previously, CACHEUS simply eliminates the use of *discount rate*. Second, for adapting the *learning rate* hyper-parameter, we investigated adaptation approaches including grid search, random search [5], gaussian, bayesian and population based approaches [14, 36, 32, 6, 3, 19], and gradient-based optimization [26, 7, 15, 28, 37,

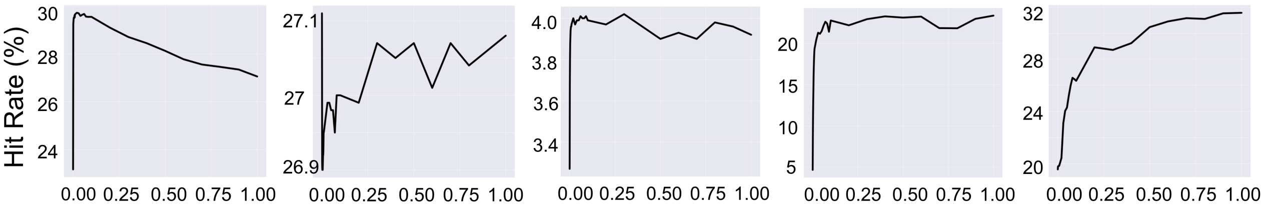


Figure 4: The optimal learning rate varies across workloads. *X-axis indicates learning rates. Cache size was chosen as 0.1% of workload footprint. We chose one workload each from CloudCache, CloudPhysics, CloudVPS, FIU, and MSR (from left to right).*

25, 8]. Ultimately, we chose a gradient-based stochastic hill climbing approach with random restart [31] for Cacheus, a choice that proved to be the most consistent. Using this technique, at the end of every window of N requests ($N =$ cache size), the gradient of the performance (average hit-rate) with respect to the learning rate over the previous two windows is calculated. If the gradient is positive (negative, resp.), then the direction of change of the learning rate is sustained (reversed, resp.). The amount of change of learning rate in the previous window determines the magnitude of the change in learning rate for the next window. Therefore, if the performance increases (decreases, resp.) by increasing the learning rate, we will increase (decrease, resp.) the learning rate multiplying it by the amount of learning rate change from the previous window, and vice versa. But, if the learning rate doesn not change for consectuive windows, and the performance degrades continuously or becomes zero, we record this. If the performance keeps degrading for a 10 consecutive window sizes [9], we reset the learning rate to the initial value. The objective behind is to make sure we restart the learning when the performance drops for a longer period. The learning rate is initialized randomly between 10^{-3} and 1.

The goal of the Cacheus framework is to enable a single cache replacement policy that uses the combination of individual decisions taken by exactly two internal experts. Algorithm 1 depicts the generalized Cacheus(A,B) algorithm with generic cache replacement experts, A and B. H_A and H_B are LRU lists of the history of items evicted by experts A and B, respectively, each of size $N/2$. Upon a cache hits, Cacheus updates the internal data structures which includes moving the item to the MRU position of the cache and updating its frequency information. Upon a cache miss, Cacheus checks the eviction histories for the requested item q, removes it from said histories, and updates the weights w_A and w_B. The weights are initialized to 0.5. Using the updated weights (Algorithm 2), Cacheus chooses the expert (A or B) to use and obtains the eviction candidate accordingly, $A(C)$ or $B(C)$. Finally, Cacheus updates its history, avoiding this update entirely if both experts suggest the same eviction candidate.

At the end of every window of N requests ($N =$ cache size), Cacheus updates its learning rate (Algorithm 3). First, the gradient of the performance (average hit-rate) with respect to the learning rate over the previous two windows is calculated. If the gradient is positive (negative, resp.), then the direction of change of the learning rate is sustained (reversed, resp.). The amount of gradient change determines the magnitude of the change in the learning rate. If the performance increases (decreases, resp.) by changing the learning rate, we will increase (decrease, resp.) the learning rate by an amount proportional to the learning rate change relative to the previous window. The learning rate is initialized randomly between 10^{-3} and 1. Finally, if the performance keeps degrading for a 10 consecutive window sizes [9], we reset the learning rate.

Like LeCaR, Cacheus uses exactly two experts. The usage of more than two experts was considered for early Cacheus versions. Interestingly, the performance with more than two experts was significantly worse than when using only LRU and LFU. Having multiple experts is generally not beneficial unless the selected experts are orthogonal in nature, and operate based on completely different and complementary strategies. The intuition here is that multiple experts will overlap in their eviction decisions thereby affecting learning outcomes and deteriorating the performance. We demonstrate in this paper that with two well-chosen experts Cacheus is able to best the state-of-the-art with statistical significance.

4 Scan Resistance

Our initial experiments with Cacheus using LRU and LFU as experts demonstrated inconsistent results when tested with a significantly wider range of workloads than the original LeCaR study did [34]. Of particular concern was the inability of Cacheus(LRU, LFU) to handle the *scan* workload primitive type. Of the 5 different datasets comprising a total of over 329 different workloads that we examined, 4 of the datasets comprised *scan* workloads (see Table 2).

To understand the impact of *scan* on classic caching algorithms, we set up synthetic workloads that interleaved reuse with scan. Figure 5 shows performance versus cache size for two synthetic workloads wherein a single scan of size 60 items is interleaved between accesses to reused items. Let us assume that the *scan* phase is greater than twice the size of the cache (say 25). In this case, classic algorithms such as LRU evict resident items to absorb the new items anticipating their future reuse, giving up on hits for resident items that get reused beyond the scan phase.

State-of-the-art caching algorithms such as ARC, LIRS

Algorithm 1: CACHEUS(A, B)

```
Data: Cache $C$; Eviction histories $H_A$, $H_B$;
Weights $w_A$, $w_B$; Current time $t$;
Learning rate update interval $i$;
$\lambda_t$ — learning rate at time $t$;
$HR_t$ — average hit-rate at time $t$
Input: Requested page $q$
if $q \in C$ then
    $C$.UPDATEDATASTRUCTURES($q$)
else
    UPDATEWEIGHT($q, \lambda, w_A, w_B$)
    if $q \in H_A$ then
        $H_A$.DELETE($q$)
    if $q \in H_B$ then
        $H_B$.DELETE($q$)
    if $C$ is full then
        if $A(C) == B(C)$ then
            $C$.EVICT($A(C)$)
        else
            action = ($A$, $B$) w/prob $(w_A, w_B)$
            if $(action == A)$ then
                if $H_A$ is full then
                    $H_A$.DELETE(LRU($H_A$))
                $H_A$.ADDMRU($A(C)$)
                $C$.EVICT($A(C)$)
            if $(action == B)$ then
                if $H_B$ is full then
                    $H_B$.DELETE(LRU($H_B$))
                $H_B$.ADDMRU($B(C)$)
                $C$.EVICT($B(C)$)
    $C$.ADD($q$)
if $(t \% i) = 0$ then
    UPDATELEARNINGRATE( $\lambda_{t-i}$, $\lambda_{t-2i}$, $HR_t$,
    $HR_{t-i}$)
```

and DLIRS each implement their own mechanisms for scan resistance. ARC limits the size of its $T1$ list used to identify and cache newly accessed items to preserve reused items in $T2$. Unfortunately, ARC's approach to scan-resistance makes it ineffective when handling the *churn* workload pattern. In particular, when a *scan* phase is followed by a *churn* phase, ARC continues to evict from $T1$ and behaves similar to LRU, as evidenced in one of our experiments (see Figures 10 and 11). Similarly, LIRS uses its stack Q to accommodate items that belong to the *scan* sequence. However, the size of Q is fixed to 1% of the cache , which cannot adapt to dynamic working sets. Finally, DLIRS reworks LIRS's solution by making Q adaptive. Despite its built-in adaptation mechanism, we note that DLIRS does not perform as well as LIRS in practice (see Figure 3).

4.1 SR-LRU

One policy that handles *scan* well is the classic Most Recently Used (MRU) policy. While LRU consistently evicts resident working-set items during scan, MRU evicts the previously inserted page placed at the top of the stack. We designed Scan-Resistant LRU (SR-LRU), an LRU variant that favors LRU friendly workloads while also being *scan* aware.

Algorithm 2: UPDATEWEIGHT(q, λ, w_A, w_B)

```
if $q \in H_A$ then
    $w_A := w_A * e^{-\lambda}$ // decrease $w_A$
else if $q \in H_B$ then
    $w_B := w_B * e^{-\lambda}$ // decrease $w_B$
$w_A := w_A / (w_A + w_B)$ // normalize
$w_B := 1 - w_A$
```

Algorithm 3: UPDATELEARNINGRATE(λ_{t-i}, λ_{t-2i}, HR_t, HR_{t-i})

```
$\delta_{HR_t} := HR_t - HR_{t-i}$
$\delta_{LR_t} := \lambda_{t-i} - \lambda_{t-2i}$
if $\delta_{LR_t} \neq 0$ then
    $sign := +1$ if $\frac{\delta_{HR_t}}{\delta_{LR_t}} > 0$, else $-1$
    $\lambda_t := \max(\lambda_{t-i} + sign \times |\lambda_{t-i} \times \delta_{LR_t}|, 10^{-3})$
    $unlearnCount := 0$
else
    if $HR_t = 0$ or $\delta_{HR_t} \leq 0$ then
        $unlearnCount := unlearnCount + 1$
    if $unlearnCount \geq 10$ then
        $unlearnCount := 0$
        $\lambda_t :=$ choose randomly between $10^{-3}$ & 1
```

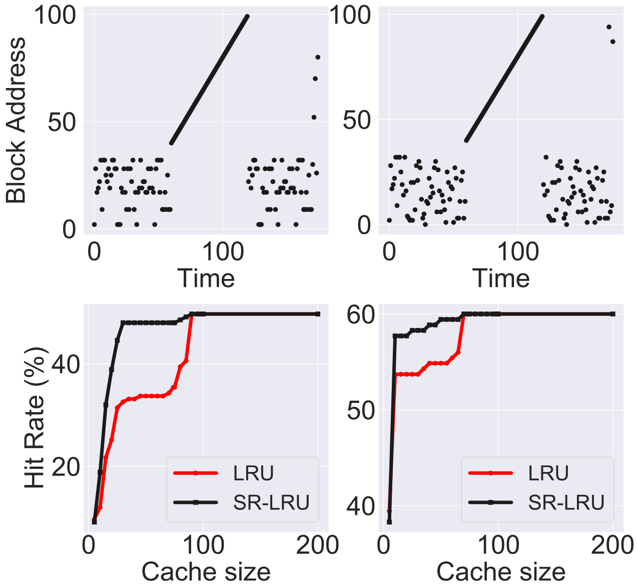


Figure 5: Motivating SR-LRU with the *scan* workload primitive type. *Two synthetic workloads are considered with 175 total requests and a single inserted scan: LFU-friendly pattern (left column) and LRU-Friendly pattern (right column). The size of the scan is 60 items in both cases.*

SR-LRU manages the cache in partitions similar to ARC

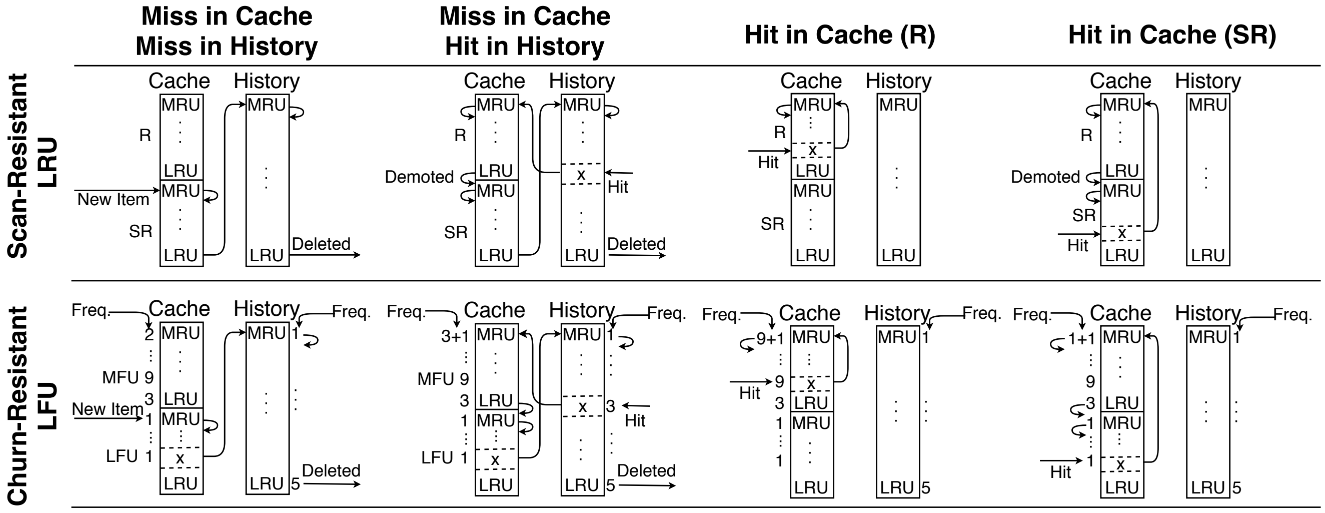


Figure 6: Understanding CR-LFU and SR-LRU. *Shown are actions taken to handle request x under:* cache miss*,* cache miss with *x* in history*,* cache hit with *x* in *SR, and* cache hit with *x* in *R*.

and LIRS. It divides the cache into two parts: one containing only items with multiple accesses (*R*) and the other for single access items as well as older items that have had multiple accesses (*SR*). The *SR* partition allows SR-LRU to be scan resistant; a partition for *new* items to be housed so that they do not affect the important items in *R*. SR-LRU only evicts from the *SR* partition — it evicts the LRU item of *SR* on a cache miss when the cache is full. Older items in *R* get *demoted* to *SR* to keep only important items that are being reused in *R*. In addition, SR-LRU maintains a history list *H* as large as the size of the cache that contains the items most recently evicted.

The basic workings of SR-LRU are as shown in Algorithm 4. We illustrate how a request for page *x* gets handled in Figure 6. On a cache miss where *x* is *not* in a history list, *x* is inserted to the MRU position of *SR*. Should the cache be full, the LRU item of *SR* is evicted to *H*, and should *H* be full the algorithm removes the LRU item of *H* to make space. On a cache miss where *x is* in *H*, *x* is moved to the MRU position of *R*. On a cache hit where *x* is in *SR*, *x* is moved to the MRU position of *R*. On a cache hit where *x* is in *R*, *x* is moved to the MRU position of *R*.

While SR-LRU could set a constant size for *SR* (similar to LIRS) and thereby be scan resistant, doing so would compromise its performance with LRU-friendly workloads for which *SR* is unfavorably sized [20]. Our approach to adapting SR-LRU is to adjust its partition sizes when we have found that SR-LRU either *demoted* or *evicted* incorrectly. If a demoted item gets referenced while in *SR*, SR-LRU infers that the size of *R* is too small and should be increased. To handle incorrect *evictions*, when an item is encountered for the first time, it gets marked as *new* after inserting it in cache. Should this item be evicted but then requested before it is removed from SR-LRU's history *H*, SR-LRU infers that the size of *SR* is too small to allow new items to be reused prior to being evicted. Items that enter the cache for the second time, after being placed in the history list previously, are not considered to be *new* items anymore.

Algorithm 4: SR-LRU

```
Data: Scan-resistant list $SR$; Reuse list $R$
$C_{demoted}$ — count of $demoted$ items in cache
$H_{new}$ — count of $new$ items in history
Input: requested page $q$
if $q \in C$ then
    if $q$ was demoted from $R$ then
        $\delta = max(1, H_{new}/C_{demoted})$
        $size_{SR} = max(1, size_{SR} - \delta)$
    $R$.MoveMRU($q$)
else
    if $q \in H$ then
        if $q$ was new from $SR$ then
            $\delta = max(1, C_{demoted}/H_{new})$
            $size_{SR} = min(|C| - 1, size_{SR} + \delta)$
        $H$.Delete($q$)
    if $C$ is full then
        if $H$ is full then
            $H$.Delete(LRU($H$))
        $H$.MoveMRU(LRU($SR$))
    $SR$.AddMRU($q$)
UpdateSizes($SR, R$)
```

To adapt itself, SR-LRU continuously computes a *target size* for *SR*. The algorithm reactively increases the size of *SR* upon hits in *H* by moving the LRU items of *R* into *SR* in order for *SR* to reach its *target* size. If the size of *SR* increases by too much, the demoted items being reused will inform the algorithm allowing it to reverse the erroneous increase.

The SR-LRU Difference Prior approaches to scan resistance are limited because they are either not adaptive (e.g.,

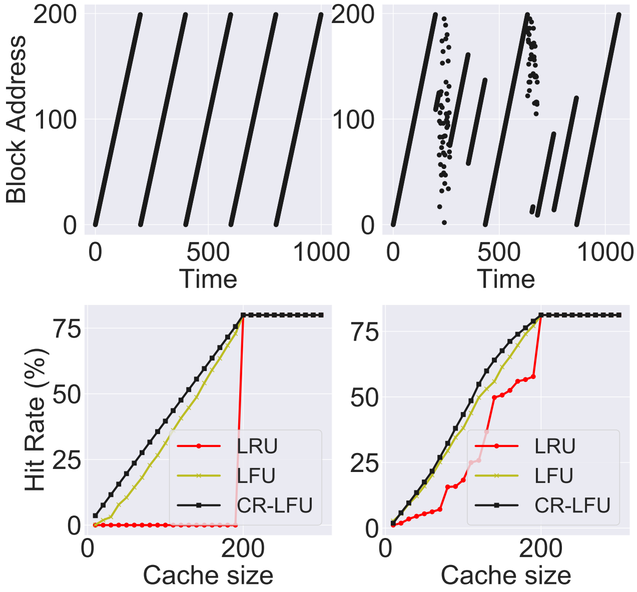


Figure 7: Motivating CR-LFU with the *churn* workload primitive type. *Two synthetic workloads are considered: a churn pattern (left column) and a combination of churn and LRU-friendly pattern (right column). The working set is 200 items.*

LIRS) or do not adapt well enough (e.g., DLIRS), or are unable to handle a scan followed by churn (e.g., ARC). The most important distinction in SR-LRU is balancing the need for being scan resistant with quickly recognizing when a workload is no longer *scanning*. In particular, SR-LRU tracks new items in history to distinguish between new items that belong to a scan from the new items that contribute to churn. As a result, SR-LRU continues to be effective immediately when a workload switches from scan to churn, as evidenced in our experiments (see Figures 10 and 11).

5 Churn Resistance

For the *churn* workload primitive type, if the number of items being accessed is larger than the size of the cache, an LRU-style algorithm would lead to churning of the cache content whereby items get repeatedly inserted into and evicted from the cache. On the other hand, the classic LFU assigns equal importance to all items with the same frequency. In a *churn* workload, all items have the same access frequency and these items may be accessed sequentially or otherwise. Other frequency-based algorithms like LRFU [18], that assign weights based on recency of access, result in LRU-based eviction for items with the same frequency; this unfortunately, does not prevent churning either.

Fortunately, a simple modification of the classic LFU turns out to be sufficient to handle the *churn* workload primitive type while continuing to retain the benefits of LFU. Churn-resistant LFU (CR-LFU) modifies the eviction mechanism in pure LFU by choosing the MRU (Most Recently Used) item to break the ties when several items have the least access frequency. By choosing the MRU item, CR-LFU effectively "locks" a subset of items with the lowest frequency into the cache, generating hits for the caching algorithm. Figure 6 illustrates the operation of algorithm CACHEUS using the SR-LRU and CR-LFU while handling a page request x in different situations.

We compare CR-LFU with LRU and LFU in Figure 7 for two different types of synthetic workloads: pure *churn* and mixed pattern of *churn* and *LRU-friendly*. Both LFU and CR-LFU outperform LRU when the cache size is less than the workload's working set size. Classic LFU evicts at random from among multiple items with the lowest frequency whereas CR-LFU evicts the MRU item. Because of that distinction, the average performance of CR-LFU is 8.67% and 3.83% higher than LFU for the *churn* and mixed pattern workloads respectively.

6 Evaluation

6.1 Experimental Setup

We conducted simulation-based evaluations of several state-of-the-art algorithms from the caching literature using publicly available production storage I/O workloads.

Algorithms: We compared CACHEUS against 6 previously proposed algorithms: *LRU, LFU, ARC, LIRS, LeCaR,* and *DLIRS*. In cases we could successfully contact the algorithm authors to obtain an implementation, we used the authors' original versions. In all other cases, we reimplemented the algorithms.

We also evaluated each of these against 3 variants of CACHEUS — **C1**: CACHEUS(ARC, LFU), **C2**: CACHEUS(LIRS, LFU) and **C3**: CACHEUS(SR-LRU,CR-LFU).

Workloads and Simulations. We used production storage I/O traces from 5 different productions systems for the simulation evaluation. Table 1 summarizes the workload datasets we used. A total of 17,766 simulations were conducted across 6 different cache sizes on 329 individual workloads contained within the 5 sets of workloads. Each individual workload represents an entire day of storage I/O activity from one storage system.

Cache Configurations. For evaluating caching algorithms, the primary metric of significance is cache hit-rate. To compare the relative performance of various caching algorithms, we chose caches that are sized relative to the size of each workload's *footprint*, i.e., all the unique data items accessed.

6.2 Time and Space Overheads

CACHEUS maintains roughly $2N$ pieces of metadata where N is the size of the cache, using N units to track cache-resident items and N additional units to track items that are in history. This is equivalent to state-of-the-art algorithms such as ARC and LIRS which each maintain approximately

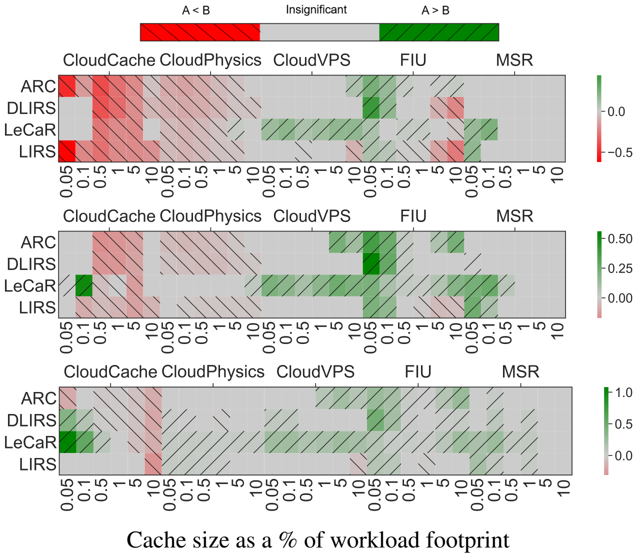


Figure 8: Paired t-test analysis to understand the difference in performance between (A) Cacheus vs. (B) Other. *The three panels compare four "Other" algorithms (i.e., ARC, DLIRS, LeCaR, and LIRS) against the following variants of* Cacheus*:* **Top:** *C1,* **Middle:** *C2* **Bottom:** *C3. Green colors indicate that the* Cacheus *variant was significantly better, red colors indicate that the* Cacheus *variant was significantly worse, and the gray color indicates no significant difference. Brighter green and red colors indicate higher effect sizes. Effect sizes were computed using Cohen's d-measure.*

N items of additional metadata to track a limited history. Cacheus merges the additional metadata of individual experts (e.g. ARC, SR-LRU, and CR-LFU) and its own history for an effective size of N history items. Specifically, when SR-LRU and CR-LFU are used as experts in Cacheus, the history metadata of each algorithm is reduced to $N/2$ for a total of N history metadata. The computational overhead of Cacheus when it uses SR-LRU and CR-LFU as experts is bound by the computational overhead of LFU — $O(logN)$. This time complexity can be improved with a more careful implementation for LFU [30].

6.3 Statistical Analysis

We performed a broad palette of paired t-tests to evaluate the three Cacheus variants against the strongest competitors across $17,766$ experiments. A p-value threshold of 0.05 was used to judge statistical significance outcomes from the t-tests. Effect sizes were computed using the Cohen's d-measure, which measures the number of standard deviations that separate the two means. Figure 8 presents the results of our t-test analysis for the three Cacheus variants.

To summarize the findings, C3 is distinctly the best performing algorithm in 47% of the workload-cache combinations with effect sizes ranging from 0.2 to 1.08 in 28% of the positive cases, is indistinguishable from the best performing state-of-the-art algorithm in about 40%, and is worse than the best performing algorithm for the remaining 13% with negative effect sizes of up to 0.31. For the 13% of the cases where an algorithm other than C3 is found to be distinctly better, no single algorithm is found to be consistently the best, indicating that C3 is an excellent choice overall. C2 is better than the best performing state-of-the-art in about 26% of the combinations with effect size in the range of 0.2 to 0.56 in 55% of the positive cases, indistinguishable from the best in 48% of the combinations, and worse in the remaining 27% of the cases with negative effect size of up to 0.17. C1 is better than the best performing state-of-the-art in about 20% of the combinations with effect size from 0.2 to 0.44 in 22% of the positive cases, indistinguishable from the best in 41% of the combinations, and worse in the remaining 39% of the cases with negative effect size up to 0.62.

We also analyze the best and worst case improvements in hit-rate for the best-performing Cacheus algorithm, C3. Figure 9 presents the absolute difference in hit-rate for C3 relative to its competitors — ARC, LIRS, DLIRS and LeCaR, shown as a set of violin plots. Violin plots have the advantage of showing summary statistics, including the median, the quartiles and outliers along with a density shape for each Y-value [11]. The worst case degradation of 15.12% is observed with the MSR workload with cache size at 5% when compared against DLIRS. The best case improvement of 38.32% is observed with CloudPhysics workload at a cache size of 10% when compared against ARC.

6.4 Understanding Cacheus

We focus our investigation and comparative analysis of Cacheus against 3 of the best performing candidates: *(i)* **state-of-the-art adaptive** algorithm (ARC), *(ii)* **state-of-the-art scan-resistant** algorithm (LIRS); we do not consider DLIRS, its adaptive variant, which performs worse than LIRS on average, and *(iii)* **state-of-the-art machined-learned** algorithm (LeCaR), a predecessor of Cacheus. To understand the performance advantage of Cacheus, we measured hit-rates over time averaged across a sliding window equal to the size of the cache. In particular, we examine the performance for the webmail day 16 workload from the FIU trace collection. As shown in Figure 11, this workload includes a combination of multiple workload primitive types. For example, we observe a long *scan* for approximately 2 hours (between 6:30 and 8:30) followed by repeated accesses over a sub-set of the items (i.e., *churn*) for more than half the total workload duration.

6.4.1 Cacheus C3 vs ARC

Figure 10 shows the performance over time for the four algorithms tested on webmail (day 16) workload. The total hit-rates for ARC, LIRS, LeCaR and C3 are 30.08%, 40.71% and 42.08% and 43.95% respectively. The leftmost plot shows the comparison against ARC. Initially a set of items that include a single large scan are accessed until the burst

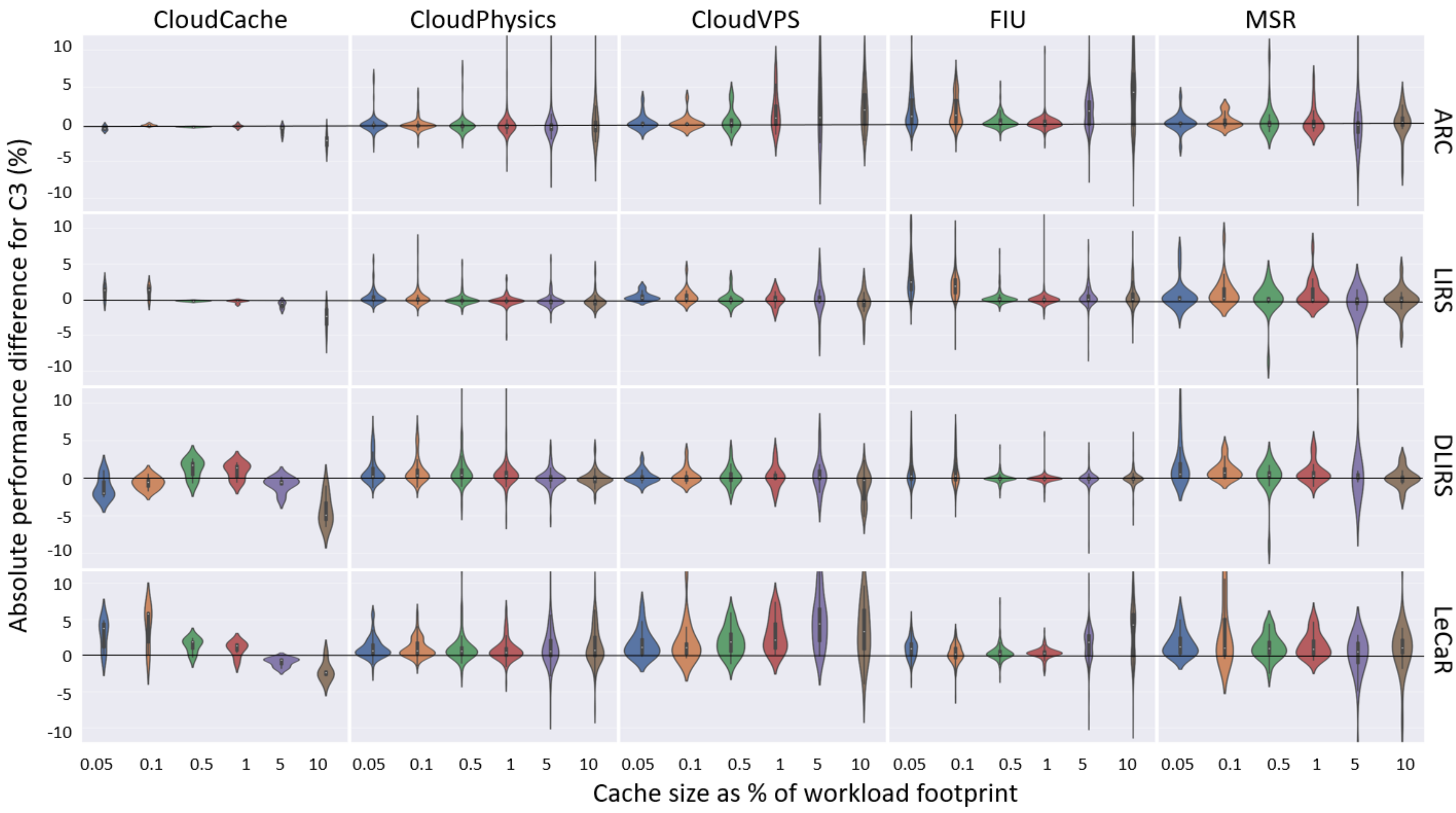


Figure 9: Absolute cache hit-rate difference distributions using CACHEUS algorithm C3 across workloads and cache sizes. *The figure displays four rows of violin plots with each row comparing the performance of CACHEUS C3 with the baselines of each row being the performance of ARC, LIRS, DLIRS, and LeCaR from top to bottom. Positive Y-values indicate that CACHEUS algorithm C3 performed better in comparison. The Y-range is truncated to the range (-12,12) for better readability, but with minimal loss of information. The violin plots show the median as a white dot, the range from the first to third quartile as a thick bar along the violin's center line, and a thin line showing an additional 1.5 times the interquartile range. It also shows the density shape at each Y-value [11], making these plots very informative.*

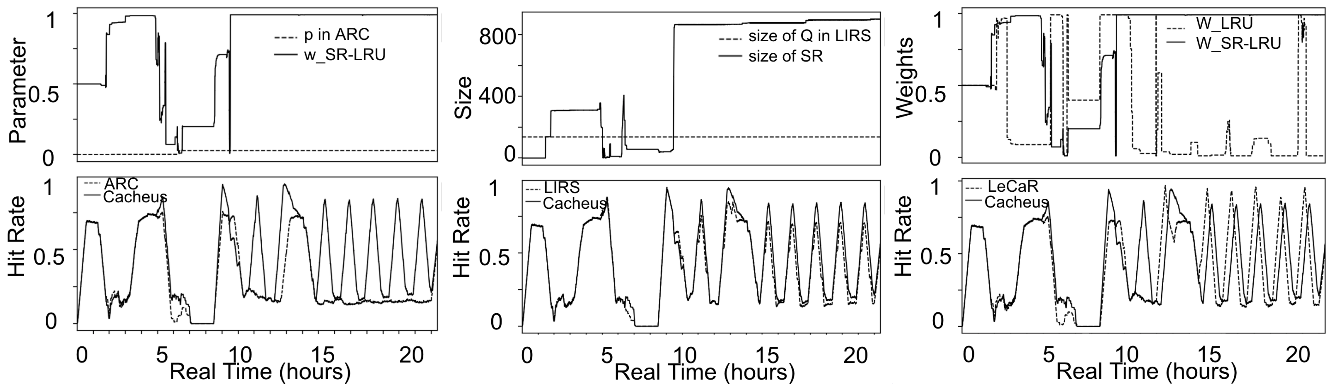


Figure 10: Detailed comparison of CACHEUS **against ARC (left), LIRS (middle) and LeCaR (right) for the webmail (day** 16) **workload.** *The lower plots show cache hit-rate computed using a sliding window equal to the size of the cache. The upper plot shows the internal parameter for each algorithm (p in ARC is normalized with respect to the size of the cache). Cache size is set to 10% of the workload footprint (*54*MB). The hit-rate improvements for* CACHEUS *with respect to ARC, LIRS, and LeCaR are* 46.11%, 7.95% *and* 4.4% *respectively.*

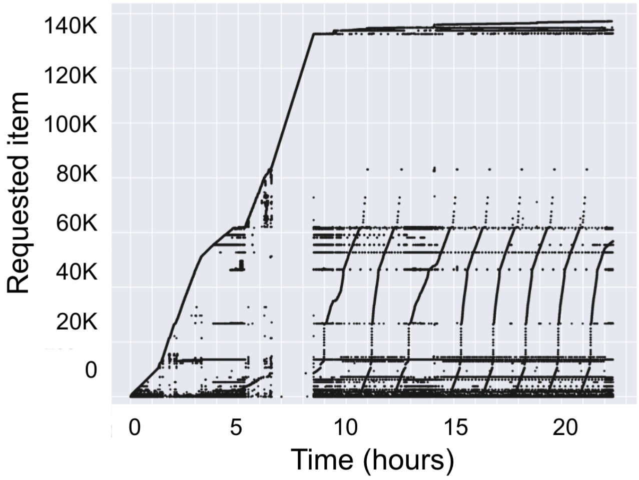


Figure 11: Access pattern for the webmail (day 16) **workload from the FIU trace collection.**

of unique accesses creates zero hits. C3 is able to maintain the previous working set in the cache, enabling it to generate hits post scan. ARC protects $T2$ as dictated by its internal parameter p close to 0 in an attempt to minimize cache pollution. Right after the scan finishes, a sequence of 8 churn phases starts to populate the cache. To respond effectively, ARC starts to increase the size of $T1$ to accommodate the new incoming items. However, the increments in p grow $T1$ slowly in steps of 1. In particular, during this entire trace ARC maintains its shadow list $B2$ empty by avoiding evictions from $T2$, even during churn. This behavior negatively impacts ARC's performance for the last 5 churn periods.

6.4.2 CACHEUS C3 vs LIRS

The center plot in Figure 10 compares LIRS and C3 using the same workload. LIRS uses a fixed size of Q equal to 1% of the cache size (138 items in this experiment). During the *scan* period, LIRS uses Q as a filter without affecting the working set previously populated in cache. For the *churn* phases, LIRS is able to keep in cache the important items by relying on its low-interference items in *S*. In particular, for *churn* phases, LIRS will always miss the first hits on the initial portion of the churn because these items will stay in cache for a short period of time. On the other side, C3 starts with a small size in *SR* to protect against the initial scan. During churn periods, C3 is able to dynamically accommodate new items in *SR* by increasing its size and therefore relaxing the scan protection. Finally, LIRS's ability to adapt to *LRU-friendly* workloads is limited by the size of *Q*.

6.4.3 CACHEUS C3 vs LeCaR

Finally, the rightmost plot in Figure 10 compares LeCaR and C3. The upper plot shows the weights for LRU and SR-LRU in LeCaR and C3 respectively, both initialized to 0.5. During the *scan* phase, LRU and SR-LRU get penalized due to the drop in performance until new hits in cache make them increase again. Even though choosing LFU is the right decision for LeCaR during *churn* phases, the delay in doing so prevents LeCaR of accumulating more hits than C3. In particular, C3 is able to capitalize during one *churn* period during the 11 hour while maintaining good performance for the last 7 hours of the workload. Most interestingly, towards the end when LeCaR mostly uses LFU, C3 exclusively relies on SR-LRU during *churn* periods. This is due to the fact that while SR-LRU was designed to handle *scan* phases, it also implements a way to avoid confusing churn periods with scan. This is done by marking items entering *SR* for the first time as *new* and keeping track of such items in H. If a *new* item is accessed again while in H, SR-LRU quickly corrects itself to disable scan protection.

7 Related Work

Past work on utilizing multiple experts within a cache replacement algorithm include ACME [1] and the follow up work on designing a master policy [10] which learned the weights of 12 distinct experts and used these to make eviction decisions. Since then, algorithms such as ARC [23], LIRS [13], DLIRS [20], and LeCaR [34] were developed and are considered the state-of-the-art.

CACHEUS builds on the successes of LeCaR. It improves upon LeCaR in a few ways. First, while LeCaR argued for using the classic LRU and LFU, CACHEUS demonstrates the importance of using more sophisticated experts. Second, CACHEUS simplifies LeCaR by identifying and eliminating redundant aspects of its machine-learning mechanism. Third, it creates a fully-adaptive version that is also lightweight. Finally, new lightweight experts, SR-LRU and CR-LFU improve upon LeCaR's experts to address two new workload primitive types, *scan* and *churn*. With these improvements, CACHEUS performs better than LeCaR as well as other state-of-the-art algorithms such as ARC, LIRS, and DLIRS.

SR-LRU is inspired by both ARC and LIRS. One important distinction between ARC and SR-LRU is that ARC evicts from either $T1$ or $T2$, while SR-LRU only evicts from a single spot: *SR*. Another distinction is SR-LRU's use of tags instead of separate histories ($B1$ and $B2$ in ARC) in order to enable reasonable adaptiveness. As to LIRS and its adaptive version, DLIRS, SR-LRU differs from these in the separation of history from internal partition/stack data structures, and its use of tags to determine relevance of items in history instead of explicitly pruning obsolete history items.

Recent works on adaptive caching include Least Hit Density (LHD) [4] which focuses on predicting an object's hits-per-space-consumed to determine evictions in a variable-sized object environment. LHD focuses on variable-sized caches of key-value stores or CDNs and was therefore not evaluated against the state of the art storage caches such as ARC and LIRS [4]. Like the state-of-the-art storage caching algorithms, CACHEUS is designed for a fixed-sized object caching environment and uses a novel reinforcement learning technique that engages exactly two complementary experts for significantly improving caching decisions.

8 Conclusions

Consistently high-performing caching continues to represent a fascinating, yet elusive, goal for storage researchers. CACHEUS serves this goal by creating a new class of lightweight and adaptive, machine-learned caching algorithms. The CACHEUS framework allows the use of exactly two, ideally complementary, experts to guide its actions. CACHEUS using the proposed new experts, SR-LRU and CR-LFU, is the most consistent algorithm for a range of workload-cache size combinations. Furthermore, CACHEUS enables easily combining a state-of-the-art caching algorithm such as ARC and LIRS with a complementary expert such as LFU to better handle a wider variety of workload primitive types. We believe that ML-based frameworks for utilizing caching experts holds great promise for improving the consistency and effectiveness of caching systems when handling production workloads. CACHEUS sources can be downloaded at `https://github.com/sylab/cacheus`.

Acknowledgments

We would like to thank the reviewers of this paper and our shepherd Ken Salem for insightful feedback that helped improve the content and presentation of this paper substantially. This work was supported in part by a NetApp Faculty Fellowship, and NSF grants CCF-1718335, CNS-1563883, and CNS-1956229.

References

[1] I. Ari, A. Amer, R. B. Gramacy, E. L. Miller, S. A. Brandt, and D. D. Long. ACME: Adaptive caching using multiple experts. In *WDAS*, pages 143–158, 2002.

[2] D. Arteaga and M. Zhao. Client-side flash caching for cloud systems. In *Proceedings of International Conference on Systems and Storage (SYSTOR)*, 2014.

[3] R. Battiti. Accelerated backpropagation learning: Two optimization methods. *Complex systems*, 3(4):331–342, 1989.

[4] N. Beckmann, H. Chen, and A. Cidon. LHD: Improving cache hit rate by maximizing hit density. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 389–403, 2018.

[5] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[6] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[7] L.-W. Chan and F. Fallside. An adaptive training algorithm for back propagation networks. *Computer speech & language*, 2(3-4):205–218, 1987.

[8] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12 (Jul):2121–2159, 2011.

[9] G. Einziger, O. Eytan, R. Friedman, and B. Manes. Adaptive software cache management. In *Proceedings of the International Middleware Conference*. ACM, 2018.

[10] R. B. Gramacy, M. K. Warmuth, S. A. Brandt, and I. Ari. Adaptive caching by refetching. In *Advances in Neural Information Processing Systems*, pages 1489–1496, 2003.

[11] J. L. Hintze and R. D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52 (2):181–184, 1998.

[12] S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen. Improving flash-based disk cache with lazy adaptive replacement. *ACM Transactions on Storage*, 12(2):8:1–8:24, Feb. 2016.

[13] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the ACM Sigmetrics Conference (SIGMETRICS)*, 2002.

[14] A. Khachaturyan, S. Semenovskaya, and B. Vainstein. Statistical-thermodynamic approach to determination of structure amplitude phases. *Sov. Phys. Crystallography*, 24(5):519–524, 1979.

[15] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598): 671–680, 1983.

[16] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST, 2010.

[17] R. Koller, A. Verma, and R. Rangaswami. Generalized ERSS tree model: Revisiting working sets. In *Proceedings of IFIP Performance*, November 2010.

[18] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, Dec. 2001.

[19] A. Li, O. Spyra, S. Perel, V. Dalibard, M. Jaderberg, C. Gu, D. Budden, T. Harley, and P. Gupta. A generalized framework for population based training. *arXiv preprint arXiv:1902.01894*, 2019.

[20] C. Li. DLIRS: Improving low inter-reference recency set cache replacement policy with dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference (SYSTOR)*, 2018.

[21] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Information and computation*, 108(2): 212–261, 1994.

[22] G. Loomes and R. Sugden. Regret theory: An alternative theory of rational choice under uncertainty. *The economic journal*, 92(368):805–824, 1982.

[23] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[24] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling Down Peak Loads Through I/O Off-Loading. *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, December 2008.

[25] V. Plagianakos, G. Magoulas, and M. Vrahatis. Learning rate adaptation in stochastic gradient descent. In *Advances in convex analysis and global optimization*, pages 433–444. Springer, 2001.

[26] H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, pages 400–407, 1951.

[27] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes and node granularity issues for large-scale multiprocessors. In *Proceedings of the International Symposium of Computer Architecture (ISCA)*, 1993.

[28] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[29] R. Santana, S. Lyons, R. Koller, R. Rangaswami, and J. Liu. To ARC or Not to ARC. In *Proceedings of the USENIX Workshop on Hot Topics in Storage Systems (HotStorage)*, 2015.

[30] K. Shah, A. Mitra, and D. Matani. An O(1) algorithm for implementing the LFU cache eviction scheme. `http://dhruvbird.com/lfu.pdf`, August 2010.

[31] W. L. Smith. Regenerative stochastic processes. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 232(1188):6–31, 1955.

[32] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[33] Storage Networking Industry Association. The SNIA's I/O Traces, Tools, and Analysis (IOTTA) Repository. `http://iotta.snia.org/`.

[34] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan. Driving Cache Replacement with ML-based LeCaR. In

Proceedings of the USENIX Workshop on Hot Topics in Storage Systems (HotStorage), June 2018.

[35] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient MRC construction with SHARDS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[36] Z. Wang, F. Hutter, M. Zoghi, D. Matheson, and N. de Feitas. Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, 55:361–387, 2016.

[37] C. C. Yu and B. D. Liu. A backpropagation algorithm with adaptive learning rate and momentum coefficient. *Proceedings of the International Joint Conference on Neural Networks*, 2:1218 – 1223, 2002.

FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays

Tianyang Jiang[†], Guangyan Zhang[†*], Zican Huang[†], Xiaosong Ma[‡],
Junyu Wei[†], Zhiyue Li[†], Weimin Zheng[†]
[†]*Tsinghua University,* [‡]*Qatar Computing Research Institute, HBKU*

Abstract

The use of all-flash arrays has been increasing. Compared to their hard-disk counterparts, each drive offers higher performance but also undergoes more severe periodic performance degradation (due to internal operations such as garbage collection). With a detailed study of widely-used applications/traces and 6 SSD models, we confirm that individual SSD's performance jitters are further magnified in RAID arrays. Our results also reveal that with SSD latency low and decreasing, the software overhead of RAID write creates long, complex write paths involving more drives, raising both average-case latency and risk of exposing worst-case performance.

Based on these findings, we propose *FusionRAID*, a new RAID architecture that achieves *consistent, low latency on commodity SSD arrays*. By spreading requests to all SSDs in a shared, large storage pool, bursty application workloads can be served by plenty of "normal-behaving" drives. By performing temporary, replicated writes, it retains RAID fault-tolerance yet greatly accelerates small, random writes. Blocks of such transient data replicas are created in stripe-ready locations based on RAID declustering, enabling effortless conversion to long-term RAID storage. Finally, using lightweight SSD latency spike detection and request redirection, FusionRAID avoids drives under transient but severe performance degradation. Our evaluation with traces and applications shows that FusionRAID brings a 22%–98% reduction in median latency, and a 2.7×–62× reduction in tail latency, with a moderate and *temporary* space overhead.

1 Introduction

The use of all-flash arrays (AFAs) has been increasing and projected to have a 400% market growth in the next five years [11]. For example, ANZ bank in Australia recently adopted an AFA solution with 400TB SSD arrays [31]. AFAs aggregate the IOPS and bandwidth of individual drives and compensate for SSDs' higher rate of uncorrectable errors [53] using parity-based fault-tolerance. So far, AFAs can support large numbers of SSDs (up to 5760) behind a single controller [20, 22, 49, 51, 52].

	Median latency (ms)	Avg. latency (ms)	P99 latency (ms)	Variance factor
HDD RAID (clean)	68.67	134.37	835.35	12.16
HDD RAID (aged)	69.18	133.61	826.77	11.95
SSD RAID (clean)	0.275	3.57	25.62	93.16
SSD RAID (aged)	0.307	14.11	221.03	719.96

Table 1: `Exchange` latency, HDD vs. SSD RAID

However, SSDs are less array-friendly than hard disks, which RAID [50] was initially designed for decades ago. Compared with single-disk accesses, RAID write significantly amplifies average write latency, thereby also delaying read requests. Also, SSD RAIDs have a much higher *tail-to-average latency ratio* than HDD ones [23, 38, 66].

We illustrate this by testing two software RAID5 arrays, built on Seagate HDDs and Intel commodity consumer SSDs. Table 1 lists the median, average, and 99 percentile (P99) latencies measured running the write-intensive `Exchange` trace from Microsoft Production Server Traces [55]. We also report the *variance factor* [68], the ratio of P99 to median latency. For each RAID, we test its *clean* and *aged* states, using the `fio` benchmark [4]. Here we adopt an existing aging method [35], writing the whole disk sequentially and then issuing random writes with total volume exceeding its capacity, to guarantee that each random write generates invalid pages.

Our results confirm that, though SSD RAID offers much smaller median latency (over 225× lower than HDD RAID), its worst-case performance deviates more from the norm, with a much higher variance level. The average latency, as a result, is much more amplified from the median with SSD than with HDD RAID. Second, the HDD RAID appears quite resilient to aging, with hardly any visible performance degradation. The SSD RAID, on the other hand, deteriorates significantly when aged, delivering a median latency of 11.6% higher, and P99 tail 8.6× higher. Such severe performance variability makes it difficult to ensure QoS to customers [18, 23, 24, 59, 69], potentially causing significant revenue losses [47]. This problem is not specific to Linux soft-

[*] Corresponding author: gyzh@tsinghua.edu.cn

	Median latency (ms)	**Avg. latency (ms)**	**P99 latency (ms)**	**P999 latency (ms)**
SSD 0	0.049	0.68	0.42	24.40
SSD 1	0.049	1.26	0.46	**702.24**
SSD 2	0.050	0.63	0.39	30.16
SSD 3	0.049	1.64	0.53	**895.02**
SSD 4	0.050	1.71	0.64	**827.91**

Table 2: `Exchange` latency, individual aged SSDs within RAID

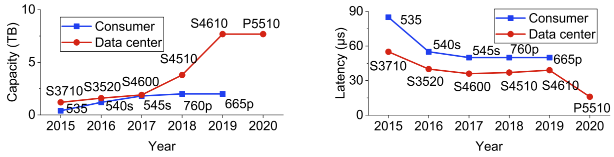
ware RAID overhead: our measurement also shows a hardware controller (LSI MegaRAID 9260-8i) producing very similar average latency (3.4ms) to Linux software RAID (3.6ms) on the same SSDs.

We further examine the latency distribution of individual SSD drives within the 4+1 RAID5 array, with results listed in Table 2 (aged state only). Comparing results from both tables, one sees that when we group SSDs into RAID, for the gain in space and bandwidth aggregation, we may be trading off individual request's processing speed. Note that with the `Exchange` workload (details in Table 4), considering the `MD` default 64KB stripe unit size, the majority of requests would each land on a single drive. However, the added complexity of parity updates not only generates more work but involves more drives, rendering a P99 latency nearly 400× higher on RAID than on individual SSDs for the same workload. The last column in Table 2 highlights a challenge with SSD RAIDs: three of the drives appear to experience garbage collection (GC) during our 15-minute trace execution and have a P999 latency over 23× higher than the other two. With highly coupled operations across multiple drives, isolated tail latency from a single drive affects more requests with RAID, making the entire array more vulnerable to performance anomalies.

In this work, we first conduct a comprehensive study to investigate the sources of SSD RAID latency. More specifically, we (1) examine 5 real-application workloads on modern SSDs, plus 3 workloads from widely used storage trace repositories, and systematically characterize the behavior of their mixes in fine time granularity, (2) perform a detailed analysis of the RAID write path and identify the software overhead, which has a dependency on the I/O performance of member disks, and becomes a major component in request latency, in both average and worst-case scenarios, and (3) conduct detailed profiling on 4 consumer- and 2 datacenter-grade SSDs to characterize the device-side degradation due to flash internal activities, finding both types plagued by severe latency spikes with clearly identifiable amplitude and long duration.

We then propose a new RAID architecture, FusionRAID, designed to simultaneously reduce the average- and worst-case latencies of SSD RAID, especially for latency-critical applications. FusionRAID runs on commodity SSD arrays without requiring any special hardware support or FTL modification, instead relying on three key techniques:

- flat resource sharing across an SSD pool to utilize available I/O concurrency in serving bursty application I/Os,
- shortened write operations that use temporary, replicated writes as a prelude to long-term RAID storage, yet carefully placed "in-stripe" so that one replica can be directly converted without data migration, and
- a lightweight latency spike detection and request redirection mechanism that allows requests (both read and write) to sidestep SSDs under severe performance degradation.



(a) Capacity

(b) Latency

Figure 1: Trends in technical specification of Intel SSDs (no consumer SSDs issued in 2020)

We argue that our solution, which temporarily trades space for performance, aligns well with current hardware trends. Figure 1(a) and 1(b) portray changes in capacity and latency, using historical data we gathered on Intel SSD models [29]. In the past years, SSDs have become much larger (especially datacenter models) and more affordable, with smaller improvement in latency. FusionRAID uses more space to quickly absorb small, random writes (with little long-term extra space overhead). In return, it makes SSD RAIDs *several times faster on average* and *orders of magnitude faster in tail cases*, as shown in our real-system evaluation. In addition, FusionRAID proposes a new way to utilize emerging large AFAs (90 SSDs or more), such as the NetApp AFF [49], EMC VMAX [20], and Fujitsu ETERNUS [22] series.

Compared with other systems that address FTL-induced latency during writes, FusionRAID considers SSDs as black boxes, without assuming SSD internal information/control. Its novelty lies in several aspects. First, to our knowledge, this is the first work that applies RAID declustering [3, 25, 46, 71] to SSD arrays. It leverages Latin-square based deterministic addressing [71], but with a significant extension to perform explicit block mapping for its two-phase writes and out-of-place updates. Second, FusionRAID adopts a new "in-position conversion" mechanism from its "replicated" to "RAID" area, removing the extra copying required by existing hybrid systems [21, 65]. Our intensive analysis reveals that SSD spikes possess clearly identifiable amplitude and long duration, making reactive methods feasible. FusionRAID's I/O redirection, based on constant spike-detection, eliminates the need for periodic probe I/Os, used by previous methods [23, 66, 69].

2 SSD RAID Latency Source Study

2.1 Workload I/O Characteristics

I/O requests from applications are often issued in a bursty manner, so the instantaneous bandwidth of a single workload varies significantly. When multiple workloads co-execute on a storage system, the instantaneous aggregated bandwidth is often dominated by a small number of workloads, as they read/write a large amount of data while others access a little, in a short period of time.

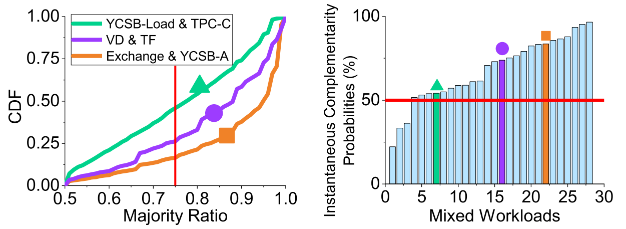
To assess the load behavior of representative applications, we collect five block-level traces from multiple popular data-intensive workloads running individually on SSDs. We also include three publicly available traces for diversity. The eight workloads are characterized in Table 4. Then, we examine the mixes of those eight workloads in fine time granularity.

We capture all five traces from our testbed (more details in Section 5). Three of them are from running representative standard YCSB [16] workloads using RocksDB [10], a popular KV store: `YCSB-A`, `YCSB-B`, and `YCSB-Load` (specifications in Table 5). The RocksDB database size is 40GB, with 4KB KV pairs. Another latency-sensitive application trace is collected from the `TPC-C` benchmark, on a 77GB MySQL database. In addition, we trace `TensorFlow` (`TF`), which reads training datasets and checkpoints a CNN model periodically in search of better accuracy. Finally, we include three traces from the SNIA repository [58]: `VirtualDesktop` (`VD`) [40], the only recent trace from server environments, plus `Exchange` and `Proxy`, the heaviest two among a total of 49 traces within the Microsoft and SPC trace collections [30, 55–57].

Next, we carefully examine the microscopic features of mixed workloads by analysing the instantaneous bandwidth at millisecond granularity using the aforementioned 8 traces. In each 1ms timeslot, we partition a workload mix into a *giant* and a *dwarf* set, each containing the same number of workloads, with any workload in the former heavier than all in the latter. We define the ratio between the total instantaneous bandwidths of the giant set and the entire mix as *majority ratio*, R_{maj}. We claim that a workload mix possesses *instantaneous complementarity* in a timeslot with $R_{maj} > 0.75$, where the dwarf set can lend spare resources to the giant one.

Based on their average throughput, we coarsely divide the 8 traces into two groups, "heavy" and "light". We inspect all the three types of 2-workload mixes: light-light, light-heavy, and heavy-heavy. Figure 2(a) shows the CDF of R_{maj} across all timeslots for such mixes. Instantaneous complementarity appears quite frequently in the light-heavy (`Exchange` +`YCSB-A`) and light-light (`VD` +`TF`) workload mixes, making 83.6% and 73.9% of all timeslots, respectively. Even with the heavy-heavy mix (`YCSB-Load` +`TPC-C`), 54.1% of timeslots have instantaneous complementarity.

With the 8 workloads, we enumerate all 2-workload mixes and find that 25 out of the 28 have instantaneous complementarity in over half of the timeslots, with an average ratio across all 28 mixes sitting at 67.8% (Figure 2(b)). Across all 70 4-workload mixes, this average ratio of timeslots possessing instantaneous complementarity rises to 91.4%.



(a) CDF of R_{maj}

(b) Ratio of timeslots w. instantaneous complementarity

Figure 2: 2-workload mix analysis, the left showing 3 selected mixes, the right all 28 mixes

Implications Our analysis shows that when workloads run together, the chances of their instantaneous complementarity are quite high. This suggests that modern commodity storage servers, easily managing dozens of devices or more, can serve concurrent workloads better in an "all-for-all" model, rather than being held back for the fear of inter-workload performance interference. Involving all disks in serving the busier workloads at the moment alleviates their queue wait time, a major source of application-induced tail latency.

2.2 Write Overhead in SSD RAID

Parity-based RAID is unfriendly to write-intensive workloads, especially for those dominated by random small writes. The inherent read-modify-write logic makes partial-stripe writes go through a lengthy sequence of ordered operations of reads, calculation, and writes. Optimizations targeting bandwidth, such as the mechanism used in the Linux `MD` software RAID driver that postpones submission of writes in anticipation that subsequent requests fall into the same stripe, may hurt latency-sensitive applications.

Figure 3 illustrates this with a comparison between (4+1) RAID-5 arrays using three types of devices: Intel 545s SATA SSDs, Seagate 7200RPM SATA HDDs, and RAM disks. All are software RAID arrays through `MD`. We run the Microsoft Enterprise `Exchange` workload and show the breakdown of write latency across operations: read, xor, and write, with the rest categorized into software overhead. The left plot describes all requests, while the right one focuses on the 1% requests with the highest latency. Numbers at the top of the bars give the average latency values for each group.

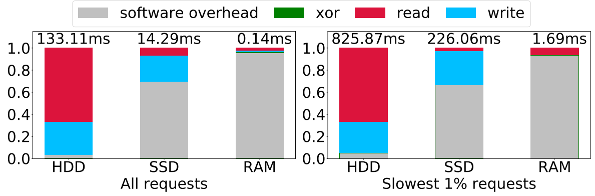


Figure 3: Write latency breakdown

With this consumer-grade SSD, software overhead already takes over as the major component of write latency, while it

	Model	Capacity (GB)	Read/Write latency (µs)	Read/Write bandwidth (MB/s)	Release year	Price ($/GB)
A	Intel 545s	240	50/60	550/500	2017	0.18
B	Intel 535	256	80/80	540/490	2015	0.15
C	Toshiba q200	240	73/36*	550/510	2017	0.20
D	Sandisk plus	240	44/193*	530/440	2017	0.17
E	Intel D3-S4510	240	36/37	560/280	2018	0.33
F	Intel DC S4500	240	36/36	500/190	2017	0.29

Table 3: Evaluated SSDs (all latency vendor specified except those marked with *)

is almost invisible with the HDD. On average, the SSD RAID tested spends 2.9× time on software overhead than on writing. This software overhead, however, involves synchronization interleaving with I/Os and is not independent of the read/write cost: with read/write cost nearly trimmed, software overhead dominates the RAM disk RAID latency, but its absolute cost is nearly two orders of magnitude smaller than on the SSD RAID. Software overhead also makes the slowest 1% requests suffer 10× the average latency, due to factors such as thread context switching and request queuing (at the block layer and the host-side dispatch queue).

As a side note, though software overhead remains the leading category, for the slowest 1% requests shown in Figure 3, SSD writes also contribute significantly to the SSD RAID tail latency, costing 20.7× the average write overhead. Section 2.3 gives a detailed discussion on this issue.

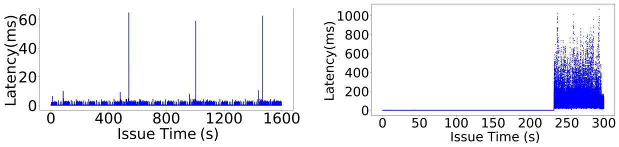
Implications Unlike an HDD array, an SSD RAID has I/O latency dominated by software overhead. Unlike a RAM disk array, it sees such overhead prolonged by actual I/Os and their coordination across disks. This suggests that a shorter write path, with fewer dependencies, may greatly reduce SSD RAID latency, both under average and worst-case scenarios.

2.3 Pathological Latency Spikes of SSDs

SSDs are known to have performance anomaly due to background I/O activities obscure to users [17, 23]. Major sources of performance variability include (1) background maintenance activities inside SSDs, in particular, garbage collection [38, 66, 69], and (2) on-SSD DRAM write buffer flushing [1]. While the existence of performance jitters is well-known [17, 23], in this paper, we quantify the distribution of their magnitude as well as duration, on multiple consumer- and datacenter-grade SSDs.

Table 3 lists basic specifications of the six commercial off-the-shelf SSDs used in our experiments, the first four being consumer-grade and the last two datacenters (DC) drives. Due to space limit, below we summarize our chief findings.

Consumer SSDs First, when running sequential writes on clean drives, all consumer models tested possess clearly periodic latency spikes with a duration between 3ms and 12ms, reaching 5-20 times of their average latency. With GC excluded under this strictly sequential workload, we attribute the regular latency spikes to on-disk DRAM buffer flushes [12]. We found these flushes block read requests as well.



(a) Clean SSD-A, seq., limited space

(b) Aged SSD-E, random

Figure 4: Sample spike behavior in write tests on two SSDs (write request size at 64KB)

To repeatedly incur GC, we repeat the sequential write test but write within a 2GB logical space. We find GC produces spikes both much taller (height around 21×) and long-lasting (duration around 18×) than those incurred by buffer flushes, as illustrated in Figure 4(a). Our zoom-in analysis of I/O behaviour shows that requests are blocked during GC inside SSDs and completed immediately in a batch once GC is over. Meanwhile, there are obvious intervals (10× spike duration) between spikes due to enough spare blocks after a GC. Finally, aged SSDs see more severe and frequent spikes. The reason is likely that when the spare blocks inside SSDs run out, GC incurs more data migration and block erasures [33].

DC SSDs Datacenter SSDs behave differently. First, periodic latency spikes are not seen under sequential write workloads, unlike with consumer SSD. This is likely because DC SSDs usually have multiple cores and an optimized FTL for better coordination between background and foreground I/Os. Note such optimization may be achieved at the cost of lowering write throughput [54]: Table 3 does show the write bandwidth of the two DC SSDs at around half of that offered by cheaper consumer models.

Under heavy write workloads, however, DC SSDs cannot hide the impact of GC. Figure 4(b) shows spikes observed with random 64KB writes on SSD-E, which incurs GC faster. Under such a constant write workload, once GC is triggered, the I/O latency goes far beyond 100ms, incuring performance degradation lasting around 60s. Furthermore, spikes cannot be divided strictly, instead a new spike often arrives before the previous one ends, differing from consumer SSDs.

Implications Our profiling confirms that both consumer and datacenter SSDs suffer from severe latency spikes. Coupling such duration with an amplitude that significantly deviates from the norm, these spikes can and should be detected at runtime, to redirect incoming requests to other devices in the SSD pool. Moreover, spikes from consumer and DC SSDs behave differently, which should be handled carefully.

3 Approach Overview

We propose a new SSD RAID architecture, FusionRAID. It reduces *both average-case and tail latencies*, with solutions targeting the three problems observed in Section 2.

Design Rationale To ease innate request bursts in workloads, FusionRAID *spreads requests to all disks* in a storage pool (such as a large commodity SSD enclosure) hosting multiple RAID volumes. Though most of their individual I/O requests can be answered by a small subset of disks, applications often have severe load bursts that directly lead to tail latency. FusionRAID trims such workload-induced tail latency by smoothing the bursts to all disks in an SSD enclosure using RAID declustering [3, 25, 46, 71]. In multi-tenant settings, this automatically lends resource elasticity to individual workloads' varying intensity.

To reduce the software overhead and inter-disk dependence in RAID writes, FusionRAID employs *replicated writes as a prelude to RAID writes*, with data lazily converted later to the more space-efficient RAID organization for long-term storage. Before such conversion, block replicas ensure the same level of fault-tolerance as the specified RAID level. *E.g.*, FusionRAID writes two copies of a block for a RAID5 volume, and three for RAID6. Doing so shortens the critical path in writes by postponing and in some cases even avoiding the long and interference-prone parity updating process. Consequently, the simpler, more independent operations of such replicated writes deliver lower (and *far more consistent*) latency. To further reduce the conversion overhead, FusionRAID places replicated blocks in a "stripe-ready" manner, in positions where sets of blocks already compose stripes (minus parity data) according to the RAID declustering algorithm adopted, to minimize data copying.

Finally, to sidestep SSDs undergoing temporary performance degradation, FusionRAID constantly watches each SSD's performance behavior to *detect temporarily unresponsive SSDs*. To this end, it uses a lightweight spike detection mechanism that issues no extra I/Os and requires no SSD internal knowledge. Uniquely enabled by RAID declustering on large SSD pools, FusionRAID easily *redirects* writes to unaffected drives, which likely remain in the majority at any given time. For reads, it could also select the less affected replica, or use existing approaches proposed by systems like ToleRAID [23] to compute a block hosted by an unresponsive SSD using parity data.

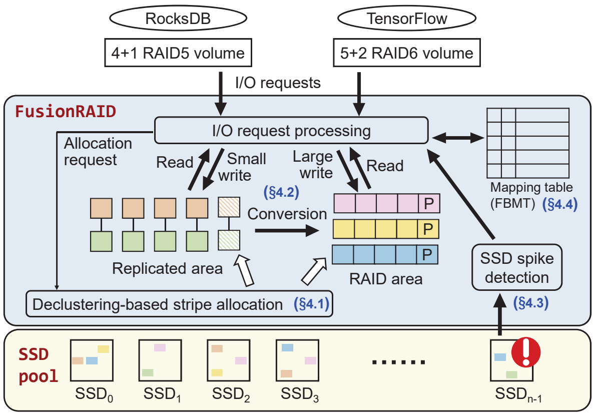


Figure 5: FusionRAID architecture

FusionRAID architecture Figure 5 illustrates the FusionRAID architecture. Multiple virtual RAID arrays (of different RAID configurations) share the same underlying pool containing dozens of commodity SSDs or more. The aggregate logical space of this SSD pool is partitioned into the *RAID* and *replicated areas*, intended for long-term, space-efficient storage and fast absorption of small, random writes, respectively. Note that there is no physical partitioning between these two *virtual* areas: actually, they are intentionally intermixed for fast conversion from the replicated to RAID storage (detailed discussion in Section 4.2). Note that though our discussion/evaluation uses RAID5 in this paper, FusionRAID applies to other RAID organizations, *e.g.*, by increasing the replication degree in the replicated area to 3 for RAID6.

Internally, FusionRAID employs a mapping table (FBMT in Figure 5) for each virtual RAID volume, to maintain the mapping of a per-volume logical block number to a FusionRAID internal logical block number (§4.4). The latter can subsequently be mapped to a logical block on a certain SSD using a deterministic RAID declustering strategy based on MOLS [71] (§4.1). For replicated writes, FusionRAID builds a list of available block pairs from a pair of stripes, enabling low-cost replicated-to-RAID conversion (§4.2).

In addition, FusionRAID performs real-time SSD latency spike detection by monitoring SSD latencies and includes the results in its decision making. In Figure 5, the last SSD is marked unresponsive and will be avoided in both reads and writes whenever possible (§4.3).

4 FusionRAID Design

4.1 Storage Organization

FusionRAID organizes the space across dozens or more SSDs, to serve bursty I/Os from concurrent workloads and provide ample alternative choices among drives to avoid using those under transient performance degradation. It does so by utilizing RAID declustering [3, 25, 46], distributing RAID stripes in a balanced way to larger arrays. The novel challenges here, unaddressed by existing RAID declustering techniques, are to design efficient and flexible support for *partial-stripe writes* to enable fast absorption of small, random writes, as well as to detour around temporarily slow drives.

To this end, FusionRAID employs a storage organization with explicit block mapping, seamlessly manages two logical areas (for replicated and striped writes), and requires low metadata space overhead. Even the transient block replicas are stored in a "stripe-aware" manner, ready for the eventual RAID storage. This way, the two logical areas are fused together, further facilitating efficient conversion from replicated to striped storage, as well as request redirection.

FusionRAID introduces *Fusion logical address space*, an internal logical block layer between the user-perceived logical and the SSD logical block address spaces, as illustrated

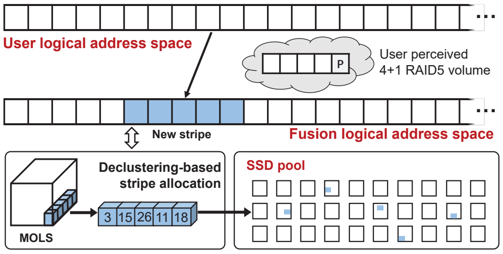


Figure 6: FusionRAID stripe allocation

in Figure 6. The mapping from user-perceived to this internal space is maintained by a block mapping table (to be detailed in Section 4.4), one per user RAID volume, which supports dynamic mapping for out-of-place writes. The mapping from a Fusion logical block to a logical SSD block is done by RAID declustering, involving a static *mapping function* instead of mapping tables. Here different declustering strategies/functions could be plugged in, such as RAID50 and pseudo-random RAID [62].

In our implementation, FusionRAID adopts the deterministic mapping proposed by RAID+ [71]. As shown in Figure 6, each FusionRAID volume will follow a 3D template based on *Mutually Orthogonal Latin Squares (MOLS)*, which resembles a large Rubik's Cube filled with disk IDs. The mapping function uses simple calculation to map a certain stripe within a user RAID volume (such as a 4+1 RAID5 configured in cloud storage, shown in the figure) to a certain subset of disks in the pool, $\langle 3, 15, 26, 11, 18 \rangle$ in this example. The advantage of MOLS-based declustering lies in its *deterministic* and *guaranteed uniform* distribution of each volume's data to all n disks within a pool, as well as low metadata overhead.

4.2 Two-phase Write Operations

Now spreading each RAID volume to the entire SSD pool to better prepare for bursty I/Os, FusionRAID further simplifies the critical path for writes, to deliver lower and more consistent latency. This is done by its *two-phase writes*, using replication as a prelude to RAID storage. While large writes are directly written in RAID stripes, smaller requests are directed to the replicated area, where instead of RAID writes with parity computation and update, we simply write two copies of data. Doing so avoids the dependency brought by the read-compute-write process, and involves fewer SSDs (hence a lower chance of running into a spike). Replicated data can be converted in the background to the more space-efficient RAID stripes. This can be carried out lazily, delayed when space is abundant, when data are hot and updated often, or when the disk pool is busy handling requests.

Two-phase write itself is not new and has been adopted by multiple systems, such as AutoRAID [65], DiskReduce [21], and LDM [67]. FusionRAID differs from them with two key innovations targeting all-flash environments, to reduce both write amplification and conversion-induced background I/Os.

First, FusionRAID performs its two-phase writes *selectively*, saving larger write requests the detour as for them the benefit does not justify the cost: their relative software overhead in RAID writes is lower, while the extra write volume generated by their replication is higher.

For larger requests, FusionRAID includes additional optimizations to lower the software overhead in RAID writes. For partial-stripe writes, it allocates a new stripe and pads the blocks untouched by the request with zero, which allows it to write both data and parity (plus appropriate metadata updates) without performing reads. Hence with requests writing x out of $(n-1)$ data blocks in a RAID stripe, going with our RAID writes as described above would require I/O of roughly n blocks, while replication requires $2x$-block writes. Therefore we set the "break-even point", $\lfloor \frac{n}{2} \rfloor$ blocks, as a size threshold to classify write requests: requests larger than this threshold do not benefit from replicated writes and would follow the aforementioned RAID write workflow.

Second, rather than following the common practice of migrating data from the replicated to the RAID area (writing the full stripe, including both data and parity blocks), we carefully create the replicated blocks "in position", so that they already *form two valid stripes* according to FusionRAID's MOLS-based template. Parity blocks are also allocated and reserved in advance, thus upon conversion, one group of the replicated blocks can be directly recorded as RAID stripe data blocks after the pre-allocated parity block is properly filled. The other group can simply be discarded.

As described in Section 4.1, FusionRAID allocates stripes using RAID declustering for both RAID and replicated writes. While the former consumes one stripe at a time, the latter consumes a pair of blocks at a time, from a pair of stripes, to store two copies of the same block. For fault tolerance, these two blocks need to sit on two different SSDs.

To this end, FusionRAID performs best-effort pairing among available stripes, using the power-of-two-choices scheme [45]. It randomly picks 4 spare stripes, selects a pair with the fewest common disk IDs, and starts block pairing by listing common disk IDs (if any) among the two stripes. It then cycles through the list diagonally, producing non-conflicting pairs. *E.g.*, if two stripes both involve disks A, B, and C, we create block pairs on A-B, B-C, and C-A. The rest of the blocks, on non-overlapping disks, are trivial to pair. This flexible scheme enables FusionRAID to easily recycle free stripes reclaimed from replicated-to-RAID conversion.

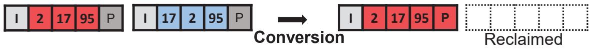


Figure 7: In-position replicated-to-RAID conversion

Figure 7 illustrates the conversion process, showing two stripes (whose blocks form 4 pairs, one of which has since been invalidated by subsequent updates while the rest carry the RAID volume logical blocks 2, 17, and 95). Here the left stripe is converted to RAID storage simply by calculating and filling the parity block (with the invalid block included), without data movement. This easily extends to more general cases: *e.g.*, for RAID-6 we reclaim two out of the three stripes forming 3-block tuples, with two parity blocks calculated for the remaining stripe. To control the replicated area's physical space consumption, the administrator can easily configure the overall size of the logical replicated area.

Finally, background conversion from the replicated to RAID area starts from the least recently accessed stripe pair. The conversion aggressiveness can be governed by policies set by user preferences or workload characteristics. Conversion may be triggered by many different configurable thresholds, such as the number of spare stripes, the ratio of space consumption between replicated and RAID areas, the current workload level, etc., and their combinations.

4.3 Spike Detection and Request Redirection

Even with perfectly spread-out request loads and short write paths, applications suffer sudden surges in request latency when the underlying SSD devices undergo activities such as GC. FusionRAID sets its final line of defense against such adversity by performing constant SSD responsiveness monitoring and request diversion when latency spikes are detected.

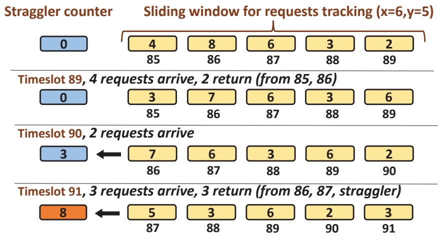

Straggler counter
Sliding window for requests tracking (x=6,y=5)
0
4 8 6 3 2
85 86 87 88 89
Timeslot 89, 4 requests arrive, 2 return (from 85, 86)
0
3 7 6 3 6
85 86 87 88 89
Timeslot 90, 2 requests arrive
3
7 6 3 6 2
86 87 88 89 90
Timeslot 91, 3 requests arrive, 3 return (from 86, 87, straggler)
8
5 3 6 2 3
87 88 89 90 91

Figure 8: FusionRAID SSD spike detection. The number under each yellow box is the timeslot ID and the one within is the count of in-progress IOs issued in that timeslot.

Motivated by our spike behavior study (Section 2.3), we argue that without SSD internal information, though the timing of spikes is hard to predict, their duration and amplitude make reactive avoidance feasible. FusionRAID tracks the number of *stragglers* (with processing time larger than a preset threshold t) among requests issued to each individual SSD, and can quickly react when a performance anomaly happens.

Light-weight spike detection The challenge, however, lies in the efficient tracking of all active requests in a large SSD pool. After all, spikes are more likely to happen under intensive workloads. Space and time overhead of keeping track of thousands of requests at such peak times is not negligible.

To this end, we propose a lightweight spike detection scheme that records the request dispatching time in coarse granularity. It divides the time-line into equal-sized timeslots. For each SSD in the pool, FusionRAID maintains the number of pending requests dispatched within the latest y timeslots (forming a sliding window with a time length of t). A separate *straggler counter* records the number of requests whose pending time exceeds t. Each request issued increments the request counter for the current timeslot by 1, with an issuing timestamp tagged with the request. Similarly, a request's completion decrements the request counter of its corresponding timeslot. Upon the expiration of the current timeslot, the window slides, and the counter of the oldest timeslot has its value aggregated to the straggler counter. An SSD is identified as unresponsive (under spike) if the straggler counter reaches a preset threshold $\hat{x}$. For an unresponsive SSD, with new traffic guided away and spike-causing internal activities receding, eventually, its straggler counter will fall below $\check{x}$, putting it back to full service. For consumer SSDs, we set $\hat{x}$ equal to $\check{x}$ because requests return in batch once spikes end. However, spikes on DC SSDs often overlap with the previous ones so we set a gap between two thresholds, avoiding oscillations.

Figure 8 illustrates this per-SSD spike detection mechanism, with a sliding window sized at 5. At the last step (timeslot 91), the oldest timeslot (86) has its counter merged with the previous straggler counter, minus 2 requests returning (from "straggler" and timeslot 86). The result ($3+7-2=8$) exceeds the threshold $x = 6$, identifying the SSD as unresponsive.

Selective request redirection When an I/O request arrives, FusionRAID judges whether the target SSD is unresponsive by reading its straggler count. When a write is directed onto an unresponsive SSD, FusionRAID searches along the spare block pair or stripe list till it finds one not involving any unresponsive SSD, and performs the update there. The search tends to be short as spikes are relatively rare events. Skipped block pairs or stripes are added to the list tail.

Such redirection also applies to reads. When data requested reside in the RAID area, a slow SSD can be skipped over by data reconstruction from all the remaining blocks in the stripe as in ToleRAID [23]. In the replicated area (which stores more active data and tends to attract more reads), FusionRAID reads from the faster of the blocks (with lower straggler count).

Since our spike detection is reactive, the damage is already done upon successful detection. It might be helpful to adopt existing strategies such as "hedged requests" [17], which resends victim requests to other SSDs when outstanding long enough. In addition, FusionRAID may even proactively trigger GC using the SSD trim mechanism [64] (which informs the SSD to recycle invalid blocks), when it "guesses" that a spike is imminent based on historical monitoring data.

4.4 Metadata Management

Now with major FusionRAID operations explained, we come back to discuss its storage organization, in particular, metadata maintenance necessary to enable partial stripe updates.

Fusion Block Mapping Table (FBMT) This central map-

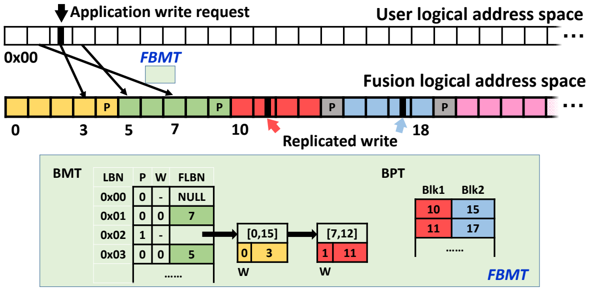


Figure 9: Block mapping and write handling

ping table in FusionRAID contains two components: a *Block Mapping Table (BMT)*, a direct address table storing the block mapping from the user logical block address to the Fusion block address, and a *Block Pairing Table (BPT)*, a hash table storing paired blocks used in replicated writes. Figure 9 gives sample illustrations. To reduce metadata storage, FusionRAID adopts a relatively large block size (64KB by default), which is also its RAID stripe unit size.

FusionRAID may write directly to the RAID area or make replicated writes. This distinction is recorded by the W bit in the 40-bit BMT entries: 0 for RAID and 1 for replicated. In the former case, the remaining 38 bits store the Fusion logical block number (FLBN), supporting a 16PB logical space with the default 64KB block size (*i.e.*, 4096 4TB-SSDs within one enclosure). In the latter, the BMT entry instead stores the smaller block number in a block pair, as a key to the BPT.

Recall the in-position replicated-to-RAID conversion shown in Figure 7. For the stripe to be kept (the left one), its three valid blocks will see their BMT entries updated during the conversion, with the W bit switched to 0 (from "replicated" to "RAID"), and its FLBN properly recorded. For the one to be discarded (the right one), the entire stripe is reclaimed and returned to the spare stripe list. Finally, the four involved block pairs are removed from the BPT.

Adopting larger blocks reduces metadata storage overhead, but creates more partial block updates. To avoid read-modify-write operations, which not only prolong the write process but incur write amplification, FusionRAID utilizes a patching method to append partial updates to the BMT entry of the updated block. The P bit here records the block's partial update history: if set to 1, the FLBN field becomes the head of a linked-list, whose nodes each contains an updated page range, along with the FLBN storing these updates (at their corresponding pages). Again, each element carries a W bit to tell whether the corresponding user logical block is in the replicated area.

Figure 9 demonstrates the handling of a write on user logical block address 0x02, which before the write was in the RAID area. This small write updates pages 7-12 of the block and incurs out-of-place updates via replicated writes. As described in Section 4.2, a pair of blocks (11 and 17, as indicated by the red and blue arrows) are allocated, with these pages written to the corresponding locations within these two blocks, without reading the unchanged data. Now, block 0x02 has its valid data distributed in three Fusion logical blocks: the previous location at block 3, plus the block pair 11 and 17. Thus its BMT entry carries a linked-list, with a node for page range 7-12 linking to the ordered block pair (11,17) in the BPT. Such a linked-list can be periodically compacted so that its length does not exceed the number of pages in a block.

Space overhead Out-of-place updates produce holes (invalid blocks). Holes in stripes cannot be reused directly since the invalid data segment is still involved in parity computation. Again the problem can be solved with periodic compaction similar to SSD GC mechanisms [34, 69], performed during our replicated-to-RAID conversion. In our experiments, we found such small partial rewrites quite infrequent across our evaluated traces (only 1.3%) leading to a small portion of BMT entries having such linked-lists.

FusionRAID maintains FBMT in battery-backed DRAM, for both performance and durability. FusionRAID's overall metadata storage overhead is quite modest. Each BMT entry is 5-bytes long, and our optimized BPT entry only takes 6 bytes (exploiting the proximity of logical addresses for paired blocks). Given the ratio of linked-list entries (each 10-bytes long) and a 10% space limit to the replicated area, these data structures altogether take 0.0084% of storage capacity. This means to manage a fully allocated 60TB SSD pool, only 5.2GB battery-backed memory is needed for FusionRAID to store its mapping data structures. In addition, FusionRAID needs to log updates in battery-backed memory, to ensure consistency of in-progress operations.

5 Performance Evaluation

5.1 Experiment Setup

Testbed We use a SuperMicro 4U storage server, with two 12-core Intel XEON E5-2650 V4 processors and 128GB DDR4 memory, running Ubuntu 16.04 with Linux kernel v4.15.0. It has two AOC-S3008L-L8I SAS JBOD adapters, each connected to a 30-bay SAS3 expander backplane via 2 channels.

For RAID evaluation, we select datacenter devices tested in our SSD performance study (§2): SSD E (Intel D3-S4510). We have 15 drives sitting on one backplane, with each test using one 30-drive SSD pool. The I/O channels provide a combined I/O bandwidth of 24GB/s, exceeding the aggregate sequential bandwidth from the 30 SSDs (195MB/s per SSD we measured, 5.71GB/s in total).

Workloads We use both trace-driven and real application tests. For the former, we implemented a trace player in C using *libaio* [7] that issues direct block I/O requests according to given timestamps. We use eight traces mentioned in Section 2.1 with major attributes in Table 4. For the latter, we evaluate FusionRAID with the popular RocksDB KV store [10] running YCSB workloads [16].

Trace	IOPS	Write ratio	Avg. write size
`YCSB-Load`	409	99%	507.0KB
`YCSB-A`	1353	64%	500.4KB
`YCSB-B`	1218	62%	500.2KB
`TPC-C`	5764	75%	29.6KB
`TensorFlow`	65	69%	80.1KB
`VirtualDesktop`	811	42%	23.8KB
`Exchange`	846	70%	13.1KB
`Proxy`	307	32%	13.8KB

Table 4: Characteristics of experimented I/O traces

RAID systems setup We implement FusionRAID as a Linux kernel module in v4.15.0 with about 5,400 LoC and evaluate it using 29-SSD pools,[1] with the stripe width of 7 (6+1 RAID-5). We compare with *LogRAID* [37], a log-structured RAID-50 that appends all updates. We used our own implementation[2] based on existing literature [13, 37], In addition, we implement *ToleRAID* [23], designed for cutting read tail latency, following its authors' guidance. We also evaluate two common organizations utilizing disk pools: 4 independent (6+1)-disk RAID-5 arrays (*4-RAID5*) and a RAID-50 system that stripes across them (*RAID50*). Finally, we implement another alternative design that adds an NVRAM write buffer above RAID50, which we label *NV-RAID*. All the above five systems use 28 SSDs in 4 RAID groups and 1 as a hot spare.

Unless otherwise noted, we test with SSDs consistently *aged*: first cleaned with the Linux `hdparm` `SECURE_ERASE` command, followed by a full-device sequential write, and finally, 6 hours random 16KB writes using `fio` [4], to guarantee each write generates invalid page(s).

5.2 Overall Performance

Trace-driven, concurrent workloads Considering the common usage of our testbed SSD pool size, we measure overall performance by co-running multiple workloads. More specifically, we select 20 *4-workload mixes* randomly from the aforementioned 8 traces, testing 4-RAID5, RAID50, LogRAID, ToleRAID, and FusionRAID on DC SSDs.

Figure 10 gives the median and tail latencies of 8 test workloads. The bars show the average value among all their executions in the 20 mixes (number of executions ranges between 9 and 13), and error bars mark the min/max values. Note that the y axis in the tail latency chart uses a log scale.

4-RAID5 shows comparatively consistent performance under light workloads (`TF` and `Proxy`) due to hardware isolation. However, the median and tail latencies on 4-RAID5 increase obviously under workloads with larger average write size, higher bandwidth or I/O bursts due to limited resources in one RAID-5. Despite spreading work to all 28 SSDs, RAID50 does not reduce median latency and often worsens tail latency. Light workloads show significant performance degradation on RAID50 when they share resources with heavy ones. RAID50's two-level striping adds further complexity and inter-SSD dependency into the I/O path, making average cases more costly. The worst cases, meanwhile, are slowed down by one or two unresponsive SSDs.

LogRAID does not appear to help: by consolidating all writes to the pool into a single log stream, it reduces concurrency and can utilize 1-2 7-SSD arrays at a time (while RAID50 and FusionRAID simultaneously use more disks). Moreover, data writes still experience the underlying RAID write path, thereby enduring higher latency. Compared with 4-RAID5, ToleRAID brings almost identical median latency under all the 8 workloads, and obviously reduces tail latency under four workloads (*i.e.* `YCSB-A`, `YCSB-B`, `VD`, `Proxy`). However, ToleRAID does not reduce tail latency under the other 4 workloads with higher write ratios, as write I/Os cannot benefit from ToleRAID's request redirection.

FusionRAID, on the other hand, significantly reduces *both* median and tail latencies. Compared with 4-RAID5, FusionRAID shows an average reduction of 49% in median latency across the 8 traces and a maximum of 87%. Compared with RAID50, LogRAID and ToleRAID, the average/maximum reductions are 81%/97%, 76%/98% and 45%/85%, respectively. FusionRAID's P99 improvement (Figure 10(b)) is even more impressive, averaging a 15× reduction (up to 32×) from 4-RAID5, 35× (up to 62×) from RAID50, 34× (up to 61×) from LogRAID, and 8.3× (up to 14×) from ToleRAID. Later we give an in-depth breakdown of sources contributing to such dramatic cut in tail latency.

In addition, for both median and tail, FusionRAID achieves shorter error bars than RAID50, LogRAID, and even 4-RAID5 (which offers hardware isolation, with a dedicated RAID-5 array for each workload). This demonstrates that spreading bursts, simplifying writes, and avoiding spikes bring more reliable performance than simply trying to protect workloads from each other. Finally, all tested RAID systems show the same throughput since trace-driven workloads issue I/O requests according to timestamps. This also allows us to observe the median and tail latency of FusionRAID and other systems under the same load intensity, for fair comparison.

Systems	Workloads	Update ratio	Read avg. latency (ms)		Update avg. latency (ms)	
			Slowest 10%	*Slowest 1%*	*Slowest 10%*	*Slowest 1%*
RAID50	`YCSB-Load`	100%			32.55	232.17
FusionRAID					4.22	29.25
RAID50	`YCSB-A`	50%	0.92	2.4	9.2	63.13
FusionRAID			0.753	1.924	2.35	11.80
RAID50	`YCSB-B`	5%	0.66	1.81	5.17	34.34
FusionRAID			0.47	1.55	1.27	5.86

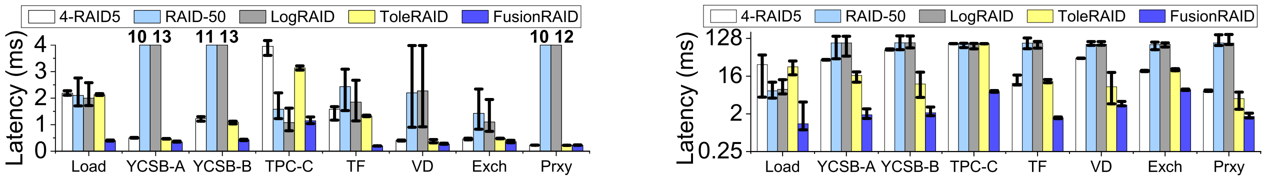
Table 5: RocksDB average and tail latency in running YCSB

Real-application workloads Next, we evaluate with representative YCSB workloads, of varied write intensity levels, running RocksDB on top of ext4 above FusionRAID and RAID50. Table 5 lists workload information and results.

For each workload, Table 5 lists the average latency of the slowest 10% and the slowest 1% of operations, for reads

[1] MOLS requires pool size to be a power of a prime number.

[2] Due to time/resource limit this system is implemented in user space, disabling it from supporting a file system and running applications.



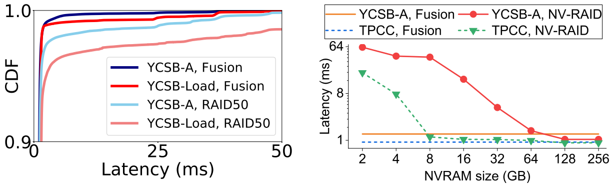
(a) Median latency. Bold numbers above bars denote average median latencies exceeding 4ms.

(b) P99 tail latency, *log scale*.

Figure 10: Overall performance comparison, from 20 randomly selected 4-workload mixes. Error bars show ranges of measured latency from all executions of each workload.

and writes separately. For the least write-intensive workload (YCSB-B), most reads are served from the RocksDB memtable, closing the performance gap between FusionRAID and RAID50. Still, across all three workloads, read tail latency consistently benefits from FusionRAID, due to its *more efficient digestion of request bursts*. For writes, even at an update rate as low as 5%, FusionRAID reduces the average latencies of the slowest 10% and 1% by 4.1× and 5.9×, respectively. As expected, its margin of improvement grows with write intensity, reaching 7.9× for YCSB-Load's slowest 1%.

As for writes, Figure 11(a) plots RocksDB update latency CDFs under the two write-intensive YCSB workloads. FusionRAID reduces the write tail latency significantly, especially with YCSB-Load. A major source of the long tail latency for update operations in RocksDB is the contention between foreground memtable flushing and background compaction [6]. FusionRAID benefits from its higher and more consistent bandwidth, reducing the probability of contention.



(a) RocksDB latency CDF

(b) P99 latency

Figure 11: Detailed latency comparison: (a) FusionRAID vs. RAID50 with YCSB-Load and YCSB-A, and (b) FusionRAID vs. NV-RAID with varying NVRAM size, *log scale*

Comparison with NV-RAID Intuitively, one can tame the tail latency of SSD arrays easily with an NVRAM buffer. To address this concern, we compare FusionRAID with NV-RAID. Since we cannot use the Intel Optane NVDIMM without a processor upgrade, we emulate an NVRAM buffer using DRAM and set a 100ns delay for each cacheline flush operation [32, 43]. To focus on the effect of such a buffer, we only add delays for data operations and not metadata ones.

Figure 11(b) illustrates the P99 latency of NV-RAID with different NVRAM sizes under YCSB-A and TPC-C, and that of FusionRAID (two straight lines) for reference. For YCSB-A, the P99 latency remains over 40ms, 30× higher than the FusionRAID result, with an NVRAM buffer size under 8GB. Here such small buffers do not sufficiently ease write pressure, incurring SSD GC activities that reduce effective SSD array bandwidth and delaying buffer flushes, which in turn results in further NVRAM buffer space shortage. As the buffer size increases, NV-RAID's tail latency improves, nearly catching up with FusionRAID at 64GB and leveling off over 128GB.

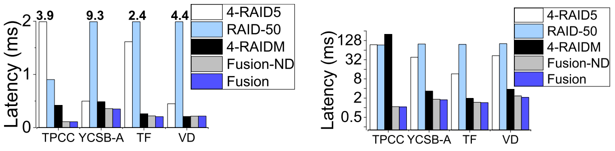
For TPC-C, as the NVRAM buffer grows, the P99 latency of NV-RAID decreases more quickly. The inter-workload difference here is due to TPC-C's smaller requests and higher issuing rates, while YCSB-A contains larger (512KB) I/Os. Therefore NV-RAID's tail latency gets fairly close to FusionRAID's, with an 8GB write buffer. This also leads to better median latency of NV-RAID than FusionRAID with TPC-C (28μs vs. 96μs) with the former using a 256GB buffer. With YCSB-A, on the other hand, even at this full NVRAM buffer capacity NV-RAID delivers a median latency of 391μs (vs. FusionRAID's 320μs).

One sees that FusionRAID delivers similar or better tail latency performance as adopting an NVRAM buffer, while the median performance is more workload-dependent. Meanwhile, FusionRAID is designed to support multiple workloads simultaneously. Under such scenarios, concurrent applications need to share the precious NVRAM space, and may suffer write bandwidth scalability problems when more concurrent threads access the NVRAM, as found by a recent study [70]. Finally, NVRAM is far more expensive than SSDs: one 256GB NVDIMM costs $2,595 [2], equal to the price of our 30-SSD pool, even without considering the cost of the required server upgrade. Also, in the case of the Intel Optane NVDIMM, unlike most storage devices, a larger NVRAM device (256GB vs. 128GB) actually costs more per GB.

5.3 Impact of Individual Techniques

For the rest of the section, due to space limit, we focus on the mix including four different types of workloads (TPC-C, YCSB-A, TF and VD). Here, to isolate the improvement brought by key FusionRAID techniques, we evaluate its two intermediate versions: *4-RAIDM*, with 4 independent 7-disk volumes like 4-RAID5 but with all write requests conducted by performing mirrored log-structure writes in a round-robin way, and *Fusion-ND*, same as FusionRAID but with SSD spike detection and request redirection removed.

Figure 12 shows the median and P99 latencies of different systems. From 4-RAID5 to RAID50, as explained earlier, building a single, shared volume on the entire SSD pool does not necessarily help. Replacing the costly RAID write with replicated write (4-RAIDM), in contrast, brings about the most significant improvement. Compared with 4-RAID5, 4-RAIDM reduces median latency by 2.1%-89% and shrinks P99 by 5.9-12×. However, the P99 latency of 4-RAIDM under `TPC-C` reaches 202ms, caused by the limited number of SSDs to serve intensive writes.



(a) Median latency (bold numbers giving values over 2)

(b) P99 latency, *log scale*

Figure 12: Incremental impact of proposed techniques

Fusion-ND enjoys the same benefits of simplified writes but spreading work to the entire pool rather than having 4 physically isolated arrays. Now with a shorter and decoupled write path, involving more disks improves processing power without propagating spikes. This produces a significant reduction from the most write-intensive workload (`TPC-C`) in tail latency (again plotted in log scale).

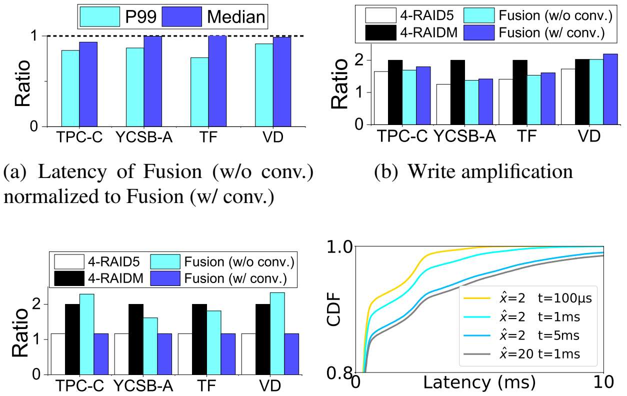
	Slowest 1% avg. latency (ms)		Slowest 0.1% avg. latency (ms)	
	Fusion-ND	*Fusion*	*Fusion-ND*	*Fusion*
TPC-C	1.90	1.47	6.34	2.79
YCSB-A	3.04	2.41	15.37	9.71
TF	2.31	2.20	52.16	49.83
VD	6.96	2.97	45.49	11.50

Table 6: Average latency at tails: Fusion-ND vs. Fusion

Finally, from Fusion-ND to FusionRAID, we add SSD spike detection and request redirection. Their difference may not seem significant from Figure 12, as it only comes into play when requests run into device-side spikes. As shown in Table 6, FusionRAID reduces the average latency of the slowest 1% requests by 1.1-2.3×, redirecting 0.73% of write requests from the Fusion-ND baseline. We also measure the frequency and duration of spikes on Fusion-ND with spike detection on and request redirection off. On average one SSD experiences 4.85 spikes/minute, each lasting for 3.79ms. Although these spikes appear quite short, bursty requests suffer performance degradation in batch once encountering them.

5.4 FusionRAID Overhead and Sensitivity

FusionRAID's major internal I/O activity is its replicated-to-RAID data conversion. Our tests presented earlier all have conversion turned on (only temporarily suspended when foreground throughput on a disk exceeds 40MB/s, our observed per-drive average). To examine its performance impact, Figure 13(a) shows performance with and without such background conversion. One sees that even with our current brute-force conversion policy (with no foreground-aware optimization such as avoiding application busy bursts), its performance overhead is quite small, thanks to our in-position stripe allocation during replicated writes.



(a) Latency of Fusion (w/o conv.) normalized to Fusion (w/ conv.)

(b) Write amplification

(c) Space overhead

(d) Latency CDF of Fusion with varying thresholds in spike detection

Figure 13: FusionRAID overhead and sensitivity

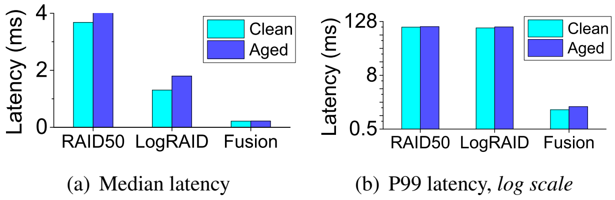
Next, Figure 13(b) reports the ratio of write amplification across all systems. Compared with 4-RAIDM, FusionRAID reduces the ratio across most of the workloads since it performs RAID writes directly for large requests. An exception is that FusionRAID brings a higher ratio under `VD`. This is because numerous replicated writes generated by dominant small writes in `VD` (see Table 4), plus parity updates in conversion, increase write amplification more than mirroring does. On average, its in-position conversion only brings around a 5.6% increase in write amplification.

Space consumption, meanwhile, depends on the aggressiveness of the background conversion policy adopted. Figure 13(c) gives the overall space consumption (vs. user data size), for all data written during the trace run. As expected, 4-RAID5's extra space overhead comes from the single parity block in its 6+1 stripe. 4-RAIDM has a constant space ratio of 2 as it performs simple mirroring. FusionRAID, with conversion turned off, has a slightly varying space ratio across workloads, due to their different write patterns. With conversion fully performed, one of the replicas is recycled (the other reclaimed) and multiple writes are compacted, returning the space consumption to the same as the 4-RAID5 level.

In addition, we examine the sensitivity of parameters in spike detection (§4.3). We first set y (determining the counting precision) at 10 and $\check{x}$ (used to speculate the end of latency spikes) at 0 empirically. Figure 13(d) shows latency CDFs under `Exchange` with different values of $\hat{x}$ and t. Smaller $\hat{x}$ and t help FusionRAID to detect spikes and react earlier while more

SSDs are identified as unresponsive, leaving fewer choices for incoming I/Os. Fortunately, FusionRAID performs well with smaller thresholds, thanks to the large 30-SSD pool.

Finally, we discuss how SSD aging affects the systems' performance. Figure 14 compares the performance of RAID50, LogRAID, and FusionRAID on clean and aged SSDs under the same workload combination as in §5.3, showing median and P99 latency on average. Aside from its capability to significantly reduce the median/tail latency in all cases, FusionRAID shows different behavior across clean and aged SSDs than the baseline systems. While RAID50 and LogRAID have higher median latency on aged SSDs due to increased GC activities, FusionRAID works as well there due to its spike aversion. For tail latency, however, FusionRAID does perform better on clean SSDs, where it has more alternatives to redirect requests. RAID50 and LogRAID, without such a mechanism, show no differences across aged and clean SSDs.



(a) Median latency (b) P99 latency, *log scale*

Figure 14: Performance on clean vs. aged SSDs

6 Related Work

SSD RAID designs SSD RAID systems have been extensively studied, with approaches roughly in three groups: 1) improving reliability by distributing parity unevenly across the array [5] or through wear leveling across member SSDs [61], 2) enhancing average-case performance and/or reliability by mitigating the parity update problem [14,15,28,36,42], and 3) taming tail-latency by alleviating GC impact [23,37,38,66,69]. Approaches in the first two groups do not address tail latency, and most in the third group [38, 66, 69] rely on host-managed/open-channel SSDs. One exception, ToleRAID [23], focuses on cutting read tails under a full-stripe read workload. In contrast, FusionRAID works on off-the-shelf SSDs and aims to reduce *both average-case and tail latencies*, while maintaining the same fault tolerance level.

A highly related system is SWAN [37], which eliminates GC impact on commodity SSD arrays (Rails [54] is similar but focuses on read protection from GC). It partitions drives into groups, which rotate in handling foreground and internal writes, shielding user writes from GC traffic. It targets settings where SSD RAIDs are built for capacity, but with aggregate bandwidth bound by network, while FusionRAID targets shared storage serving latency-critical applications. Compared with SWAN, FusionRAID is reactive rather than proactive, but protects and redirects both reads and writes.

RAID data layout optimization Parity declustering [46] utilizes as many disks as possible to serve application requests in data reconstruction. It has been extended and optimized by many [3,25,26,48,63]. It is also widely adopted in industry, by products such as PanFS [63], the IBM GPFS Native RAID [19] and Spectrum Scale RAID [27], HPE 3PAR [60], and Huawei's RAID2.0+ [44]. FusionRAID's design leverages existing Latin-square based deterministic addressing of RAID+ [71] but augments it with explicit block mapping to enable two-phase writes and out-of-place updates.

RAID write optimization Purity [15], Flash-Aware RAID [28], and PPC [14] use NVRAM to buffer the incoming data and/or parity information and delay parity updates, so as to conduct full-stripe writes and reduce the reads involved in parity updates [15], or to reduce the number of parity commits to SSDs [14, 28]. However, they require large amounts of NVRAM for storing data and/or parity information. ESAP-RAID [36] and RAID-Z [9] organize the incoming data in elastic-width stripes to reduce parity-induced read overhead, at the cost of increased stripe management complexity.

Two-phase writing was used by existing systems: AutoRAID [65], DiskReduce [21], and Log Disk Mirroring (LDM) [67] all write data to a replicated zone, with future background conversion to the RAID zone. However, FusionRAID is unique in its in-place conversion by replicating in a stripe-ready manner. Note that it specifically targets SSD RAIDs, where massive data migration may cause frequent GC and consume more SSD write cycles.

Alleviating GC Interference Harmonia [39] and Global Garbage Collection (GGC) [38] synchronize GC across a set of SSDs, thereby reducing overall performance variability. Application-managed Flash [41] and LightNVM [8] eliminate GC overhead by letting the host software manage the exposed flash channels. Several other systems cut read tail latency by issuing an extra read to parity block and rebuild the "late" data [23, 66, 69], and write one by enforcing at most one active GC in every RAID group and writing data to a no-GC member [66, 69]. Unlike the above, FusionRAID works without assuming SSD internal information/control, by observing SSDs' performance behavior to detect the onset of degradation and steer away if possible.

7 Conclusion

With FusionRAID, we argue that SSD RAID systems can be much faster *and* more consistent, by eagerly spreading application load, lazily performing parity writes (instead trading space temporarily for simple replicated writes), and carefully watching individual SSD's performance and waiting out their transient latency spikes. Large SSD enclosures, once not constrained with the rigid routines of traditional RAID arrays, simultaneously provide high concurrency to serve co-executing applications' bursty I/Os, and high flexibility in avoiding drives under transient performance degradation.

Acknowledgment

We thank all reviewers for their insightful comments and helpful suggestions. We are especially grateful to our shepherd, Keith Smith, for his thorough, detailed, and patient guidance during our camera-ready preparation. This work was supported by the National key R&D Program of China under Grant 2018YFB0203902, and the National Natural Science Foundation of China under Grants 61672315 and 62025203.

References

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, and Rina Panigrahy. Design tradeoffs for SSD performance. In *the 2008 USENIX Annual Technical Conference (USENIX'08)*, pages 57–70, 2008.

[2] Paul Alcorn. Intel Optane DIMM Pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html, 2019.

[3] Guillermo A. Alvarez, Walter A. Burkhard, Larry J. Stockmeyer, and Flaviu Cristian. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings of 25th International Symposium on Computer Architecture (ISCA'98)*, pages 109–120, 1998.

[4] Jens Axboe. FIO: Flexible I/O Tester. https://github.com/axboe/fio, 2019.

[5] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential RAID: Rethinking RAID for SSD reliability. *ACM Transactions on Storage (TOS)*, 6(2):4, 2010.

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 753–766, 2019.

[7] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Linux Symposium*, pages 371–386, 2003.

[8] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 359–374, Santa Clara, CA, February 2017. USENIX Association.

[9] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. https://www.snia.org/sites/default/orig/sdc_archives/2008_presentations/monday/JeffBonwick-BillMoore_ZFS.pdf, 2008.

[10] Dhruba Borthakur. RocksDB: A persistent key-value store. https://rocksdb.org/, 2014.

[11] Eric Burgener. Justifying investment in all-flash arrays. https://www.emc.com/collateral/analyst-reports/justifying-investments-in-all-flash-arrays.pdf, 2019.

[12] Feng Chen, David A Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 181–192. ACM, 2009.

[13] Tzi-cker Chiueh, Weafon Tsao, Hou-Chiang Sun, Ting-Fang Chien, An-Nan Chang, and Cheng-Ding Chen. Software orchestrated flash array. In *Proceedings of International Conference on Systems and Storage (SYSTOR'14)*, pages 1–11. ACM, 2014.

[14] Ching-Che Chung and Hao-Hsiang Hsu. Partial parity cache and data cache management method to improve the performance of an SSD-based RAID. *IEEE Transactions on Very Large Scale Integration (VLSI'14) Systems*, 22(7):1470–1480, 2014.

[15] John Colgrove, John D Davis, John Hayes, Ethan L Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*, pages 1683–1694. ACM, 2015.

[16] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10)*, pages 143–154. ACM, 2010.

[17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[18] 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 *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[19] Veera Deenadhayalan. GPFS Native RAID for 100,000-Disk Petascale Systems. In *25th Large Installation System Administration Conference (LISA'11)*, 2011.

[20] DELL EMC. VMAX All Flash Family, https://www.dellemc.com/en-us/collaterals/unauth/data-sheets/products/storage-2/h16051-vmax-all-flash-250f-950f-ss.pdf. 2020.

[21] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. DiskReduce: Replication as a prelude to erasure coding in data-intensive scalable computing. *SC'11*, 2011.

[22] FUJITSU. FUJITSU Storage ETERNUS AF650 S3. https://www.fujitsu.com/global/products/computing/storage/all-flash-arrays/eternus-af650-s3/index.html, 2020.

[23] 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, Santa Clara, CA, February 2016. USENIX Association.

[24] Md E Haque, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, Kathryn S McKinley, et al. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ACM SIGPLAN Notices*, volume 50, pages 161–175. ACM, 2015.

[25] Mark Holland and Garth A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*, pages 23–35, 1992.

[26] Mark Holland, Garth A. Gibson, and Daniel P. Siewiorek. Fast, on-line failure recovery in redundant disk arrays. In *Proceedings of The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS'93)*, pages 422–431, 1993.

[27] IBM. IBM Spectrum Scale RAID. https://www.ibm.com/support/knowledgecenter/en/SSYSP8_5.3.1/raid_adm.pdf, 2017.

[28] Soojun Im and Dongkun Shin. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Transactions on Computers*, 60:80–92, 01 2011.

[29] Intel. Intel Solid State Drives. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives.html. 2020.

[30] I/O Umass Trace Repository. OLTP Application I/O and Search Engine I/O. http://traces.cs.umass.edu/index.php/Storage/Storage.

[31] Itnews. ANZ Bank goes all-flash for storage. https://www.itnews.com.au/news/anz-bank-goes-all-flash-for-storage-490262. 2020.

[32] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[33] Dawoon Jung, Jeong Uk Kang, Heeseung Jo, Jin Soo Kim, and Joonwon Lee. Superblock FTL: A superblock-based flash translation layer with a hybrid address translation scheme. *ACM Transactions on Embedded Computing Systems*, 9(4):1–41, 2010.

[34] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*, 2014.

[35] Jaeho Kim, Donghee Lee, and Sam H Noh. Towards SLO complying SSDs through OPS isolation. In *13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 183–189, 2015.

[36] Jaeho Kim, Jongmin Lee, Jongmoo Choi, Donghee Lee, and Sam H Noh. Improving SSD reliability with RAID via elastic striping and anywhere parity. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, pages 1–12. IEEE, 2013.

[37] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. Alleviating garbage collection interference through spatial separation in all flash arrays. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 799–812, Renton, WA, July 2019. USENIX Association.

[38] Youngjae Kim, Junghee Lee, Sarp Oral, David A Dillow, Feiyi Wang, and Galen M Shipman. Coordinating garbage collection for arrays of solid-state drives. *IEEE Transactions on Computers*, 63(4):888–901, 2014.

[39] 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 *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST'11)*, pages 1–12. IEEE, 2011.

[40] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of*

the 10th ACM International Systems and Storage Conference, pages 1–11, 2017.

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

[42] Yongkun Li, Helen HW Chan, Patrick PC Lee, and Yinlong Xu. Elastic parity logging for SSD RAID arrays. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16)*, pages 49–60. IEEE, 2016.

[43] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices*, 52(4):329–343, 2017.

[44] Huawei Technologies Co., Ltd. RAID 2.0+ Technical White Paper. `https://actfornet.com/HUAWEI_STORAGE_DOCS/Storage_All2/Enterprise%20Unified%20Storage%20RAID%202.0+%20Technology-HUAWEI%20OceanStor%20Technical%20White%20Paper.pdf`, 2014.

[45] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[46] Richard R. Muntz and John C. S. Lui. Performance analysis of disk arrays under failure. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB'90)*, pages 162–173, 1990.

[47] Sampann N. Amazon found every 100ms of latency cost them 1% in sales. `https://www.linkedin.com/pulse/amazon-found-every-100ms-latency-cost-them-1-sales-sampann/`, 2016.

[48] David Nagle, Denis Serenyi, and Abbie Matthews. The Panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC'04)*, pages 1–10. IEEE Computer Society, 2004.

[49] NetApp. AFF A-Series All Flash Arrays. `https://www.netapp.com/us/products/storage-systems/all-flash-array/aff-a-series.aspx#technical-specifications`. 2020.

[50] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM International Conference on Management of Data (SIGMOD'88)*, pages 109–116, 1988.

[51] PureStorage. FlashArray//X. `https://www.purestorage.com/products/nvme/flasharray-x.html`. 2020.

[52] Sandisk. SanDisk InfiniFlash System. `https://www.solidstateworks.com/InfiniFlash.asp`. 2020.

[53] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 67–80, Santa Clara, CA, 2016. USENIX Association.

[54] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on Rails: Consistent flash performance through redundancy. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*, pages 463–474, Philadelphia, PA, June 2014. USENIX Association.

[55] SNIA. Microsoft Enterprise Traces. `http://iotta.snia.org/traces/130`, 2007.

[56] SNIA. Microsoft Production Server Traces. `http://iotta.snia.org/traces/158`, 2007.

[57] SNIA. MSR Cambridge Traces. `http://iotta.snia.org/traces/388`, 2007.

[58] SNIA. SNIA Block I/O Traces. `http://iotta.snia.org/tracetypes/3`, 2017.

[59] P Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, pages 513–527. USENIX Association, 2015.

[60] Karl L. Swartz. 3PAR Fast RAID: High performance without compromise. `http://www.kls2.com/~karl/papers/raid-wp-10.0.pdf`, 2010.

[61] Wei Wang, Tao Xie, and Abhinav Sharma. SWANS: An interdisk wear-leveling strategy for RAID-0 structured SSD arrays. *ACM Transactions on Storage (TOS)*, 12(3):10, 2016.

[62] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*, pages 1–12. IEEE, 2006.

[63] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the Panasas parallel file system. In *6th Usenix Conference on File and Storage Technologies (FAST'08)*, pages 17–33, 2008.

[64] Wikipedia. Trim (computing). https://en.wikipedia.org/wiki/Trim_(computing). 2020.

[65] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, 1996.

[66] Suzhen Wu, Haijun Li, Bo Mao, Xiaoxi Chen, and Kuan-Ching Li. Overcome the GC-induced performance variability in SSD-based RAIDs with request redirection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[67] Suzhen Wu, Bo Mao, Xiaolan Chen, and Hong Jiang. LDM: Log disk mirroring with improved performance and reliability for SSD-based disk arrays. *ACM Transactions on Storage (TOS)*, 12(4):22, 2016.

[68] Zhe Wu, Curtis Yu, and Harsha V Madhyastha. CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, pages 543–557, 2015.

[69] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. *ACM Transactions on Storage (TOS)*, 13(3):22, 2017.

[70] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 169–182, 2020.

[71] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. RAID+: Deterministic and balanced data distribution for large disk enclosures. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 279–294, Oakland, CA, 2018. USENIX Association.

Behemoth: A Flash-centric Training Accelerator for Extreme-scale DNNs

Shine Kim[†‡*] Yunho Jin[†*] Gina Sohn[†] Jonghyun Bae[†] Tae Jun Ham[†] Jae W. Lee[†]
[†]*Seoul National University* [‡]*Samsung Electronics*

Abstract

The explosive expansion of Deep Neural Networks (DNN) model size expedites the need for larger memory capacity. This movement is particularly true for models in natural language processing (NLP), a dominant application of AI along with computer vision. For example, a recent extreme-scale language model GPT-3 from OpenAI has over 175 billion parameters. Furthermore, such a model mostly consists of FC layers with huge dimensions, and thus has a relatively high arithmetic intensity. In that sense, an extreme-scale language model does not suit well to the conventional HBM DRAM-based memory system that lacks capacity and offers extremely high bandwidth. For this reason, we propose to pair the neural network training accelerator with the flash-based memory system instead of the HBM DRAM-based memory system. To design the effective flash-based memory system, we optimize the existing SSD design to improve the SSD bandwidth as well as endurance. Finally, we evaluate our proposed platform, and show that Behemoth achieves 3.65× cost saving over TPU v3 and 2.05× training throughput improvement over the accelerator attached to a commercial SSD.

1 Introduction

Deep Neural Networks (DNNs) have become pervasive in various application domains. Early DNN models demanded only high computation, but recent models additionally require increasing memory capacity with continued scaling of DNNs. This is especially true for Natural Language Processing (NLP) models [5, 18, 39, 40, 44, 54], targeting problems including language translation [2, 50, 65], text generation [5, 53, 59] and summarization [35, 41, 69], and sentiment analysis [18, 40].

This advent of extreme-scale NLP models (with more than several billion parameters) is one of the most important recent breakthroughs in DNNs. Transformer [66] introduced in 2017 demonstrated that neural networks could substantially outperform existing NLP techniques, and the introduction of BERT [18] showed that the concept of training the Transformer language model with a large corpus could produce a versatile language model that can be utilized for various natural language processing tasks. Following BERT, many Transformer-based NLP models [4, 5, 16–18, 31, 34, 39, 40, 44, 54, 59, 66, 68, 72] have emerged. Specifically, GPT-3 [5], one of the most recent language models, has established state-of-the-art performance in various NLP tasks.

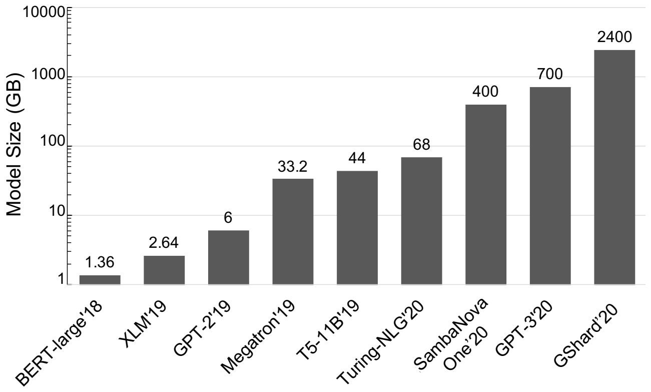


Figure 1: Trends of model size scaling with large NLP models

These NLP models explosively expand their sizes, taking hundred billions of parameters. Figure 1 shows that the size of the model has increased by more than 1000× over the last two years. For example, GShard [39] from Google contains roughly 2.4TB of parameters. This demand for memory capacity has been partly satisfied by simply supplying more memory. However, relatively stagnant DRAM scaling cannot keep pace with this increase in the DNN model size. Therefore, the solution of merely augmenting the system with extra DRAM is impractical.

It is impossible to process these models in a data parallel manner on the conventional hardware because it would require each device to hold the entire model [60]. There are mainly two solutions to this capacity problem. The first approach is to simply discard some computation results in the forward path and recalculate them during the backward path [8]. Unfortunately, this approach can incur a substantial

[*] These authors contributed equally to this work.

amount of extra computations. The other approach is the utilization of the model parallelism. This technique divides the model into multiple partitions and distributes them across multiple devices so that the system as a whole can accommodate the model. Following the GShard example from the previous paragraph, at least 75 devices with 32GB of memory [27, 64] are needed to run this model. Unfortunately, model parallelism comes with its inherent drawbacks. The dimensions and types of layers in a model are not identical. Thus careful load balancing of partitioned models is required. Additionally, stalls due to dependency may arise, and extra inter-node communication may be required to exploit pipeline parallelism.

To tackle this capacity problem differently, we first analyze the characteristics of those emerging NLP models. Our analysis reveals that, unlike conventional models, these extreme-scale NLP models consume huge memory proportional to the parameter size, but do not fully utilize the bandwidth of high-performance DRAM (e.g., high bandwidth memory (HBM)) because of a much higher degree of data reuse. This high arithmetic intensity stems from the huge model sizes, as parameters are shared by much more input elements. Thus, we have identified the opportunity to use high-capacity, low-performance NAND flash instead of low-capacity, high-performance HBM to train these extreme-scale NLP models.

Thus, we propose Behemoth, a NAND flash-centric training accelerator targeting extreme-scale NLP models. Behemoth allows those NLP models to be trained in a *data parallel* manner, where the training data set (not the model) is partitioned across multiple devices. To satisfy the computation, memory, and bandwidth requirements simultaneously, Behemoth integrates one Weight Node with multiple Activation Nodes. Like the parameter server in a data-parallel distributed system, Weight Node is responsible for feeding the Activation Nodes with weight data for each layer and reducing the weight gradients produced from them. Activation Nodes are the actual worker nodes processing each layer of the model. These nodes are composed of a DNN-specific Compute Core and enhanced large NAND flash memory, which can feed the core in time and store the data generated during the training process. The need for enhanced high-bandwidth flash memory is engendered by large data size and high-throughput Compute Cores. The NAND flash memory bandwidth can be scaled by increasing the number of channels and chips. However, in order to deliver the required performance of the DNN training workload, for example, tens of GB/s or more, the bottleneck caused by firmware must be resolved. Behemoth provides a performance scalable flash memory system for DNN training by hardware automation of the write datapath in a controller. Furthermore, Behemoth drastically extends the endurance of NAND by leveraging tradeoffs between retention time and endurance (P/E cycles) of NAND flash.

To summarize, our contributions are listed as follows:

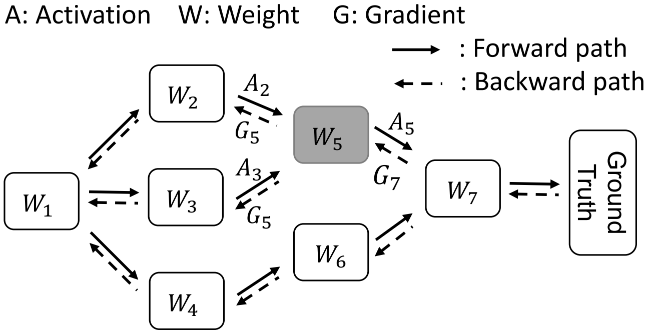


Figure 2: DNN training process and dataflow. The solid line represents the forward path in Layer 5 where A_2 and A_3 are provided as inputs. The dotted line depicts the backward path of the same layer, where A_5 is retained with W_5 and G_7 is received to compute G_5.

- We carefully analyze the memory capacity problem that arises when training extreme-scale DNN models and identify new opportunities to leverage NAND flash devices, replacing expensive DRAM devices.
- For efficient training of these models, we present Behemoth, a novel flash-centric training accelerator targeting those models.
- To satisfy the bandwidth and endurance requirements of DNN training, we propose Flash Memory System (FMS) for Behemoth, which provides both high bandwidth and high endurance.

2 Background and Motivation

2.1 DNN Training

DNN training is a process where a neural network model utilizes a training dataset to improve its performance (e.g., accuracy) by updating its parameters. It is essentially a repetitive process of matrix operations. Both activations and weights, represented as matrices, are multiplied and added in every layer. Figure 2 describes an *iteration*, where activations follow the path predefined by the model, repeating the forward and the backward path. The training process is deterministic as the process follows a predefined forward and ensuing backward path. It is also iterative in that the paths are repeated until the desired result is generated. One set of a forward and backward path is called, surprisingly, *iteration*, and the set of inputs being processed in the iteration is termed *batch*.

In the forward path, input activations are passed to a layer, as shown in Figure 2. Upon receiving the activations A_2 and A_3, they are multiplied with the weight W_5 and generate output activation A_5. The output activation is retained in the layer for use in the backward path and sent as input to the next layer in the forward path. This process is repeated until the last layer. The output of the last layer is compared with the ground truth to calculate the error. This error is fed back to the last layer, thus starting the backward path. In this path, the gradients

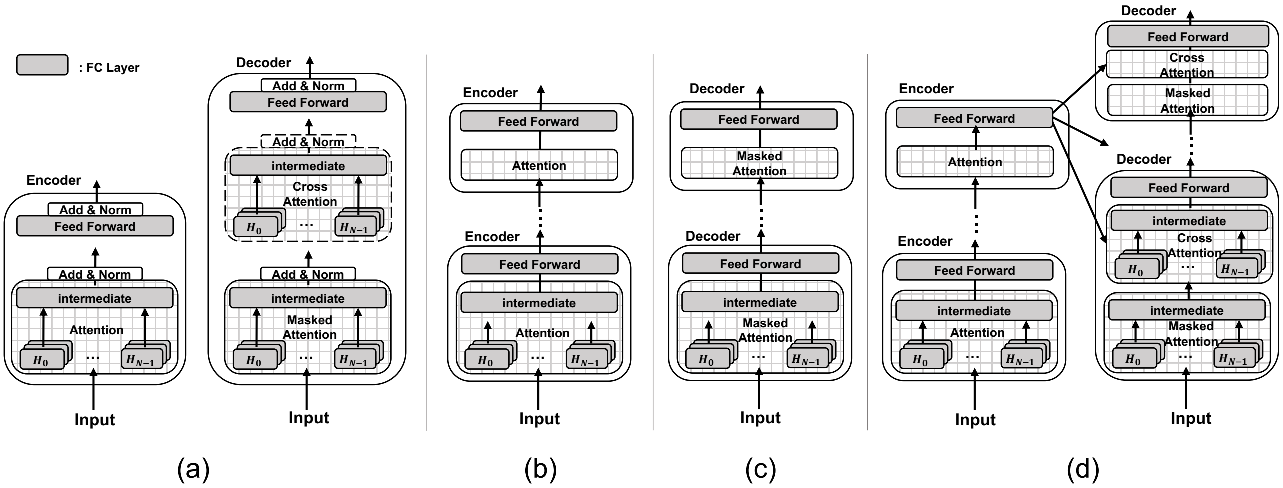


Figure 3: Simplified view of Transformer-based models. (a) Transformer block as a whole which consists of an encoder block(left) and a decoder block(right). Each H_n represents the n^{th} head in the attention layer which performs matrix multiplication to calculate a vector which is concatenated in the n^{th} position in the resulting vector. (b) BERT-like models where only the encoder part of the transformer is used. (c) GPT-like models where only the decoder part of the transformer is used. The main difference from (b) is that these models typically adopt masked attention, where the attention operation can only attend tokens appearing before the current one. (d) T5-like models where both encoders and decoders are used. Unlike those illustrated in (b) and (c) they have cross-attention layers in the decoder modules. The final output of encoders in T5 is distributed across all cross-attention layers in their decoders.

of both activations and weights with respect to the error are created. Input gradients are then propagated backward, generating weight gradients and updating weights along the way. Every DNN model training follows this pattern, regardless of which application it targets. However, the size and structure of the models differ by their usage. There are diverse application domains of DNNs, such as natural language processing (NLP), reinforcement learning (RL), computer vision (CV), and so on. This work primarily focuses on NLP.

2.2 Extreme-scale Language Models

Many of the emerging, extreme-scale NLP models share the same internal structure. Although the details may vary, all of these NLP models essentially consist of Transformer blocks shown in Figure 3(a). Specifically, some NLP models (e.g., BERT [18], RoBERTa [44], BART [40]) are constructed by stacking the encoder blocks of the Transformer model as shown in Figure 3(b), and some other NLP models (e.g., GPT-2 [53], GPT-3 [5]) are constructed by stacking the decoder blocks of the Transformer model as shown in Figure 3(c). Finally, as in the original Transformer, some models(e.g., T5 [54], Transformer-XL [17]) have stacked encoder blocks followed by extended decoder blocks as shown in Figure 3(d). Conceptually, encoder blocks can focus on relevant parts of the input sentence through attention layers. As a result, encoder-only models are mostly utilized for comprehension tasks (e.g., sentiment analysis,question-answering). Decoder blocks contain a masked attention layer, which is identical to the encoder's attention layer except that words coming after the current position are masked. Based on such characteristics, many decoder-only models are utilized for text generation tasks. In models that exploit both encoder and decoder blocks, a cross-attention layer is added to the decoders, which helps the decoder focus on the input sentence's related positions by utilizing the encoder block's output. Such models are often utilized for tasks like translation.

One notable characteristic of these Transformer-based language models is their gigantic sizes. For example, GPT-3 has 175 billion parameters, and the parameter size has been scaled by over 1000× over the past two years as shown in Figure 1. This implies that the model size will scale even further in the future. The gigantic size of these emerging language models brings many unique challenges to the existing neural network processing system. One of the most notable ones is the *memory capacity wall*, explained in the following.

2.3 Challenges for Extreme-scale Language Model Training

As stated above, the GPT-3 model has 175 billion parameters, where each parameter is represented in the FP16 format. In such a case, it takes 350GB of storage to store the parameters for this model. The storage cost is worse in training since training requires extra storage to buffer each layer's output activations as well as weight gradients. Here, the size of the weight gradients is identical to the weights themselves and thus cost 350GB. The sum of output activation sizes for the GPT-3 model is 43.7GB for a single input sequence with 2048 tokens, and this linearly increases with the number of input sequences in a single batch (i.e., batch size). For example, if one wants to train the GPT-3 with a batch size of 32, it will

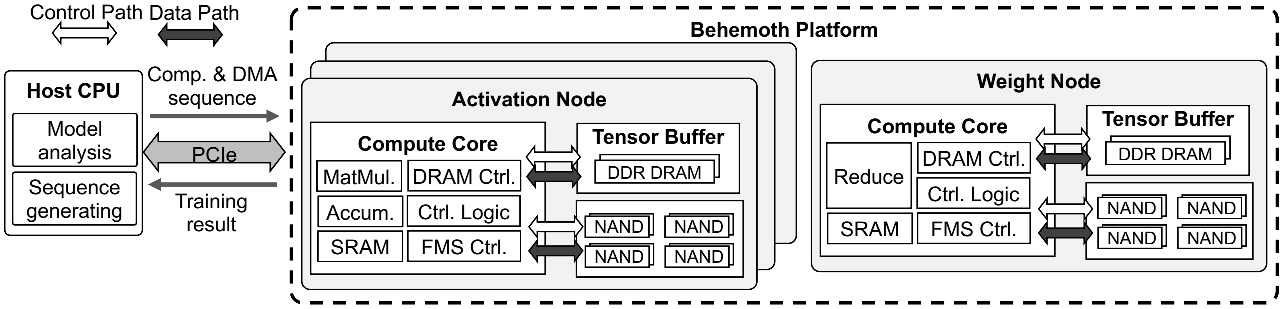


Figure 4: Behemoth architecture

require around 2.1TB storage space.

Unfortunately, conventional DNN training platforms such as NVIDIA GPUs or Google TPUs are equipped with HBM, which offers very limited storage capacity. For example, a single TPU chip, as well as a single V100 GPU only has (a maximum of) 32GB memory space. In order to train the GPT-3 model on these platforms, a minimum of 66 devices are required. In terms of computation, this is not a significant problem since training an extreme-scale model like GPT-3 in a reasonable time frame requires a very large computational capability, which often exceeds that of the 66 GPUs or TPU chips. However, the inefficiency here is that existing platforms such as TPU chips or GPU comes with an unnecessarily expensive memory system that is inadequate for the large-scale language model training.

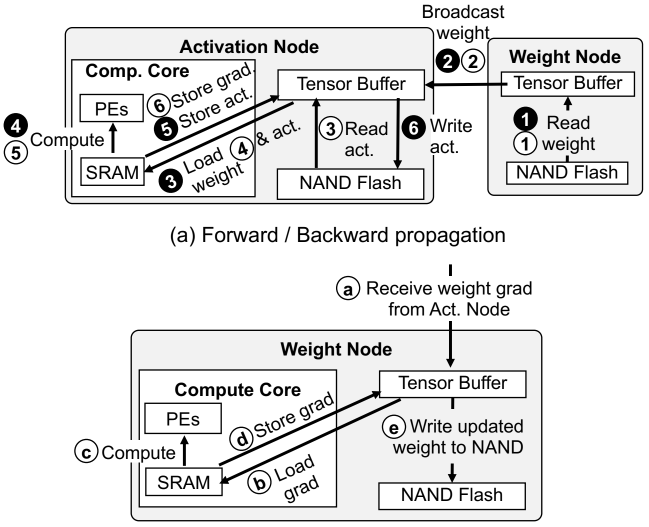
As shown in Figure 3, large-scale, Transformer-based language models are mostly piles of fully-connected (FC) layers. Here, the dimensions of the FC-layers are very large. For example, a feedforward layer in the encoder/decoder block includes matrix multiplication between a 2048×12288 matrix and a 12288×49152 matrix to process a single input. In such cases where the matrix dimensions for the fully-connected layer are large, each value in the matrix is reused many times, and thus the layer ends up requiring a relatively small number of memory accesses compared to the amount of computation (i.e., the arithmetic intensity is low). Specifically, if the input matrix size for the FC layer is $m \times n$ and the weight matrix for the FC layer is $n \times k$, the amount of required multiply-accumulate (MAC) operations is mnk, and the amount of data that needs to be communicated from/to memory is $mn + nk + mk$. Thus, a larger m, n, or k increases the ratio of mnk to $mn + nk + mk$. When $m = 2048 * 32, n = 12288, k = 49152$ as in the feedforward layer of the encoder/decoder block processing 32 inputs (i.e., sequence length = 2048, batch size = 32), the operation requires 73.728 TFLOP for this matrix multiplication, and the total amount of data transfer from/to memory is 9.26GB, assuming FP16 datatype. For a single TPU chip, which can perform 105 TFLOP per second, this matrix multiplication takes 0.7 seconds. Since the TPU v3 is equipped with two HBM memory channels whose aggregate bandwidth is 600GB/s, this is enough time for the chip to transfer 420 GB. However, the operation only requires 9.26GB data transfer with the memory, grossly underutilizing the memory bandwidth. Thus, it is critical to match the arithmetic intensity of the models and the compute-to-memory (disk) bandwidth ratio of accelerators. Note that the released pretrained versions of smaller-scale Transformer-based models such as BERT and GPT-2 do not support processing of such long sequences, thus featuring lower arithmetic intensities.

This analysis of bandwidth underutilization indicates that HBM is not the ideal system for this workload. A similar argument applies for NVIDIA V100 GPU, which pairs some 112 TFLOPS with an HBM memory system having 900GB/s aggregate bandwidth.

Our Work. Observing this significant memory bandwidth underutilization, we propose to utilize the *flash memory system* (FMS) to design a more cost-effective large-scale language model training platform. However, naively replacing the HBM memory-system to a commodity SSD is not the right solution. In order to architect an efficient FMS for language model training, several challenges need to be addressed. First, an SSD has extremely-low bandwidth, especially when the access pattern is not sequential. Even if the access pattern is sequential, the sustained write bandwidth is often substantially lower than the peak bandwidth due to SSD garbage collection operations (GC) [33,55]. The second challenge is endurance. Because SSDs can only sustain a limited number of writes, utilizing SSD as a memory for the DNN training can significantly reduce the lifetime of SSDs. This problem becomes exacerbated when the access pattern is not sequential because random writes tend to increase the write amplification factor (WAF). Our work proposes solutions for these challenges and demonstrates that FMS can be effectively utilized for DNN training.

3 Overview of Behemoth

We design Behemoth to fully accommodate extreme-scale DNN models in a single node enabling data-parallel training. As illustrated in Figure 4, Behemoth consists of Compute Core, Tensor Buffer, and FMS. Compute Core is the computing substrate for training. Control Logic in Compute Core receives a sequence of commands from a host CPU and generates both computation and data transfer commands by parsing the sequence. The imminent weight and activation tensors are kept in the DRAM buffer (named Tensor Buffer) serving as a staging area. Upon receiving the read/write commands issued by the control logic, FMS controller executes the commands, retrieving and storing data in the NAND chips.



(a) Forward / Backward propagation

(b) Weight updating

Figure 5: Example walk-through of Behemoth (a) Forward and backward propagation (b) Weight updating

3.1 Training DNN Models on Behemoth

Most of the recent DNN frameworks employ a Python model. This model is pre-processed on the host CPU for analysis, extraction of layer information, and then generation of a sequence of commands that Behemoth can execute. This command sequence is communicated to Behemoth for execution.

Model Analysis. A user can define a DNN model to train using the PyTorch [52] format. In this model analysis step, information is collected about each layer's order, arguments used in the layer's operation, and input/output tensors to use. This step works similarly to the process of creating a static computation graph in Caffe [3] and TensorFlow [1].

Generating Command Sequences. Based on the collected model data, this step generates two types of command sequences: computation command sequence and direct memory access (DMA) command sequence. The DMA command sequence controls data transfers between Tensor Buffer and NAND flash devices. A DMA command includes fields about the direction of transfer (read/write), logical block address (LBA) of the NAND device, and Tensor Buffer address. The computation command sequence lists operation commands to perform on Compute Core. A computation command includes fields about the type of the layer (e.g., Fully-connected (FC), Convolution (Conv)), and the address in Tensor Buffer where the layer's input and output tensors will be stored. Both command sequences are transferred to Behemoth to be saved in the region for the non-volatile stream (NV-Stream in Section 4.1).

3.2 Hardware Components of Behemoth

Behemoth platform consists of a single Weight Node and multiple Activation Nodes to fully utilize the bandwidth between Tensor Buffer and NAND flash. Each node consists of Compute Core, Tensor Buffer, and NAND flash. Weight Node stores the weights of the target training model in the NAND flash. The Activation Nodes create activation tensors during forward propagation and store them locally in the NAND flash for reuse during backward propagation.

Compute Core. Compute Core is not bound to a specific DNN accelerator architecture. Thus, we assume a generic DNN accelerator that abstracts popular commercial/academic accelerators [9,10,24,27,58]. The DNN accelerator is specialized for DNN processing and performs matrix multiplication and addition for weights and activations. It consists of a 2D array of processing elements (PEs), where each PE can perform a single MAC operation every clock cycle. Behemoth assumes a weight stationary dataflow architecture [9], where weights are directly loaded from Tensor Buffer and kept in the local registers inside the PE. Every cycle, a new input is provided to the PEs from the SRAM buffer of Compute Core. This input is multiplied by the corresponding weight in the PE, and the result is accumulated. Once the computation is done, the output values are transferred to the SRAM buffer, and eventually to Tensor Buffer.

Control Logic. Control Logic is responsible for sequencing computation commands and orchestrating data transfers between SRAM buffer, Tensor Buffer, and FMS. Specifically, it decodes the commands provided by the host CPU and inspects if this command can be scheduled (i.e., satisfies all dependencies). If so, Control Logic dispatches this command to Compute Core (if it is a computation command) or DRAM Controller or FMS Controller (if it is a DMA command) to initiate the requested DMA. Note that this is a very simple logic, which sequences the commands in order.

Tensor Buffer. Tensor Buffer is a DRAM region that serves as a staging area between Compute Core and FMS. The primary role of his buffer is to smooth the traffic between FMS and Compute Core simply. Thus, Tensor Buffer only stores temporal data and does not require persistence.

Flash Memory System (FMS). FMS is the main storage in Behemoth replacing the HBM in the conventional DNN accelerators (e.g., TPU). As in SSDs, this component includes a set of NAND chips. However, unlike the conventional SSD, it has a hardware-based FMS controller that replaces the flash translation layer (FTL) running on general-purpose cores. This component interfaces with Control Logic and transfers data to Tensor Buffer. The details of this component are explained in Section 4.

3.3 Example Execution Walk-Through

Figure 5(a) and (b) illustrate the process of forward and backward propagation in Behemoth. In what follows, we explain this process in greater detail.

Forward Propagation. Executing forward propagation in Behemoth consists of 6 steps, which can be overlapped. While computation is being performed on Activation Node, weights on Weight Node are prefetched. ❶ Executing a layer starts with reading the weights stored in the NAND flash of Weight Node into Tensor Buffer. ❷ The weight tensor is broadcasted and transferred to Tensor Buffer of Activation Node. ❸ Then, the weight tensor is loaded into on-chip SRAM, and ❹ computation begins. When computation is completed on Activation Node, ❺ the activation tensor stored in SRAM is copied to Tensor Buffer. ❻ Finally, the activation tensor in Tensor Buffer is written to NAND flash for reuse during backward propagation, and the weight tensor is deallocated from Tensor Buffer.

Backward Propagation. Figure 5(a) and (b) show the process of backward propagation (labeled with empty circles). It is divided into two parts. The first part is executed for each layer, and the second one only once at the end of each iteration. Like forward propagation, all steps are pipelined and operated in parallel. The processes of ① and ② are the same as forward propagation. Once the weight tensor is received from Weight Node, ③ Activation Node reads the activation tensor of the corresponding layer stored in the NAND flash during the forward propagation into Tensor Buffer. ④ Upon completion of loading the activation tensor, both activation and weight tensors are loaded into SRAM, and then ⑤ computation is started. When the operation is completed, ⑥ the resulting gradient tensor is stored in Tensor Buffer.

After the calculation of all layers is completed, ⓐ the final weight gradient tensor is transferred from Activation Node to Tensor Buffer of Weight Node. After confirming that all weight gradients have been received, ⓑ Weight Node loads the weight gradients into SRAM and ⓒ updates the training results of the iteration to the weights. ⓓ The updated weights are stored to SRAM and ⓔ written to the NAND flash for the next iteration.

3.4 DNN Model Coverage

Behemoth targets training workloads for extreme-scale models whose memory bandwidth requirement does not exceed the sustainable bandwidth of FMS. Suitability for other models can be determined by analyzing their arithmetic intensity. Our model analyzer can compare the arithmetic intensity of a model with Behemoth's compute-to-bandwidth ratio (i.e., FLOPS / GB/s) to check whether the model is provided with enough bandwidth from FMS [49].

The key enabler of FMS as a storage medium of tensors is a higher degree of data reuse resulting from long sequence length. State-of-the-art models that are capable of handling such long sequences have an extremely large size. Small-sized models exhibit much lower arithmetic intensity, hence not being the primary target of Behemoth. For example, small NLP models [18, 40, 53] have limited sequence length (e.g., 512). This is much smaller than 2048 for GPT-3. Conventional vision models such as ResNet heavily utilize convolution layers. Converting convolutions into matrix multiplications through convolution lowering [12] results in a dimension m that is much smaller than those of FC layers, hence failing to provide enough bandwidth from FMS. This issue can be mitigated to a certain extent by increasing the batch size, which in turn increases the degree of reuse for weight parameters.

Table 1: DNN training data types and multi-stream support

#: Stream name (Act. Node / Weight Node)	**Persistency**	**Retention**	**Access permission**	
			Host	**Behemoth**
1: NV-Stream (Training inputs / –)	Non-volatile	Years	Append-only seq. write	Read only
2: V-Stream (Activations / Interm. weights)	Volatile	Minutes	N/A	Read & Append-only seq. write
3: NV-Stream (– / Trained weights)	Non-volatile	Years	Read only	Read & Append-only seq. write

4 Architecting Specialized Flash Memory System (FMS) for DNN Training

As stated in Section 2, the main challenges in adopting NAND flash memories for the language model training is the limited bandwidth and the endurance of the NAND flash memories. This section presents our solution to the two challenges and explains FMS's implementation in detail.

4.1 Improving Effective Bandwidth of FMS

Modern NAND flash memory-based storage adopts a number of flash channels and ways to increase bandwidth and capacity. A host interface for the storage has also run a neck and neck race with the storage's internal bandwidth to meet the user's performance requirement. In terms of hardware bandwidth of the NAND flash-based storage, the interface speed and the number of NAND channels, as well as the number of NAND chips attached to a channel, define the maximum reachable speed of a NAND flash-based storage.

Technically, the bandwidth of a flash-based memory system can be improved by utilizing a large number of NAND channels as well as the sufficient number of NAND chips per channel to saturate the channel bandwidth. Indeed, some recent proposals [11, 25] demonstrate that it is possible to build a high-bandwidth NAND system by increasing the number of channels or the channel bandwidth itself. However, to fully utilize the high peak bandwidth of such a NAND device, one needs to i) make writing sequential as much as it can and ii) prevent the slow NAND firmware running on a general-purpose processor from being a bottleneck [11, 71].

Data type Separation. Generally, it is challenging to identify

Table 2: NAND block layout for a chip and multi-stream attributes of Activation Node

NAND Block Layout					Stream attributes	
Plane PBN	0	1	. . .	7	Capacity	P/E cycle/ Retention
0	FTL Metadata (LBN2PBN map, PB metadata, etc)					
9						
10	1: NV-Stream (training input)				249 GiB	50K / 1 year
92						
93	2: V-Stream (activation data)				1737 GiB	2M / 1 day
671						
672	Reserved blocks for bad block replacement					
682						

a workload's data access patterns before it is executed. However, a DNN training accelerator (or NPU) has a deterministic data access pattern that can be statically analyzed. The DNN training accelerator accesses three types of data, each having a very specific characteristic as listed in Table 1.

First, Activation Node's FMS houses two types of data: training input data and the activation data. Here, training input data is a set of text data used as training inputs of the DNN model. This data is written by a host before the training starts and then discarded once the training finishes. Activation data are written by Compute Core of the FMS platform during a forward path of the training and then consumed during a backward path of the training. The data is not written or read by the host, and the life cycle of these data is very short (in order of seconds, or minutes at most) as they are lived only within a single iteration.

Similar to Activation Node, FMS of Weight Node also houses two types of data. First, it holds the final model weights, that is only updated at the end of the training (or after a certain number of iterations to checkpoint the intermediate weights), and then later read by the host CPU. Second, it stores intermediate model weights that are updated at the end of each iteration.

Since two data types housed in the same device (i.e., training input data vs. activation data in Activation Node; final trained weights vs. intermediate model weights in Weight Node) have completely different characteristics, it is beneficial to separate them to two logically isolated spaces as in multi-stream SSDs [30, 33, 55]. In particular, we employ two streams: the non-volatile stream (NV-Stream) and the volatile stream (V-Stream). Table 2 summarizes a block layout for a single NAND chip and capabilities of each data stream of Activation Node. Weight Node layout is the same, but only the capacity and physical block number (PBN) division are different from storing the weight result. The streams are physically separated by block address boundary, hence able to function as if each stream were an individual storage space, and thus each single stream can have their own logical address space, access permission, and allowed P/E cycle based on retention requirement, which is determined by characteristics of the DNN training data. This separation of different data types enables several useful optimizations as follows.

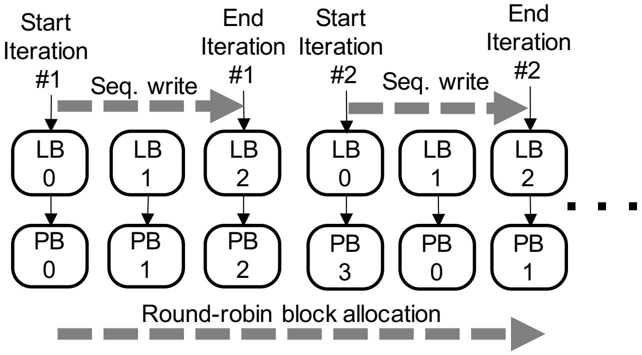


Figure 6: Sequential append-only writes with RR allocation

Lightweight Flash Translation Layer. Two major functionalities of the flash translation layer (FTL) are garbage collection (GC) and wear-leveling. However, since writes to each data for our FMS is guaranteed to be sequential, complicated garbage collection and wear-leveling are mostly unnecessary. Thus, we remove the FTL's garbage collection functionality and then replace the wear-leveling block allocator with a simple round-robin block allocator shown in Figure 6. For example, as shown in the figure, assume that FMS has four Physical Blocks (PBs), and the host writes three Logical Blocks (LBs) sequentially during a single DNN training iteration. During the first training iteration, FMS uses PB 0, 1, and 2 by mapping to LB 0, 1, and 2. And then, in the second training iteration, FMS allocates PBs according to Round-Robin (RR) policy from PB 3 to PB 0, 1 for writing to LB 0, 1, 2 of the host. This simple RR block allocation policy strictly levels wear of all NAND blocks. The utilization of this simple wear-leveling scheme as well as the removal of the garbage collection greatly simplified the FTL.

Hardware Automation of Write Path. Most modern commercial SSD controllers adopt a read automation feature that accelerates the read operation exploiting specialized hardware that substitutes (part of) the read path of the SSD firmware [11]. On the other hand, the write data path still relies on the firmware with high overhead or is merely partially replaced by hardware logic with substantial functional restrictions [28, 70]. This is mostly because the write data path is much more complex than the read path. For example, the write path needs to perform many additional operations compared to the read path. Specifically, it needs to i) reserve NAND blocks for the GC operations, ii) perform wear-leveling to ensure that all NAND blocks are used evenly, iii) guarantee data consistency among the internal R/W operations generated by the GC, the wear-leveling, and the user write commands, iv) manage metadata necessary for recovery from expected or unexpected power-reset, and v) handle exceptions for P/E failures.

However, we note that our FMS's common write data path does not need to perform many additional operations than the read path. It does not require a garbage collection and utilizes a very simple wear-leveling block allocator. Metadata

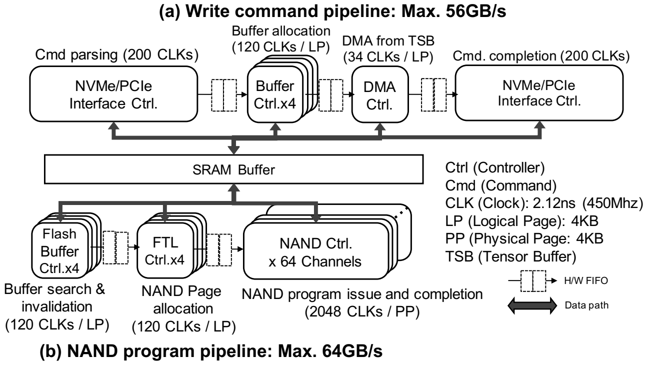

(a) Write command pipeline: Max. 56GB/s
Cmd parsing (200 CLKs)
Buffer allocation (120 CLKs / LP)
DMA from TSB (34 CLKs / LP)
Cmd. completion (200 CLKs)
NVMe/PCIe Interface Ctrl.
Buffer Ctrl.x4
DMA Ctrl.
NVMe/PCIe Interface Ctrl.
SRAM Buffer
Ctrl (Controller)
Cmd (Command)
CLK (Clock): 2.12ns (450Mhz)
LP (Logical Page): 4KB
PP (Physical Page: 4KB
TSB (Tensor Buffer)
Flash Buffer Ctrl.x4
FTL Ctrl.x4
NAND Ctrl. x 64 Channels
H/W FIFO
Data path
Buffer search & invalidation (120 CLKs / LP)
NAND Page allocation (120 CLKs / LP)
NAND program issue and completion (2048 CLKs / PP)
(b) NAND program pipeline: Max. 64GB/s

Figure 7: Automated write data path of FMS

management is not on a critical path and unnecessary for temporary data such as activation data and intermediate weight data. Finally, the exception handling is a rare event. Thus, it becomes relatively easy to automate the write data path by utilizing specialized hardware. By doing so, it is possible to prevent the firmware from being a bottleneck.

Figure 7 shows hardware pipeline stages for the write path of FMS and a timing of each pipeline stage. The automated write path is composed of (a) write command pipeline that transfers data from Tensor Buffer to an SRAM buffer in the FMS controller and (b) NAND program pipeline that programs data in the SRAM to NANDs. As shown in Figure 7, we carefully design each pipeline stage to meet a memory bandwidth requirement for DNN training. In particular, a buffer search/invalidation and a NAND page allocation stage of the NAND program pipeline, which was handled by the firmware of an existing SSD product [11], have been completely replaced with FMS controller logic.

Note that the hardware pipeline does not update the metadata necessary for persistence. For temporary data that accounts for the most portion of FMS, persistency support is not performance-critical as the iteration can be re-executed from the last checkpoint. For the data that needs the storage to be persistent, a user can make an explicit request (e.g., flush command [48] after writes) that initiates the firmware to ensure that the data is persistent.

4.2 Improving Endurance of FMS

The endurance of a NAND flash based storage relies on the program and erase (P/E) cycle for NAND blocks. The P/E operation wears the NAND block, accelerating the leakage of the electrons in the NAND cells. Additionally, such damage from the P/E cycles is cumulative and irreversible and gives rise to a myriad of read error bits, which cannot be corrected by an ECC engine of a storage controller.

FMS essentially utilizes a flash as a temporary buffer for the activation and intermediate weights. At a glance, it may seem like such a frequently re-programmed value will substantially affect the lifetime of the SSDs, which are often defined as the number of P/E cycles that a NAND cell can sustain. However, we argue that this is not the case (and present a quantitative analysis in Section 5.3).

Typically, each P/E cycle damages a NAND cell, and such a damage keeps reducing the retention time of the cell. Once the retention time falls below the guaranteed retention time (e.g., 1 year in consumer-grade SSDs [14]), the cell is considered having failed. At that point, the cell may not be suitable for storing the data for a long time; however, it is likely to be still sufficient to store the data that will only last for a few minutes. In light of device physics, the programmed NAND flash cells gradually lose their electrons from a floating gate over time, and in case a cell is damaged by the repetitive P/E cycles, the cell loses charge faster [26, 57]. However, with the low retention requirement, the cell can still maintain sufficient level of charges until the end of the retention time. In fact, many studies [6, 43, 45] already demonstrated that the SSD endurance (# of P/E cycles) is larger when the retention requirement is relaxed. Note that the benefits of reduced retention does not require additional hardware resources (e.g., more complex ECC engines or an extra over-provisioning space).

Considering that FMS (V-Stream data) requires only a few minutes (e.g., 5 minutes) of retention time that is almost five orders of magnitude smaller than a typical consumer-grade SSD, it is expected that the cell can sustain a substantially large number of P/E cycles before a cell's minimum retention time to fall below a few minutes.

5 Evaluation

5.1 Methodology

We evaluate our platform's effectiveness by i) comparing our platform's memory cost to conventional TPU-based DNN training system, and ii) comparing our platform with the specialized FMS to the hypothetical platform with conventional SSDs.

Simulation Framework. To model the performance of the Behemoth platform, we utilize MAESTRO [36] for Compute Core and MQSim [63] for modeling our FMS. Specifically, we utilize PyTorch [52] to obtain the layer dimensions of the large-scale language models, and then use that information on MAESTRO [36] to obtain the number of cycles that Compute Core (PE arrays with weight-stationary dataflow) needs for the computation of a specific layer in the language model. Then, based on this information, we generated the traces for our FMS and fed these traces to the MQSim (modified to support our proposed changes detailed in Section 4) to obtain the flash memory-related statistics. Both simulators are validated with NPU hardware RTL [9, 37] and a commercial SSD product [13], exhibiting an average of 5% errors [32, 36].

Workloads. We evaluate twelve workloads representing three types of widely adopted transformer models. Table 3 lists these models. As listed in Table 3, we evaluate our work with two types of models: (a) BERT/GPT-like and (b) T5-like. We

Table 3: DNN models evaluated with Behemoth. We use a sequence length of 2048 (tokens) for each model.

Model	Size	Total act. (GB)	Total weight (GB)	PFLOP
BERT/GPT3-like [5, 18]	1×1	44	350	2.15
	1×2	88	698	4.42
	1×4	175	1393	8.56
	2×1	88	1395	8.56
	2×2	175	2786	17.12
	2×4	349	5569	34.21
T5-like [54]	1×1	40	305	0.62
	1×2	80	609	1.25
	1×4	160	1218	2.49
	2×1	80	1218	2.49
	2×2	160	2436	4.99
	2×4	319	4871	9.97

enlarge the dimensions in FC layers of the models and/or stack more encoders/decoders, respectively, for diverse comparison. $W \times D$ notation is defined as W-fold enlarged FC layers dimension (width) and D times increased depth was implemented by stacking more encoders/decoders or transformers blocks. Our workloads present various Transformer-based models. Transformer is a key enabling primitive for DNN to advance the state-of-the-art in the domains of NLP [4, 5, 16–18, 31, 34, 39, 40, 44, 54, 59, 66, 68, 72], image detection [7, 19], point cloud [21], and recommendation systems [62]. All of these models can be classified into either BERT/GPT-like or T5-like.

The rationale behind binding BERT/GPT-like models is as follows. They do not use a combined transformer but separately utilize encoders and decoders. While they have certain different characteristics, such as in the computation process, their structure is identical, as described in Figure 3. Thus, the two systems' total activation and weight are equal when having the same number of parameters. The structure of the T5-like models makes it difficult to exactly match the number of parameters with the encoder-only or decoder-only models. In turn, they show the different sizes of activation and weight. The sequence length of 2048 tokens can be interpreted as roughly 2048 words in a sequence. The sequence of 2048 tokens comprises a batch size of 1, and the activation size is calculated for one batch.

5.2 Memory Cost Evaluation

Baseline NPU with HBM DRAM. In order to train very large scale language models like GPT3, existing HBM-based neural processing accelerators need to be configured in a model-parallel manner. In such a configuration, each TPU is assigned a portion of the model (i.e., a distinct set of consecutive layers). Once a neural processing accelerator finishes the computation for the layers it is assigned to, it passes its outputs to the other neural processing accelerator in charge of the following layers. This model parallelism makes it possible to train a very large model that does not fit in a single neural processing accelerator's HBM-based memory. One notable drawback of this approach is the difficulty in load-balancing. For example, training GPT-3 requires a minimum of 393GB storage, which translates to a 24-stage pipeline assuming that a hardware corresponding to each stage has a 16GB HBM memory system. Depending on the nature of the model, partitioning the model into 24 slices in a load-balanced manner may be very difficult, if not impossible. For the cost comparison, we configure an NPU resembling the structure of TPU, as shown in Table 4. A single device here has 16 compute cores, each having 52.5TFLOPS peak throughput. Each of these cores is attached to a single, 16GB HBM memory whose peak bandwidth is 300 GB/s. To achieve sufficiently high throughput, multiple copies of these devices are utilized in parallel.

Table 4: Platform configurations for the cost evaluation of Behemoth.

NPU Parameters		
Number of cores	16 cores (52.5 TFLOPs per core)	
Number of PEs	524,288	
Peak throughput	840 TFLOPs	
Host I/F conf.	PCIe Gen4 × 32 lane [51]	
Memory Parameters		
	Resembled TPU [27]	**Behemoth**
Buffer conf.	16GB HBM	16GB DDR4 DRAM + 2TB NAND flash
Peak bandwidth	300GB/s	50GB/s
Compute Parameters		
Parallel comp. method	Model parallelism	Data parallelism

Behemoth with FMS. Unlike the baseline, Behemoth platform utilizes data parallelism, which enables the complete model to be trained on a single device, and thus does not suffer from load imbalance issues. We configure the single device for the cost comparison to having 16 compute cores, each having 52.5 TFLOPS peak throughput as in the baseline NPU. However, instead of HBM, 16 computation cores in Behemoth device share a single FMS with 2 TB capacity and 50 GB/s peak bandwidth. In addition to this device utilized as an Activation Node (see Figure 5), there is a separate device utilized as a Weight Node. However, since there will be many Activation Nodes that share a single Weight Node, the cost of the Weight Node is amortized. Note that we carefully configured 16 cores to share a single FMS. To determine the number of cores to share a FMS, we selected the maximum number of cores that satisfies the following criteria: (a) the size of the data fits inside our storage, and (b) the data transfer between the FMS and compute cores can be completely hidden.

Cost Evaluation. Figure 8 demonstrates the difference of memory cost between Behemoth and TPU v3 [27], a popular training accelerator deployed by Google. We assumed that the user utilizes the 432 Behemoths and 864 TPUs that are just enough to train each workload in 10 days. For this calculation, we assumed HBM device cost to be $20/GB [23], DDR4 DRAM device cost to be $4/GB [47], and flash device cost to be $0.67/GB. Note that the cost of Behemoth SSD, using 128Gb V-NAND based SLC, was set as four times a commercial SSD price (0.167$/GB) which uses 512Gb V-

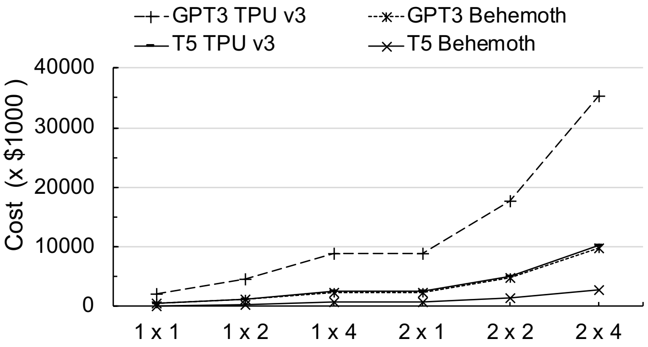


Figure 8: Memory cost comparison between TPU v3 [27] and Behemoth. $W \times D$ in the figure illustrates that the dimension of each layer is increased by W times and the number of layers is increased by D times.

NAND TLC [15]. It can be seen from the figure that the cost gap between the two systems increases commensurate with model size. The maximum difference of $25.7M for BERT/GPT3-like models and $7.5M for T5-like models.

5.3 FMS Evaluation

Configuration. We compare a DNN training platform utilizing BehemothFMS and the other utilizing the conventional storage using commodity SSDs. The configuration details are tabulated in Table 5.

Impact of FMS on Training Throughput. We compare the training throughput of a DNN training platform utilizing BehemothFMS and one that utilizes the commodity SSDs. As shown in Figure 9, the DNN training platform with BehemothFMS performance is close to the ideal case where there is zero overhead from memory system accesses. On the other hand, a DNN training platform with commodity SSDs often achieves much lower training throughput in many workloads. This is because the baseline SSD achieves limited throughput bottlenecked by an SSD firmware. To be exact, since a write data path of the baseline SSDs requires a minimum of 1.45μs to write a 4KB page, a single commodity SSD device's write throughput is limited to about 2.75GB/s despite its high aggregate channel bandwidth or external interface bandwidth. It aggregates four SSD device's throughput results in about 11.0GB/s bandwidth, which is substantially smaller than the 50GB/s bandwidth that Behemoth can provide. Note that the speedup of BehemothFMS is a little lower on wider models (e.g., 2×1, 2×2, 2×4). This is because wider models have even higher data reuse (see Section 2), thus requiring even less memory or storage bandwidth.

Endurance for Training Workloads. Figure 10 shows lifespan of tensors that are generated during DNN training. As shown in the figure, all tensors created during the DNN training have a lifespan of up to a single iteration period [22]. Therefore, the longest lifespan of tensors equals the retention time necessary for the V-Stream of FMS: 41 sec. Based on the previous studies [6, 43, 45] that demonstrate the number of P/E cycle of NAND can be increased by at least 40× [6] (up-to 600× [43]) if 1-year retention is reduced to 3 days, we also conservatively assume that the P/E cycle of our SSD is improved by 40 times, despite our retention requirement (i.e., less than a minute) is much shorter than three days. The Samsung Z-SSD [13] has 50K P/E cycles, and improving its P/E cycles by 40× results in the 2M P/E cycles. Two million P/E cycles on 1.85TB storage for V-Stream of FMS translates to the 3,700,000 TBW (TeraBytes Written). Considering that Behemoth FMS can sustain up to 17.6GB/s write bandwidth on average for T5-like models, Behemoth FMS is guaranteed to function for 6.6 years (i.e., 3.7M TBW / (17.6GB/s) = 210M seconds = 6.6 years). As shown in Figure 11, this is an even longer period than the 5-year warranty of typical commercial SSDs. Here, note that we assume the write amplification factor (WAF) of one because Behemoth only performs monotonic sequential writes and reads during the entire DNN training iteration without GC operations [29] as shown in Figure 12.

Table 5: FMS and conventional storage configuration.

Storage Parameters		
	Behemoth FMS	**Baseline SSD**
NAND Configurations	2TB, 64 channels, 2 chips/channel, 1 die/chip	500GB, 16 channels, 2 chips/channel, 1 die/chip
Channel Speed Rate	1200MT/s (MT/s: Mega Transfers per Second [20])	
NAND Structure	128Gb SLC / die: 8 planes / die, 683 blocks / plane, 768 pages / block, 4KB page	
NAND Latency	Read: 3μs, Program: 100μs, Block erase: 5ms	
Buffer Configurations	SRAM 16MB: 6MB for FTL metadata, 10MB for I/O buffer	DRAM 512GB: FTL metadata SRAM 8MB: I/O buffer, GC Buffer
FTL Schemes	Block mapping	Page mapping, Preemtible GC [38]
OP ratio	N/A	7%
Firmware Latency	N/A	Write: 1.45μs / a page (4KB)
Contoller Latency	Read: 1.93μs / an NVMe Cmd, Write: 1.18μs / an NVMe Cmd	Read: 1.93μs / an NVMe Cmd

6 Related Work

Heterogeneous Memory System for Tensor Management. Due to the memory capacity wall, researchers train the model with a limited number of parameters and batch sizes that the memory capacity allows, or parallelize the model by distributing the data needed for computation across multiple DNN training devices. However, training on a small model shows low accuracy, and the distributed learning of large models through multiple devices is a waste of memory bandwidth compared to memory capacity usage in several cases. To address the memory capacity wall, several proposals [22, 56, 67]

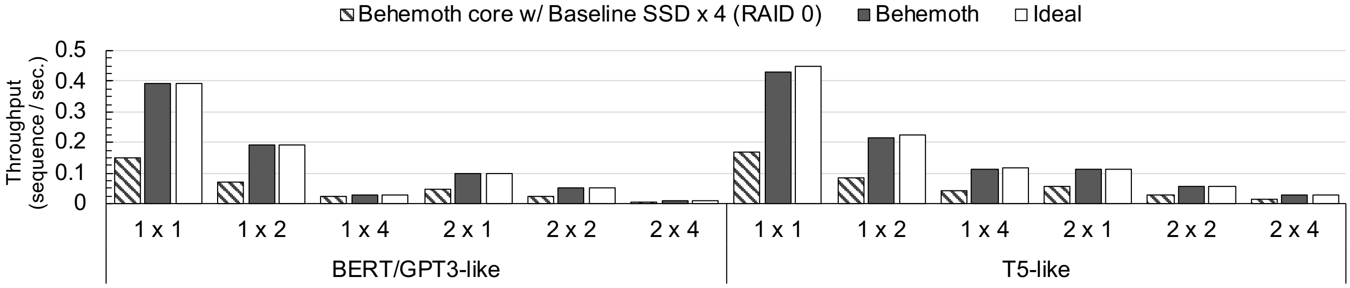
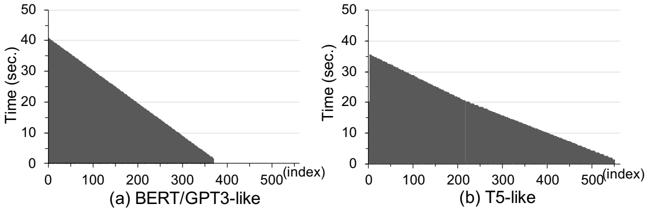


Figure 9: DNN training throughput of 432 Behemoths over various model sizes.



(a) BERT/GPT3-like

(b) T5-like

Figure 10: Tensor lifespan

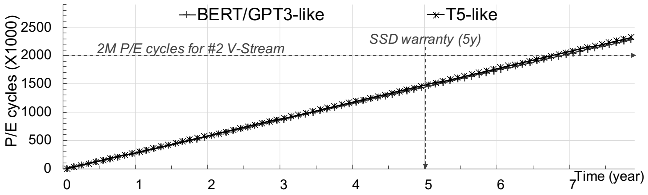
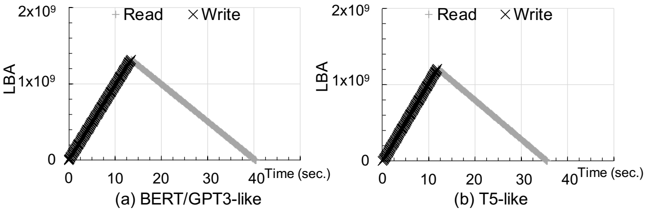


Figure 11: Behemoth FMS endurance



(a) BERT/GPT3-like

(b) T5-like

Figure 12: Block access pattern of DNN training workloads running on Behemoth FMS

introduce effective memory management techniques that migrate tensors in a heterogeneous memory system. HALO [22] analyzes the hotness and lifetime of tensors and constructs an offloading schedule for each tensor, which focuses on how to migrate and place tensors across the heterogeneous memory nodes. vDNN [56] offloads tensors to the host memory during the forward pass and prefetches tensors from the host memory to be sent to the GPU during the backward pass. SuperNeuron [67] partially adopts the idea of vDNN by only offloading/prefetching marked tensors and recomputing unmarked tensors during the backward pass. These researches utilize another DRAM memory as offloading media. However, this is more of a one-time solution since the large language model does not fit in the host main memory. Behemoth adopts the idea of offloading and prefetching tensors according to their lifetime and dependency, but completely overcomes the memory capacity wall by using a dense NAND-based flash memory system configured for the high bandwidth.

Storage-Centric Machine Learning System. Several proposals [42, 46, 61, 70] use SSD as a secondary storage in the compute core to run large-scale applications. BLAS-on-flash [61] builds a library to achieve efficient flash memory speed in computing machine learning algorithms (e.g., ISLE, XML) that are used on large datasets. Cognitive SSD [42] constructs an engine designed for unstructured data retrieval involving DNN inference in flash memory devices. In contrast, Behemoth proposes the flash-based memory system for the extreme-scale neural-network language models with over hundreds of billions of parameters.

7 Conclusion

Recent DNN models are getting wider and deeper, increasing the memory requirements for training. This trend is especially obvious in NLP with extreme-scale models showing exponential growth in its size. However, conventional DNN training platforms such as NVIDIA GPUs or Google TPUs provide insufficient storage capacity, which leads to excessive cost and memory bandwidth underutilization. To address this problem, we propose Behemoth, a flash-based memory system for a cost-effective training platform targeting extreme-scale DNN models. Behemoth overcomes the low-bandwidth and endurance problem of SSDs by separating data according to their characteristics. This enables a simplified firmware and hardware automation of the write path, which significantly improves the bandwidth. Furthermore, by exploiting the much shorter required retention time, we also showed that the SSD could be safely utilized for over six years. In the end, this Behemoth flash memory system based DNN training platform achieves a much smaller memory system cost than the conventional DNN training platform utilizing HBM devices.

Acknowledgments

We thank Suparna Bhattacharya for shepherding this paper. We also thank Young H. Oh for his help with DNN training. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2020R1A2C3010663). Jae W. Lee is the corresponding author.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283. USENIX Association, 2016.

[2] Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. Unsupervised Neural Machine Translation. *arXiv e-prints*, page arXiv:1710.11041, October 2017.

[3] BAIR. Caffe: Deep learning framework. https://caffe.berkeleyvision.org.

[4] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv e-prints*, April 2020.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv e-prints*, May 2020.

[6] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Adrian Cristal, Osman S. Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *Proceedings of the 2012 IEEE 30th International Conference on Computer Design*, pages 94–101, 2012.

[7] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-End Object Detection with Transformers. *arXiv e-prints*, page arXiv:2005.12872, May 2020.

[8] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *arXiv e-prints*, page arXiv:1604.06174, April 2016.

[9] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, page 367–379. IEEE Press, 2016.

[10] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DianNao family: Energy-efficient hardware accelerators for machine learning. *Commun. ACM*, 59(11):105–112, October 2016.

[11] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, Jaehong Kim, Jaechun Park, Ki-Whan Song, Ki-Tae Park, Sangyeun Cho, Hwaseok Oh, Daniel DG Lee, Jin-Hyeok Choi, and Jaeheon Jeong. A flash memory controller for 15μs ultra-low-latency SSD using high-speed 3D NAND flash with 3μs read time. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 338–340. IEEE, 2018.

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv e-prints*, page arXiv:1410.0759, October 2014.

[13] Samsung Z-SSD SZ985. https://www.samsung.com/semiconductor/global.semi.static/Brochure_Samsung_S-ZZD_SZ985_1804.pdf.

[14] Samsung SSD 980 PRO. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/980pro/.

[15] SSD price: Samsung SSD 970 EVO. https://www.amazon.com/Samsung-970-EVO-1TB-MZ-V7E1T0BW/dp/B07BN217QG/ref=sr_1_2?dchild=1&keywords=970+evo&qid=1600645757&sr=8-2.

[16] Zihang Dai, Guokun Lai, Yiming Yang, and Quoc V. Le. Funnel-Transformer: Filtering out Sequential Redundancy for Efficient Language Processing. *arXiv e-prints*, June 2020.

[17] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv e-prints*, January 2019.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv e-prints*, October 2018.

[19] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and

Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *arXiv e-prints*, page arXiv:2010.11929, October 2020.

[20] Alan Freedman. MT/sec. *The Computer Desktop Encyclopedia*. https://www.computerlanguage.com/results.php?definition=MT/sec.

[21] Meng-Hao Guo, Jun-Xiong Cai, Zheng-Ning Liu, Tai-Jiang Mu, Ralph R. Martin, and Shi-Min Hu. PCT: Point Cloud Transformer. *arXiv e-prints*, page arXiv:2012.09688, December 2020.

[22] Myeonggyun Han, Jihoon. Hyun, Seongbeom. Park, and Woongki. Baek. Hotness- and lifetime-aware data placement and migration for high-performance deep learning on heterogeneous memory systems. *IEEE Transactions on Computers*, 69(3):377–391, 2020.

[23] Radeon VII 16GB HBM 2 memory cost around $320. https://www.fudzilla.com/news/graphics/48019-radeon-vii-16gb-hbm-2-memory-cost-around-320.

[24] Brian Hickmann, Jieasheng Chen, Michael Rotzin, Andrew Yang, Maciej Urbanski, and Sasikanth Avancha. Intel nervana neural network processor-t (NNP-T) fused floating point many-term dot product. In *Proceedings of the 2020 IEEE 27th Symposium on Computer Arithmetic*, pages 133–136, 2020.

[25] Jae-Woo Im, Woo-Pyo Jeong, Doo-Hyun Kim, Sang-Wan Nam, Dong-Kyo Shim, Myung-Hoon Choi, Hyun-Jun Yoon, Dae-Han Kim, You-Se Kim, Hyun-Wook Park, Dong-Hun Kwak, Sang-Won Park, Seok-Min Yoon, Wook-Ghee Hahn, Jin-Ho Ryu, Sang-Won Shim, Kyung-Tae Kang, Sung-Ho Choi, Jeong-Don Ihm, Young-Sun Min, In-Mo Kim, Doo-Sub Lee, Ji-Ho Cho, Oh-Suk Kwon, Ji-Sang Lee, Moo-Sung Kim, Sang-Hyun Joo, Jae-Hoon Jang, Sang-Won Hwang, Dae-Seok Byeon, Hyang-Ja Yang, Ki-Tae Park, Kye-Hyun Kyung, and Jeong-Hyuk Choi. A 128Gb 3b/cell V-NAND flash memory with 1Gb/s I/O rate. In *Proceedings of the 2015 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 1–3, 2015.

[26] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 61–74. USENIX Association, 2014.

[27] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, page 1–12. ACM, 2017.

[28] Myoungsoo Jung. OpenExpress: Fully hardware automated open research framework for future fast NVMe devices. In *Proceedings of the 2020 USENIX Annual Technical Conference*, pages 649–656. USENIX Association, July 2020.

[29] Behemoth FMS storage trace. https://github.com/SNU-ARC/storage-trace.

[30] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*. USENIX Association, June 2014.

[31] Douwe Kiela, Suvrat Bhooshan, Hamed Firooz, and Davide Testuggine. Supervised multimodal bitransformers for classifying images and text. *arXiv e-prints*, September 2019.

[32] Shine Kim, Jonghyun Bae, Hakbeom Jang, Wenjing Jin, Jeonghun Gong, Seungyeon Lee, Tae Jun Ham, and Jae W. Lee. Practical erase suspension for modern low-latency SSDs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, page 813–820. USENIX Association, July 2019.

[33] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully automatic stream management for multi-streamed SSDs using program contexts. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, pages 295–308. USENIX Association, February 2019.

[34] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient Transformer. *arXiv e-prints*, January 2020.

[35] Wojciech Kryściński, Nitish Shirish Keskar, Bryan McCann, Caiming Xiong, and Richard Socher. Neural Text Summarization: A Critical Evaluation. *arXiv e-prints*, page arXiv:1908.08960, August 2019.

[36] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, page 754–768. ACM, 2019.

[37] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 461–475, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, and Jongman Kim. Preemptible I/O scheduling of garbage collection for solid state drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(2):IEEE, 247–260, 2013.

[39] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. *arXiv e-prints*, June 2020.

[40] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv e-prints*, October 2019.

[41] Piji Li, Wai Lam, Lidong Bing, and Zihao Wang. Deep Recurrent Generative Decoder for Abstractive Text Summarization. *arXiv e-prints*, page arXiv:1708.00625, August 2017.

[42] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *Proceedings of the 2019 USENIX Annual Technical Conference*, pages 395–410. USENIX Association, 2019.

[43] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, page 11. USENIX Association, 2012.

[44] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv e-prints*, July 2019.

[45] Yixin Luo, Yu Cai, Saugata Ghose, Jongmoo Choi, and Onur Mutlu. WARM: Improving NAND flash memory lifetime with write-hotness aware retention management. In *2015 31st Symposium on Mass Storage Systems and Technologies*, pages 1–14, 2015.

[46] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia de Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. DeepStore: In-storage acceleration for intelligent queries. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, page 224–238. ACM, 2019.

[47] Newegg.com. https://www.newegg.com.

[48] NVM Express 1.4. https://nvmexpress.org.

[49] Young H. Oh, Seonghak Kim, Yunho Jin, Sam Son, Jonghyun Bae, Jongsung Lee, Yeonhong Park, Dong Uk Kim, Tae Jun Ham, and Jae W. Lee. Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

[50] Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. Scaling Neural Machine Translation. *arXiv e-prints*, page arXiv:1806.00187, June 2018.

[51] PCI Express 4. https://pcisig.com.

[52] PyTorch. https://pytorch.org.

[53] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[54] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text Transformer. *arXiv e-prints*, October 2019.

[55] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. FStream: Managing flash streams in the file system. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, pages 257–264. USENIX Association, 2018.

[56] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 18:1–18:13. IEEE, 2016.

[57] C. Sandhya, Apoorva B. Oak, Nihit Chattar, Udayan Ganguly, C. Olsen, S. M. Seutter, L. Date, R. Hung, Juzer Vasi, and Souvik Mahapatra. Study of P/E cycling endurance induced degradation in SANOS memories under NAND (FN/FN) operation. *IEEE Transactions on Electron Devices*, 57(7):1548–1558, July 2010.

[58] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, page 14–27. ACM, 2019.

[59] Nitish Shirish Keskar, Bryan McCann, Lav R. Varshney, Caiming Xiong, and Richard Socher. CTRL: A conditional transformer language model for controllable generation. *arXiv e-prints*, September 2019.

[60] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv e-prints*, September 2019.

[61] Suhas Jayaram Subramanya, Harsha Vardhan Simhadri, Srajan Garg, Anil Kag, and Venkatesh Balasubramanian. BLAS-on-flash: An efficient alternative for large scale ML training and inference? In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, pages 469–484. USENIX Association, 2019.

[62] Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. BERT4Rec: Sequential Recommendation with Bidirectional Encoder Representations from Transformer. *arXiv e-prints*, page arXiv:1904.06690, April 2019.

[63] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: A framework for enabling realistic studies of modern multi-queue SSD devices. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, pages 49–66. USENIX Association, 2018.

[64] NVIDIA tesla V100 architeture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[65] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2Tensor for Neural Machine Translation. *arXiv e-prints*, page arXiv:1803.07416, March 2018.

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv e-prints*, June 2017.

[67] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. SuperNeurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 41–53. ACM, 2018.

[68] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. XLNet: Generalized autoregressive pretraining for language understanding. *arXiv e-prints*, June 2019.

[69] Haoyu Zhang, Jianjun Xu, and Ji Wang. Pretraining-Based Natural Language Generation for Text Summarization. *arXiv e-prints*, page arXiv:1902.09243, February 2019.

[70] Jie Zhang and Myoungsoo Jung. ZnG: Architecting GPU multi-processors with new flash for scalable data analysis. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, pages 1064–1075, 2020.

[71] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable parallel flash firmware for many-core architectures. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, pages 121–136. USENIX Association, February 2020.

[72] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. PEGASUS: Pre-training with extracted gap-sentences for abstractive summarization. *arXiv e-prints*, December 2019.

FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks

Jonghyun Bae[†] Jongsung Lee[†‡] Yunho Jin[†] Sam Son[†] Shine Kim[†‡] Hakbeom Jang[‡]
Tae Jun Ham[†] Jae W. Lee[†]
[†]*Seoul National University* [‡]*Samsung Electronics*

Abstract

Deep neural networks (DNNs) are widely used in various AI application domains such as computer vision, natural language processing, autonomous driving, and bioinformatics. As DNNs continue to get wider and deeper to improve accuracy, the limited DRAM capacity of a training platform like GPU often becomes the limiting factor on the size of DNNs and batch size—called *memory capacity wall*. Since increasing the batch size is a popular technique to improve hardware utilization, this can yield a suboptimal training throughput. Recent proposals address this problem by offloading some of the intermediate data (e.g., feature maps) to the host memory. However, they fail to provide robust performance as the training process on a GPU contends with applications running on a CPU for memory bandwidth and capacity. Thus, we propose FlashNeuron, the *first* DNN training system using an NVMe SSD as a backing store. To fully utilize the limited SSD write bandwidth, FlashNeuron introduces an offloading scheduler, which selectively offloads a set of intermediate data to the SSD in a compressed format without increasing DNN evaluation time. FlashNeuron causes minimal interference to CPU processes as the GPU and the SSD directly communicate for data transfers. Our evaluation of FlashNeuron with four state-of-the-art DNNs shows that FlashNeuron can increase the batch size by a factor of 12.4× to 14.0× over the maximum allowable batch size on NVIDIA Tesla V100 GPU with 16GB DRAM. By employing a larger batch size, FlashNeuron also improves the training throughput by up to 37.8% (with an average of 30.3%) over the baseline using GPU memory only, while minimally disturbing applications running on CPU.

1 Introduction

Deep neural networks (DNNs) are the key enabler of emerging AI-based applications and services such as computer vision [19, 22, 38, 53, 54], natural language processing [2, 11, 13, 51, 67], and bioinformatics [46, 73]. With a relentless pursuit of higher accuracy, DNNs have become wider and deeper to increase network size [65]. It is because even a 1% accuracy loss (or gain) potentially affects the experience of millions of users if the AI application serves a billion of people [47].

DNNs must be *trained* before deployment to find optimal network parameters that minimize the error rate. Stochastic Gradient Descent (SGD) is the dominant algorithm used for DNN training [15]. In SGD, the entire dataset is divided into multiple (mini-)batches, and weight gradients are calculated and applied to the network parameters (weights) for each batch via backward propagation. Unlike inference, the training algorithm reuses the intermediate results (e.g., feature maps) produced by a forward propagation during the backward propagation, thus requiring a lot of memory space [55].

This GPU *memory capacity wall* [33] often becomes the limiting factor on DNN size and its throughput. Specifically, such a large memory capacity requirement forces a GPU device to operate at a relatively small batch size, which often adversely affects its throughput. The use of multiple GPUs can partially bypass the memory capacity wall because a careful use of multiple GPUs can achieve near-linear improvements in throughput [27, 28, 59]. However, such a throughput improvement comes with the linear increase in the GPU cost, which is often a major component of the overall system cost. As a result, the use of multiple GPUs often ends up with sub-optimal cost efficiency (i.e., throughput/system cost) as it does not change the fact that each GPU is not operating at its full capacity due to the limited per-GPU batch size.

This memory capacity problem in DNN training has drawn much attention from the research community. The most popular approach is to utilize the host CPU memory as a backing store to offload some of the tensors that are not immediately used [8, 9, 24, 42, 55, 62]. However, this *buffering-on-memory* approach fails to provide robust performance as the training process on the GPU contends with applications running on the CPU for memory bandwidth and capacity (e.g., data augmentation tasks [5, 41, 57, 61] to boost training accuracy). Moreover, these proposals focus mostly on increasing batch size but less on improving training throughput. Therefore, they often yield a low training throughput as the cost of CPU-GPU data transfers outweighs a larger batch's benefits.

Thus, we propose FlashNeuron, the first DNN training system using a high-performance SSD as a backing store. While NVMe SSDs are a promising alternative to substitute or augment DRAM, they have at least an order of magnitude

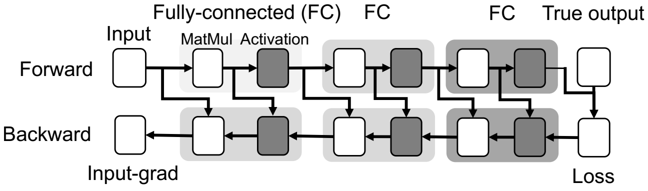


Figure 1: DNN training iteration and data reuse pattern.

lower bandwidth than both HBM DRAM on GPU and DDRx DRAM on CPU. Therefore, it is critical to effectively smooth bandwidth utilization to minimize bandwidth waste while overlapping GPU computation with GPU-SSD data transfers. To this end, FlashNeuron introduces an offloading scheduler, which judiciously selects a set of tensors to offload to the SSD. On the host side, FlashNeuron is realized by a lightweight user-level I/O stack, which leaves a minimal resource footprint on CPU cycles and memory usage as the GPU and the SSD directly communicate for tensor data transfers utilizing GPUDirect [17] technology.

We prototype FlashNeuron on PyTorch [50], a popular DNN framework, and evaluate it using four state-of-the-art DNN models. Our evaluation with the state-of-the-art NVIDIA V100 GPU with 16GB DRAM shows that FlashNeuron can scale the batch size by a factor of 12.4× to 14.0× over the maximum allowable batch size using the GPU memory only. By selecting the optimal batch size, FlashNeuron improves the training throughput by 30.3% on average over the baseline with no offloading, with a maximum gain of 37.8%. At the same time, FlashNeuron also provides excellent isolation between CPU and GPU processes. Even under an extreme condition of CPU applications utilizing 90% host memory bandwidth, the slowdown of the DNN training on GPU falls within 8% of standalone execution, while the slowdown of buffering-on-memory can be as high as 67.8% (i.e., less than one-third of the original throughput).

Our contributions can be summarized as follows:

- We identify a bandwidth contention problem in recent buffering-on-memory proposals [8, 9, 24, 42, 55, 62] and propose FlashNeuron, the first *buffering-on-SSD* solution to overcome this problem.

- We introduce a novel offloading scheduler to fully utilize the scarce SSD write bandwidth.

- We implement a lightweight user-space I/O stack customized for DNN training, which orchestrates SSD-GPU direct data transfers with minimal CPU intervention.

- We prototype FlashNeuron on PyTorch, a popular DNN framework, and evaluate it using four state-of-the-art DNNs to demonstrate its effectiveness for increasing batch size and hence training throughput, while minimally disturbing applications on CPU.

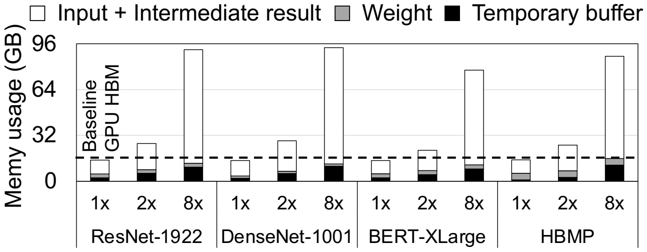


Figure 2: Breakdown of GPU memory usage in DNN training.

2 Background and Motivation

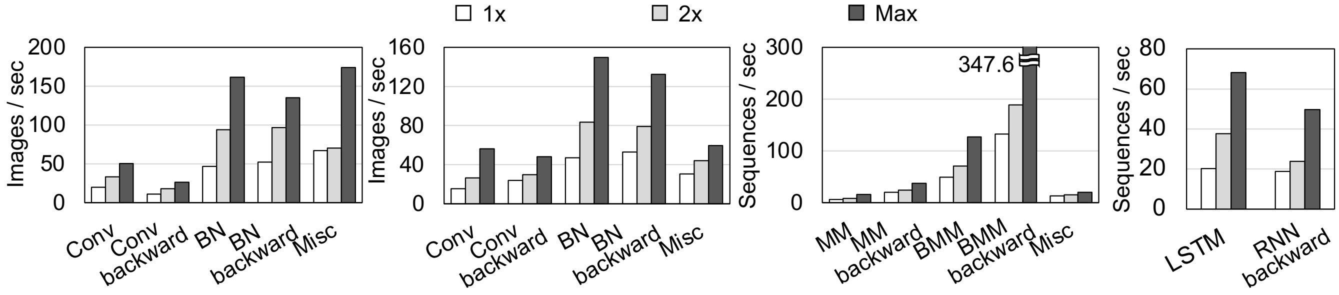
2.1 DNN Training

Deep Neural Networks (DNN) are widely used for many machine learning tasks such as computer vision [19, 22, 38, 53, 54], natural language processing [2, 11, 13, 51, 67], bioinformatics [46, 73] and so on. For a DNN to effectively perform a target task, it has to learn optimal network parameters using a large amount of labeled data — a process called *training*. DNN training is often performed using mini-batch stochastic gradient descent (SGD) algorithm [15]. In this algorithm, a training process is divided into multiple *epochs*, where a single epoch processes the entire dataset exactly once. Then, a single epoch is further divided into multiple *iterations*, where each iteration processes a single partition of the dataset, called *(mini-)batch*, to update network parameters.

As shown in Figure 1, an iteration consists of two steps: forward pass — a process of computing error for the given input batch, and backward pass — a process of back-propagating errors and updating network weights. A forward pass starts from the very first layer. Given input data, it simply performs the computation associated with the first layer using the layer's current weight and then passes the outcome to the next layer. This process is repeated until the last layer is reached. At that point, the error (also called loss) of the model is computed by comparing the last layer's outcome with the correct output. Then, the backward propagation starts from the last layer. During this step, i) the gradient of a layer's inputs to the final error is computed (using the gradient of the next layers' inputs to the final error) and passed to the next layer, and ii) the gradient of the layer's weights to the final error is computed using i) and stored. Once the backward propagation finishes in the first layer, the weights of all layers are updated accordingly based on the weight gradient computed during the backward pass.

2.2 Memory Capacity Wall in DNN Training

This training process exhibits an interesting data reuse pattern. Specifically, during a forward pass, inputs and intermediate computation results (e.g., products of weights and inputs in the feed-forward layer, followed by activation) of each layer should be buffered. Then, during the backward pass, i) the buffered intermediate computation results of a



(a) ResNet-1922 (b) DenseNet-1001 (c) BERT-XLarge (d) HBMP

Figure 3: Per-layer throughput of key layers in various DNN models. (`Conv`: Convolution, `BN`: BatchNorm, `MM`: Matrix Multiplication, `BMM`: Batched Matrix Multiplication). *1*× represents the maximum batch size of the baseline using GPU memory only. *Max* represents the batch size that saturates the training throughput with an idealized assumption of zero offloading overhead.

layer are used to compute the layer's gradient, and ii) the buffered inputs are used to compute the gradient of the layer's weights. Arrows in Figure 1 illustrates this data reuse pattern. This data buffering does not cause a problem when the network is shallow. However, a recent trend in deep learning is to utilize networks with a large number of layers (i.e., *deep* networks). With this trend, the amount of memory capacity required to buffer data (i.e., inputs and intermediate computation results for each layer) becomes much larger. It is not feasible in these deep networks to train the network using a large batch size as the required memory capacity for data buffering exceeds the available GPU memory size.

Figure 2 shows the required memory capacity of the DNN models across different batch sizes. Specifically, for each model, the figure shows the minimum memory capacity required to perform training for a certain batch size successfully. Here, 1× represents the maximum batch size that this model can run on a state-of-the-art GPU (i.e., Tesla V100) with 16GB memory. 2× and 8× represent the 2× and 8× batch sizes of the base (1×) batch size. To run on these large batch sizes, we offload all the tensors to host CPU memory except for those of the layer currently being executed. On the base batch size (1×), the required capacity is just below 16GB, indicating that this model almost fully utilizes the provided GPU memory. However, this model cannot be run on a GPU with 16GB memory when we set the batch size to be 2× or 8× as the required memory size far exceeds the available memory size. This figure also shows that most of the memory capacity is occupied by the inputs and the intermediate computation results for each layer. Other memory objects such as weights or temporary buffers (e.g., temporary workspace for convolution operations) take a relatively small portion of these models' total memory consumption.

GPU memory capacity bottleneck described above significantly limits the per-GPU batch size of the DNN models. Using a small batch size often results in the GPU's lower utilization, which leads to lower throughput [3, 16, 68, 69]. Figure 3 presents the per-layer throughput of key layers (i.e., layers accounting for a significant fraction of the total time) in various DNN models. We run each layer in isolation for this exploratory experiment without considering the overheads of tensor offloading, host-GPU communication, etc. The figure shows that there is still significant room for additional throughput by increasing the batch size. GPU resources are being underutilized even at the maximum per-GPU batch size if only GPU memory is used. In this scenario, a GPU throughput can be improved by ameliorating the GPU memory capacity bottleneck. One potential concern is that larger batch size can sometimes negatively affect the model accuracy [20,30,40]. However, for extremely deep neural network models, the base batch size is relatively small, and thus an increase in batch size is expected not to affect the final model accuracy severely.

2.3 Overcoming GPU Memory Capacity Wall

A popular approach to overcome GPU memory capacity bottleneck is to buffer data in the host CPU memory. For example, both vDNN [55] and SuperNeurons [62] (selectively) offload activation tensors to the CPU memory. These buffering-on-memory solutions can interfere with the CPU processes for memory bandwidth and capacity to pay a significant opportunity cost. For example, running *data augmentation* on CPU at every iteration of DNN training is a common practice [32, 41, 66] to prevent overfitting to the training data set. A typical data augmentation pipeline consists of image loading, decoding, and a sequence of geometric transformations [5, 41, 57, 61], requiring high memory bandwidth.

Figure 4 shows the GPU training throughput of a buffering-on-memory system while the CPU is continuously running a multi-threaded data augmentation task composed of rotation, transposition, and color conversion. By adjusting the number of data augmentation threads, we control the amount of memory bandwidth consumed by the CPU task (i.e., *50%*: 21GB/s, *70%*: 29GB/s, *90%*: 36GB/s). The throughput of DNN training with buffering-on-memory is noticeably degraded due to memory bandwidth contention. To avoid this problem, we

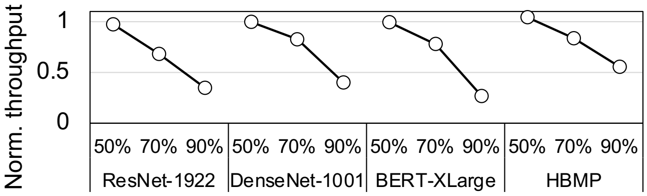


Figure 4: Normalized throughput of buffering-on-memory system (vDNN [55]-like) when the host CPU is running a data augmentation with varying degrees of contention.

propose a new solution that buffers inputs and intermediate data (tensors) to SSDs. Specifically, we leverage direct peer-to-peer communication between the GPU and NVMe SSD devices so that data buffering does not consume either host CPU cycles or memory bandwidth. This *buffering-on-SSD* approach complements the popular buffering-on-memory approach, hence improving overall resource utilization over various use cases.

3 FlashNeuron Design

3.1 Overview

FlashNeuron is a library that can be integrated into popular DNN execution frameworks. Figure 5 shows the system overview of FlashNeuron. FlashNeuron consists of three components: offloading scheduler, memory manager, and peer-to-peer direct storage access. Specifically, the offloading scheduler identifies a set of tensors (i.e., multidimensional matrices) to offload and generates an offloading schedule by considering multiple factors such as tensor sizes, tensor transfer times, and forward/backward pass runtime. Once the schedule is determined, the memory manager orchestrates data transfers between the GPU memory and the SSDs using peer-to-peer direct storage access to minimize performance overheads from offloading.

3.2 Memory Manager

Tensor Allocation/Deallocation. Instead of buffering all input and intermediate data in memory, FlashNeuron chooses to buffer selected tensors in SSDs, which requires extra tensor allocations and deallocations. Since frequent GPU memory allocations and deallocations using runtime (e.g., CUDA) incur noticeable performance overheads, FlashNeuron employs a custom memory allocator. The custom memory allocator first reserves the whole GPU memory space initially and manages memory allocation/deallocation itself. In FlashNeuron, tensors are allocated when i) a tensor is first created during the forward propagation or ii) an offloaded tensor is prefetched from the SSD to the memory during a backward pass. On the other hand, tensors are deallocated when i) a tensor is completely offloaded from the memory to the SSD, or ii) a tensor is no longer used by any layer during the iteration. To track the lifetime of a tensor, a reference counting mechanism (used in PyTorch [50] and TensorFlow [1]) is utilized. For DNN frameworks employing a static computational graph like Caffe, the memory manager traverses the computational graph and tracks the tensor lifetime through pointer chasing of tensors attached to each layer.

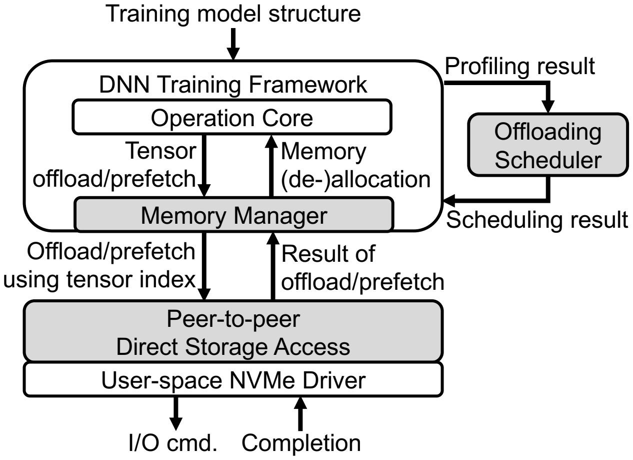


Figure 5: System overview of FlashNeuron.

One crucial issue in the tensor allocation and deallocation is fragmentation. If we allocate memory addresses for all tensors from the beginning, severe memory fragmentation can occur since only some tensors are offloaded to the SSDs, effectively making holes in the GPU physical memory address space. To avoid this issue, we allocate memory-resident (i.e., not offloaded) tensors from the lowest end of the memory address space and allocate ephemeral (i.e., offloaded) tensors from the highest end of the memory address space. Since the ephemeral data has a very short lifetime during the forward pass, only a tiny portion of the memory address space is utilized for such data, and thus the amount of fragmented memory space becomes negligible.

Managing Offloading and Prefetching. The memory manager interacts with peer-to-peer direct storage access (P2P-DSA) to perform offloading and prefetching. It initiates an offloading request of a tensor to P2P-DSA during the forward pass immediately after its use by the next layer. At the end of each layer's execution, the memory manager checks whether the offloading request is completed (i.e., the tensor is wholly offloaded to the SSD). Then, the tensor is deallocated from the GPU memory at this point.

The memory manager issues prefetch requests to the SSD during the backward pass. At the beginning of the backward pass, it first allocates memory and initiates prefetch requests for the set of tensors that are soon to be used. Then, whenever those tensors are used and freed, the memory manager eagerly prefetches additional tensors using the available memory space while reserving enough memory for execution to run the largest layer.

Augmented Compressed-Sparse Row (CSR) Compression and Decompression. When offloading a tensor, the memory manager applies CSR compression if the compression ratio estimated during the profiling iteration is greater than one. The CSR compression is only applied to output tensors of ReLU. We observe that ReLU outputs have a high sparsity, ranging from 43% up to 75% during the training process. Since a tensor is a multi-dimensional matrix, we cast the tensor into a two-dimensional matrix whose column has 128 entries. Then, we apply a slightly different CSR format where we replace a vector storing the column index of each element (often called JA vector) to a set of bit-vectors where each bit vector represents a set of nonzero elements for a row. By doing so, the size of CSR format representation decreases by *8 bits (to represent the column index)* × *the number of nonzero elements in the matrix* and increases by *1 bit* × *the total number of elements in the matrix*. This representation is beneficial when more than one-eighth of all the elements are nonzero. Since this is the typical case for input and intermediate tensors, we apply this technique to improve the compression ratio. We implement a specialized routine to perform this augmented-CSR compression/decompression in GPU. According to our evaluation, the runtime overhead of these compression/decompression operations is negligible.

Use of a Half-precision Floating Points (FP16) for Offloaded Tensors. To further reduce the traffic between the GPU and the SSD, the memory manager exploits the fact that neural network can tolerate a certain level of precision loss without significantly degrading the final model accuracy. Specifically, during a forward path, the memory manager first converts the offloaded tensor to FP16 format (from FP32) and then stashes them in the SSD. Later, during a backward propagation, the offloaded FP16 tensor is prefetched, padded to FP32, and reused. Naturally, the use of a lower-precision tensor comes with the potential degradation in the final model accuracy [44, 63]. However, a previous study demonstrates that the technique of selectively utilizing FP16 for the offloaded tensors incurs less accuracy degradation than processing all tensors in FP16 [25]. The key observation is that the FP32 tensor is utilized during forward propagation, and the stashed FP16 version of the same tensor is only utilized during a backward propagation (Delayed Precision Reduction in [25]).

3.3 Offloading Scheduler

The offloading scheduler in FlashNeuron takes a DNN model as input and derives an optimal tensor offloading schedule, which is designed with the following rationale. First, it should offload enough tensors so that the GPU can correctly run at the target batch size without triggering an out-of-memory error. Second, it should avoid excessive data transfers from the GPU to the SSD (and vice versa), thus minimizing the increase of the iteration time induced by tensor offloading.

The offloading scheduler finds an optimal schedule for a given target batch size. It first performs a profiling iteration. At this iteration, all the tensors that are buffered during the forward pass (i.e., input and intermediate data for each layer) are offloaded to the SSD so that the system can run with a large target batch size without causing an out-of-memory error. Profiling iteration collects i) the size of each buffered tensor, ii) the time it takes to offload each buffered tensor, iii) the expected compression ratio of a tensor using CSR (Compressed Sparse Row) format and half-precision floating-point conversion, iv) the execution time for the forward pass and the backward pass (excluding tensor offloading time), and v) the total size of the other memory-resident objects (e.g., weights, temporary workspace). Once this profiling iteration completes, information collected during this iteration is passed to the offloading scheduler.

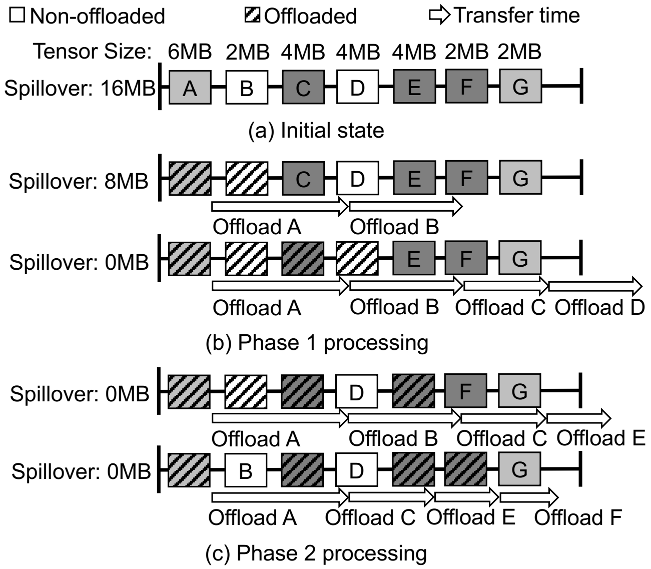


Figure 6: Tensor selection walk-through. Darker boxes indicate tensors with a higher compression ratio, whereas lighter boxes those with lower compression ratio.

Phase 1: Linear Tensor Selection. The first phase of the scheduler is to iteratively select a certain number of tensors from the beginning until the total size of the unselected tensors plus the total size of the other memory-resident objects (i.e., weights and temporary workspace) becomes smaller than the total GPU memory size. Figure 6 illustrates this process, where the forward-pass with seven buffered tensors (labeled A through G) is to run on a hypothetical GPU with 8MB physical memory. Figure 6(b) shows the example selection process of Phase 1. At this point, the scheduler checks whether the total data transfer time, which is computed by summing up the individual tensor offloading times, is smaller than the total execution time of all layers in the forward pass. If this condition is satisfied, the scheduler adopts this schedule and stops because it can fully overlap tensor offloading with layer computation via scheduling. If not, the offloading scheduler enters the second phase.

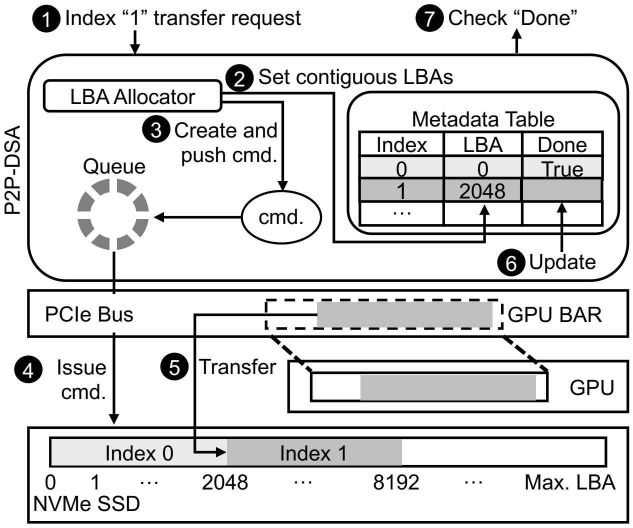
Phase 2: Compression-aware Tensor Selection. The second phase of the scheduler is run only when a satisfactory schedule is not found in the first phase. This indicates that the current schedule spends too much time offloading the tensors, and such the transfer time has now become the new bottleneck. To solve the issue, our scheduler replaces the already selected tensors with compression-friendly tensors expected to have high compression ratios with CSR and FP16 conversion illustrated in Figure 6(c). Specifically, the scheduler performs the following steps in an iterative way to refine the existing schedule. First, the scheduler excludes the last uncompressible tensor selected from Phase 1. It is replaced with one or more tensors having the highest expected compression ratios among the tensors that are not yet selected, such that the size of the newly selected tensors exceeds the excluded tensor size. Then, it recomputes the expected total data transfer time, assuming that the compressed tensor takes a fraction (inversely proportional to the compression ratio) of the original offloading time. If this total transfer time does not exceed the forward pass's total execution time, the scheduler stops. Otherwise, it repeats this process until the condition is satisfied or there exist no compression-friendly tensors (i.e., tensors whose size does not decrease after compression).

If a *satisfactory* schedule is found, the corresponding batch size is likely not to increase the iteration time and achieve higher throughput than the baseline. On the other hand, if our scheduler stops as it cannot find more compression-friendly tensors, the generated schedule is expected to incur some delay from tensor transfers. However, this schedule can still be used to run DNN training at a larger batch size (but likely at a lower throughput).

3.4 Peer-to-Peer Direct Storage Access

Peer-to-peer direct storage access (P2P-DSA) enables direct memory access between a GPU and NVMe SSDs without using the host DRAM buffer to minimize host intervention during SSD read/write. P2P-DSA builds on GDRCopy [14] and SPDK [58] to communicate tensors from/to a GPU to/from NVMe SSDs. GDRCopy is a fast GPU memory copy library based on NVIDIA GPUDirect [17], a technology that exposes the GPU memory to be accessed directly by other PCIe peripherals. Intel SPDK exposes a block-level I/O interface directly to the user-space software. P2P-DSA is a lightweight layer that leverages these two technologies to enable direct offloading/prefetching tensors from GPUs to SSDs. To maintain each tensor's metadata offloaded to SSDs, P2P-DSA contains a metadata table consisting of a `long` LBA value and a `boolean` value to check the I/O completion.

Example Walk-through of a Transfer Request. Figure 7 illustrates the operations of the transfer request (offloading from the GPU to the SSD) in greater details. When `P2PDSA_issue` is called, ❶ P2P-DSA get the `index`, `buffer`, and `direction(write)` information from the transfer request. Then, ❷ the logical block address (LBA) allocator is called to allocate a set of contiguous blocks on a single SSD device or multiple SSD devices (when multiple SSDs are utilized to boost offloading/prefetching bandwidth). The LBA of the first block allocated from the LBA allocator is updated at the metadata table's appropriate location. After this point, ❸ P2P-DSA creates a command for each logical block and then enqueues it to the command queue. Here, an NVMe command contains i) the source address (GPU memory address is translated to PCIe bus address by GPUDirect), and ii) the device address (computed using the LBA in metadata table).



Index	LBA	Done
0	0	True
1	2048	
···		

Figure 7: Overall structure of P2P-DSA with an example walk-through (write).

When `P2PDSA_update` is called, ❹ commands queued in the software command queue are fetched and issued to NVMe SSD as long as the NVMe device submission queue has space. Then, ❺ NVMe SSD devices will execute these requests and perform direct data transfers between SSD devices and the GPU. Sometime later, these transfer requests will be completed, and their status will be updated in the NVMe device completion queue. When the `P2PDSA_update` is called once again, ❻ `P2PDSA_update` will clear the completion queues and updates the metadata table by setting corresponding done bits. At this point, ❼ if the application calls `P2PDSA_is_done` for the already offloaded tensor, it will return true. The reverse-path (prefetching data from the SSD to the GPU memory) is performed similarly except that i) LBAs are read from the metadata table instead of being allocated, and ii) read commands are issued instead of write commands. In both offloading and prefetching cases, most data accesses are sequential accesses, which are more advantageous than random accesses in throughput and endurance.

Implications on SSD lifetime. The endurance of an SSD depends on the program and erase (P/E) cycles for the NAND blocks. Therefore, for write-intensive workloads, a primary concern for the flash-based SSDs is the endurance degradation by wear-out of NAND blocks. We estimate the guaranteed (minimum) endurance of the SSD when running the P2P-DSA workload, using drive writes per day (DWPD) (i.e., 3 DWPD

Table 1: System configurations.

CPU	Intel Xeon Gold 6244 CPU 8 cores @ 3.60GHz
GPU	NVIDIA Tesla V100 16GB PCIe
Memory	Samsung DDR4-2666 64GB (32GB × 2)
Storage	Samsung PM1725b 8TB PCIe Gen3 8-lane × 2 (Seq. write: 3.3GB/s, Seq. read: 6.3GB/s)
OS	Ubuntu server 18.04.3 LTS
Python	Version 3.7.3
PyTorch	Version 1.2

for five years of Samsung PM1725b SSD [49], assuming a 50% write-50% read workload like P2P-DSA). If the training workload is running 24×7, the endurance is estimated to be about 7,374 hours, which is 307 days (i.e., 3 DWPD × 5 years × 365 days × 8,000 GB × 2 (50% write) / 3.3 GB/s / 86,400 seconds/day). While a longer lifetime would be desirable, we note that our estimation is conservative as P2P-DSA only performs sequential writes to sustain the write amplification factor (WAF) of (nearly) one to maximize endurance. SSD manufacturers typically use 4KB random write [49] to report the endurance number, which has a higher WAF than sequential writes at least by 3.5× [6, 21]. Furthermore, if P2P-DSA uses the emerging low-latency SSDs, such as Intel Optane SSD [45] and Samsung Z-SSD [70], the endurance can be further improved by a factor of 5× to 10×. Finally, we can further extend the SSD lifetime by leveraging tradeoffs between cell retention time and P/E cycles [7, 26]. An offloaded object has a very short lifetime and hence requires a much lower retention time (i.e., one training iteration time, which is in order of seconds and minutes at most, rather than years as required by modern SSDs). This can improve the P/E cycles by 46× or more [7]. With these optimizations, the expected SSD lifespan can increase by multiple orders of magnitude.

4 Evaluation

4.1 Methodology

System Configurations. We evaluate FlashNeuron on a Gigabyte R281-3C2 rack server with NVIDIA Tesla V100 and two Samsung NVMe PM1725b SSDs. The details of hardware and software configurations are summarized in Table 1.

Workloads. Among various DNN models, we choose four state-of-the-art models and scale them up to represent the future DNN models with very deep structures: ResNet-1922 [19] and DenseNet-1001 [22, 74] are state-of-the-art deep CNN models for image processing. BERT-XLarge [13] and HBMP [60] are two of the top-performing models for natural language processing tasks. ResNet-1922 and DenseNet-1001 [74] are selected based on the deepest network of ResNet and DenseNet models. The depths of BERT-XLarge and HBMP are increased by 2× and 4× from the maximum size stated in the original papers. Note that these naively scaled models do not necessarily improve accuracy. Our purpose is to use them as proxies for future DNN models requiring a much larger capacity for efficient training. The specifics are summarized in Table 2.

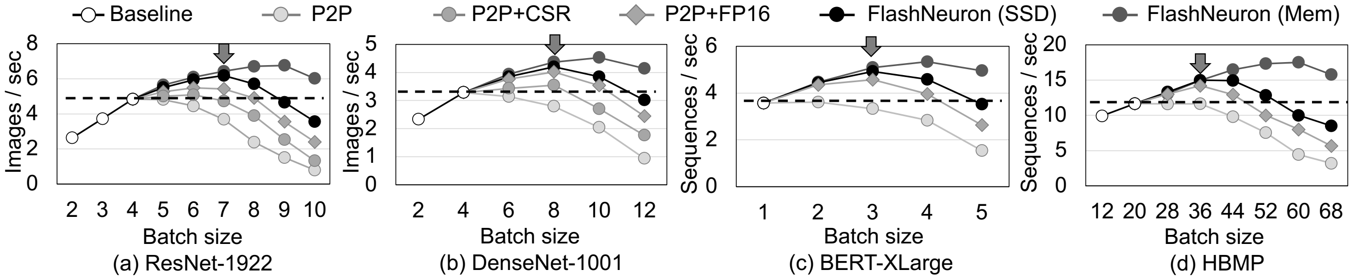
Table 2: Suite state-of-the-art DNN models and datasets used, major layer types and counts.

Network	Dataset	# of layers	Structure
ResNet-1922 [19]	ImageNet [12]	1922	(Conv1→BN1→ReLU→Conv2→BN2→ReLU)^n
DenseNet-1001 [22, 74]	ImageNet [12]	1001	(Conv1→BN1→ReLU)$^{n-1}$→Conv2→BN2→ReLU
BERT-XLarge [13]	SQuAD 1.1 [52]	48 blocks	(Embd1→Embd2→Embd3→FC1→Attn→FC2→LNorm)^n
HBMP [60]	SciTail [31]	24 hidden layers	FC^m→LSTM^n

4.2 Performance Evaluation

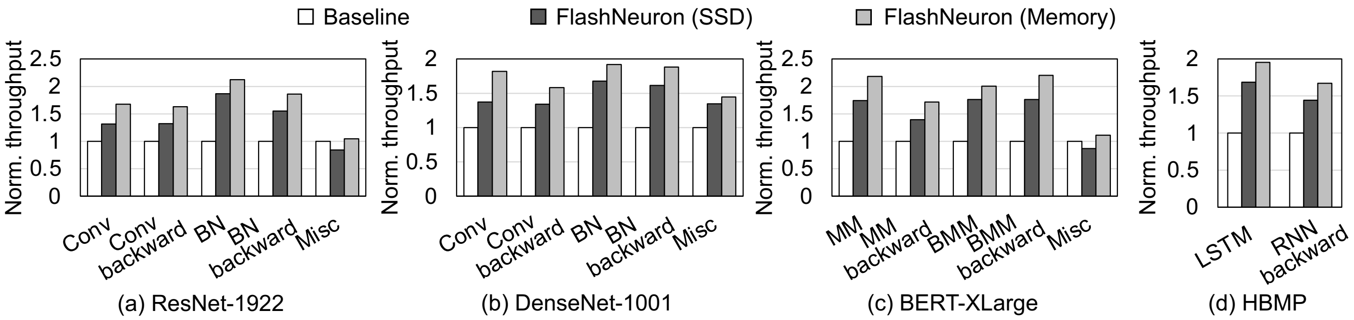
Overview. Figure 8 shows the training throughput over varying batch sizes. The baseline uses GPU memory only. FlashNeuron (SSD) and FlashNeuron (Memory) offload tensors to SSD and CPU memory, respectively, with no interference from CPU processes. Note that FlashNeuron (Memory) represents a state-of-the-art buffering-on-memory scheme. The dotted line shows the best throughput that can be achieved by the baseline. To demonstrate the effectiveness of FlashNeuron, we mark with an arrow the maximum batch size for which the proposed offloading scheduler is able to find an effective schedule (i.e., a schedule that does not increase the estimated forward-pass time). The training throughput indeed peaks at the batch size marked with the arrow. As we further increase the batch size, the throughput gets degraded as the cost of tensor offloading outweighs the benefits of the increased batch size.

FlashNeuron (SSD) improves the training throughput by up to 37.8% by selecting the optimal batch size and can increase the batch size by up to 5.0× while achieving at least the same throughput as the baseline (or higher). In some cases, increasing batch size may give additional benefits to reduce total training time further. For example, the effectiveness of batch normalization is known to diminish for small batches, and increasing the batch size can yield higher accuracy, faster convergence, or both [36]. However, when the batch size is too large, the limited bandwidth between the GPU and the SSD becomes the bottleneck and offsets the higher utilization benefits. Some configurations in Figure 8 (e.g., batch size of 8+ in ResNet-1922 and 10+ in DenseNet-1001) represent these cases. The performance gap between FlashNeuron (SSD) and FlashNeuron (Memory) is attributed to the difference in sustainable write throughput. While FlashNeuron (Memory) can utilize the nearly full PCIe write bandwidth (13.0GB/s), FlashNeuron (SSD) is limited by the write throughput of the SSD device (3.0GB/s×2). Thus, FlashNeuron (Memory)



(a) ResNet-1922 (b) DenseNet-1001 (c) BERT-XLarge (d) HBMP

Figure 8: Throughput of FlashNeuron with varying batch sizes (`P2P`: Baseline with P2P, `P2P+CSR`: With P2P and CSR compression, `P2P+FP16`: With P2P and FP16 conversion). The arrow shows the maximum throughput of FlashNeuron (SSD).



(a) ResNet-1922 (b) DenseNet-1001 (c) BERT-XLarge (d) HBMP

Figure 9: Normalized per-layer throughput of key layers across training scenarios (`Conv`: Convolution, `BN`: BatchNorm, `MM`: Matrix Multiplication, `BMM`: Batched Matrix Multiplication).

achieves up to 49.1% throughput gain (with an average of 43.9%) over the baseline. This performance gap can be closed by FlashNeuron (SSD) employing additional SSDs to saturate the PCIe channel bandwidth.

Source of Performance Improvement. Overall, FlashNeuron benefits from its two optimizations: CSR and offloading tensors using FP16 representation. Figure 8 shows the improvements from each optimization. Here, `P2P` represents the configuration where tensors are offloaded using P2P-DSA, but without any other optimization (e.g., CSR compression or FP16). This configuration enables the use of a larger batch size beyond the GPU memory capacity limit. However, the limited bandwidth between the GPU and the SSD limits the performance. P2P with CSR compression (`P2P+CSR`) improves the baseline performance by 7.14%. Note that the tensor compression does not improve the performances of BERT-XLarge and HBMP because those models do not utilize a ReLU layer. P2P with FP16 conversion (`P2P+FP16`) improves the performance by 21.41% over the baseline. The improvement is greater than that of P2P+CSR because the use of the FP16 format cuts the traffic by half, while the CSR compression is only applied for a limited set of tensors (e.g., output tensors of ReLU).

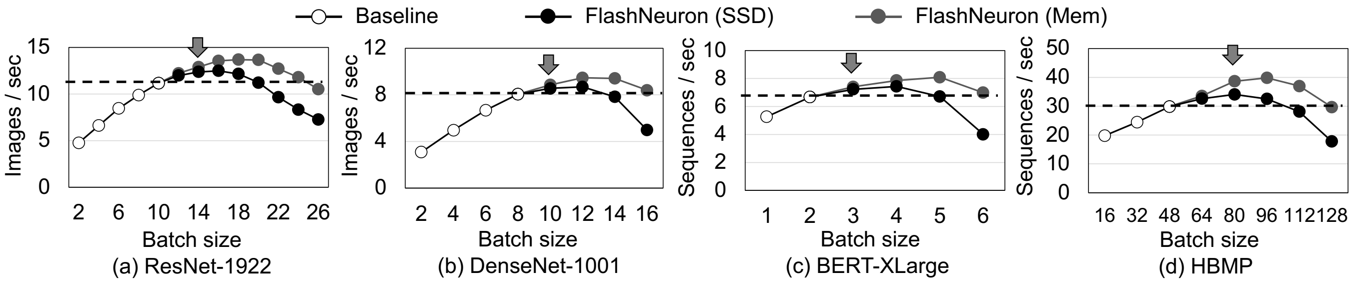
Figure 9 shows the per-layer throughput of FlashNeuron at the optimal batch size normalized to the baseline using GPU memory only. By employing a larger batch size, FlashNeuron substantially increases the throughput for the key layers. Batch normalization (BN) and LSTM layers benefit the most from the increase in batch size, whereas convolution (Conv) layers demonstrate relatively modest improvements. It is because the Conv layer is known to be compute-intensive and already has a high resource utilization even for the baseline.

Table 3: Batch sizes achieving maximum throughput and the maximum batch size FlashNeuron can run.

Network	Baseline	Maximum throughput		Runnable maximum	
	Batch	Batch	Ratio	Batch	Ratio
ResNet-1922	4	7	1.75×	56	14.0×
DenseNet-1001	4	8	2.00×	52	13.0×
BERT-XLarge	1	3	3.00×	14	14.0×
HBMP	20	36	1.80×	248	12.4×

Since the bandwidth of the PCIe channel and SSD writes is the limiting factor for performance, future scaling of both PCIe and SSD write bandwidth will further improve the throughput. For example, PCIe 5.0 interconnects [48] will provide 4× higher bandwidth than PCIe 3.0 used in this work, enabling FlashNeuron (SSD) to utilize 4× larger batch size. Figure 3 demonstrates that there is still substantial room for further throughput improvement, and higher bandwidth interconnects in the future will close this performance gap.

Maximum Batch Size. Table 3 shows the largest batch size for different configurations. The first column ("*Baseline*") is the maximum batch size using GPU memory only (i.e., without using FlashNeuron). The second column ("*Maximum throughput*") shows the batch size for which FlashNeuron (SSD) yields the highest throughput, which is marked by



(a) ResNet-1922 (b) DenseNet-1001 (c) BERT-XLarge (d) HBMP

Figure 10: Throughput of FlashNeuron with half-precision. The arrow shows maximum throughput of FlashNeuron (SSD).

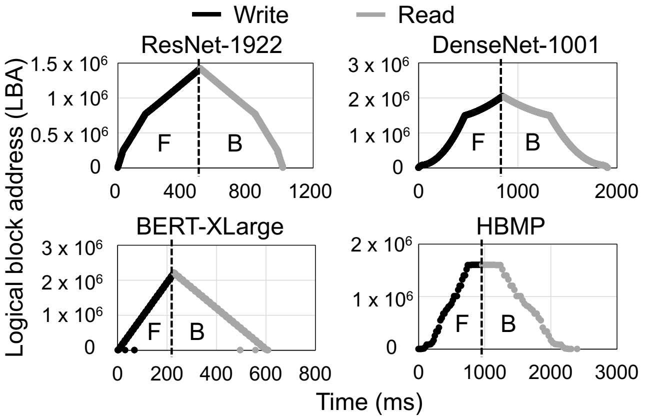


Figure 11: LBA access pattern during a single iteration (F: Forward propagation, B: Backward propagation).

an arrow in Figure 8. Finally, the third column ("*Runnable maximum*") shows the maximum batch size that FlashNeuron can run to completion. On average, FlashNeuron uses 2.09× larger batch size to maximize the training throughput and increases the maximum runnable batch size by a factor of 13.4×. When FlashNeuron is operating with the runnable maximum batch size, FlashNeuron offloads 59.2GB of tensors on average (up to 68.6GB for DenseNet-1001) and occupies 33.2GB (up to 48.9GB for HBMP) storage space.

Half-precision Training. Based on the observation that many neural network models can still maintain the competitive accuracy using FP16 representations, FP16 training is gaining popularity. For example, the high-end NVIDIA GPUs come with Tensor Cores, which are specialized functional units for FP16 computations. Unfortunately, when the tensor is already represented in FP16, FlashNeuron does not benefit from converting the offloaded tensors to FP16. However, our experiments still demonstrate that FlashNeuron can result in extra speedup as well as an increase in the per-GPU batch size. Figure 10 shows the throughput of FlashNeuron over varying batch sizes when FP16 values are used for both weights and activations. FlashNeuron (SSD) enables the use of a 1.8× larger batch size while preserving the training throughput compared to the baseline. By employing larger batch sizes, FlashNeuron (SSD) and FlashNeuron (Memory) achieve 8.04% and 22.98% throughput improvement over the baseline, respectively. This FP16 training requires less memory/storage capacity per batch and thus enables even larger batch sizes. The speedup from FlashNeuron is smaller in this scenario than full-precision as the overall iteration time becomes shorter, thus having a much narrower window for tensor offloading.

I/O Pattern. Ensuring the sequential read/write by P2P-DSA is important for both performance and endurance. Figure 11 shows the SSD's logical block address (LBA) access pattern during a single iteration. During a forward propagation, offloaded tensors are allocated sequentially in the LBA space (on the left side of the figure's dotted line). In a backward propagation, the most recently written tensors are read first, and the least recently written tensors are read last, as shown on the right side of the dotted line. Each offloaded/prefetched tensor's size ranges from 2MB to 310MB, and it is sufficiently large to saturate the SSD's read/write bandwidth thoroughly.

Cost Efficiency. As of September 2020, DDR4 DRAM on the host CPU costs about $3.6/GB on average and NAND flash SSD about $0.102/GB [29,43,56]. Assuming the same capacity, FlashNeuron (SSD) achieves 35.3× higher cost-efficiency. HBM2 DRAM has much higher $/GB than DDR4, and thus scaling its capacity will be much more costly.

4.3 Case Studies

Our premise is that the common practice of leaving CPU (mostly) idle while running DNN training on GPU is suboptimal. Thus, we envision co-locating CPU jobs with DNN training to improve resource utilization substantially. To not degrade DNN training throughput running at large batch size, it is crucial to provide performance isolation between the co-located CPU and GPU processes. FlashNeuron is a unified framework that flexibly supports both SSD and memory offloading to minimize resource contention for a wide range of co-located CPU workloads. Superior performance isolation of FlashNeuron can enable consolidation of CPU applications and DNN training jobs. The two case studies in this section are presented *not* to claim that they are common use cases today, *but* to demonstrate that even memory-intensive CPU workloads can be effectively co-located with DNN training using FlashNeuron (SSD). For I/O-intensive workloads, one can opt to use FlashNeuron (Memory) to avoid I/O contention.

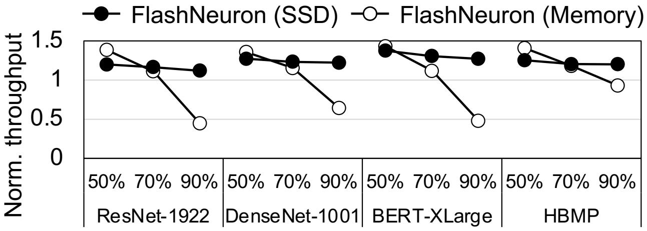


Figure 12: Normalized throughput of FlashNeuron (SSD) and FlashNeuron (Memory) when the host CPU is running a memory-intensive image transformation workload [23].

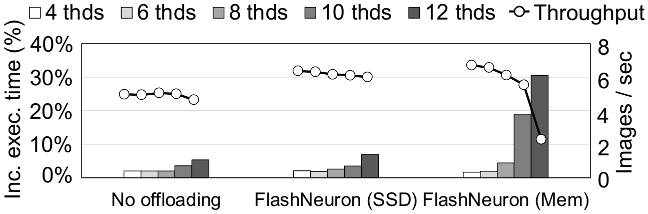


Figure 13: Increase in execution time of data augmentation tasks processing 256 2K (2048×1080) resolution images on CPU and training throughput of ResNet-1922 on GPU.

4.3.1 Co-locating Bandwidth-Intensive Tasks on CPU

The first use case is data augmentation tasks [5, 41, 57, 61] running on CPU while executing DNN training on GPU. Employing a data augmentation for DNN training is a common practice to prevent the model from overfitting to the data set, hence providing more robustness. Our example data augmentation transforms 2K (2048×1080) resolution images with a sequence of geometric operators such as rotation and transposition, as well as re-coloring operators such as color conversion. These operators are commonly used in data augmentation [10,34,37]. Note that the actual DNN model works with smaller images, but the data augmentation often works with the original image, and then the augmented image is resized to the model's input image size (e.g., 224×224).

Throughput of DNN Training on GPU. Buffering-on-memory can potentially achieve higher throughput than buffering-on-SSD for the higher write bandwidth of the CPU DRAM than the SSDs. However, the DNN training throughput with buffering-on-memory can be heavily affected by CPU processes' characteristics due to the memory bandwidth contention between CPU and GPU processes. Figure 12 shows the impact of CPU workload on the DNN training throughput on GPU for both FlashNeuron (SSD) and FlashNeuron (Memory). By controlling the number of data augmentation threads, we make the CPU process consume a certain portion of the CPU memory bandwidth. In particular, we use three configurations according to the portion of the CPU DRAM bandwidth consumed by the data augmentation task: *50%* (21GB/s), *70%* (29GB/s), *90%* (36GB/s).

When the CPU consumes 50% of the available memory bandwidth, the training throughput is still at least 35% higher than the baseline for both FlashNeuron (SSD) and FlashNeuron (Memory). However, when the CPU workload is more memory bandwidth-intensive (*75%*), FlashNeuron (Memory) yields only 14.0% throughput gains. This performance loss becomes even worse when the CPU workload utilizes nearly all of the available memory bandwidth (*90%*), where the training throughput degrades by 40.2% on average compared to baseline. On the other hand, FlashNeuron (SSD) still achieves 22.6% and 20.2% throughput gains over the baseline even if the CPU consumes 75% and 90% of the available memory bandwidth, respectively. Even in the worst case, the throughput loss of FlashNeuron (SSD) falls just within 8% of standalone execution, whereas that of FlashNeuron (Memory) can be as high as 67.8% (i.e., having lower than one-third of the original training throughput).

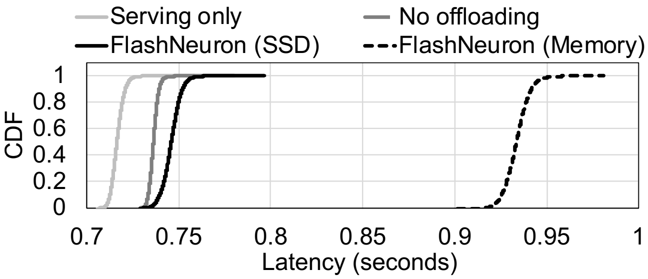


Figure 14: Query latency CDF of CPU inference across the various training scenario.

Execution Time of Data Augmentation Task on CPU. Figure 13 shows both the increase in execution time of the data augmentation task on CPU (bar graph) and DNN training throughput of ResNet-1922 using FlashNeuron on GPU (line graph). All bars are normalized to the baseline, which is standalone execution of the data augmentation pipeline with no co-located GPU processes. *No offloading* represents the case when GPU is running DNN training with no tensor offloading to either host memory or SSDs. FlashNeuron (SSD) only utilizes a minimal amount of the host memory bandwidth (mostly for PyTorch application code) to incur a comparable degree of the slowdown with *No offloading*. In contrast, FlashNeuron (Memory) consumes a large amount of the host CPU memory bandwidth (roughly equal to the maximum bandwidth of a 16-lane PCIe interface) to incur a substantial performance slowdown. The figures show that this memory bandwidth contention can break performance isolation between the CPU and GPU processes to make it much more challenging to deploy them in a consolidated environment.

4.3.2 Co-locating Latency-Critical Tasks on CPU

For the second case study, we select a *DNN inference* task, which is latency-critical; according to Facebook, inference tasks are mostly running on CPUs while requiring a large memory space for users and contents data [18]. We run a BERT-as-service [64] on CPU, which takes user-provided

Table 4: 50%, 95%, and 99% percentile of query latency and delay time ratio compared to *Serving only*.

	No offloading	**FlashNeuron (Memory)**		**FlashNeuron (SSD)**	
	Latency	**Latency**	**Delay**	**Latency**	**Delay**
50%	0.736s	0.933s	30.3%	0.746s	4.11%
95%	0.740s	0.944s	30.7%	0.754s	4.35%
99%	0.743s	0.950s	30.9%	0.758s	4.45%

sentences as input and invokes BERT to return their embedding, while concurrently running a BERT training on GPU.

Figure 14 shows the cumulative distribution function (CDF) of the CPU inference. *Serving only* is a case when there is no process running on GPU, whereas *No offloading* is when BERT is training but using GPU memory only. As shown in this figure and Table 4, FlashNeuron (SSD) incurs less than 5% and 2% slowdown compared to *Serving only* and *No offloading*. In contrast, over 30% latency increase is observed for FlashNeuron (Memory) compared to *Serving only* due to memory bandwidth contention. As for training throughput, FlashNeuron (SSD) experiences only a 1.8% slowdown, whereas FlashNeuron (Memory) as much as 27.5%. This slowdown is not sensitive to the model or dataset and is largely attributed to the bandwidth consumption to offload tensors.

5 Related Work

Augmented GPU Memory for DNN Training. Many proposals build on NVIDIA Unified Virtual Memory (UVM) [42], which enables transparent data sharing over both GPU and CPU memory. However, its performance is often limited due to its excessive page fault handling overhead [39]. To address this problem, several specialized schemes that do not rely on demand-fetching have been proposed to accelerate DNN training [8, 9, 24, 55, 62]. Similar to vDNN [55], moDNN [9] offloads and prefetches tensors in convolution layers in addition to accumulating gradients.

Alternatively, Chen et al. [8] propose to mark the outputs of convolution layers and free unmarked tensors. The freed data is recomputed during a backward pass. Merging the two ideas, SuperNeurons [62] offloads the marked tensors to host memory and saves device memory space. Ooc_cuDNN [24] divides the data in a single layer and performs for a piece of data at a time. The unused data is prefetched from the host memory concurrently with computation. Such mechanisms, however, experience substantial performance degradation when the host CPU is running memory-intensive workloads. To complement this, FlashNeuron offloads tensors directly to SSDs, and thus do not suffer performance degradation even under the presence of memory-intensive processes on the CPU.

Data Transfer Methods between GPU and Storage Devices. Several proposals introduce effective data transfer methods between GPU and storage devices [4,39,71]. Dragon [39] leverages the page-faulting mechanism of CPU and read-ahead operation of OS. Upon page fault, page cache in host memory is used as a bridge between GPU memory and NVM storage. SPIN [4] and NVMMU [71] take a step further, removing the usage of the host side buffer, thus allowing direct access from GPU to SSD. However, they are more general-purpose solutions, which perform sub-optimally for DNN training as they do not sequentially read/write.

Reducing Memory Footprint of DNN Models. Another way to relieve the capacity limitations of GPU memory is to optimize the DNN model without compromising the accuracy [25, 35, 72]. Echo [72] reduces the memory footprint by stashing small input values of the attention layers and recomputing the feature maps during the backward passes. Gist [25] applies various footprint reduction techniques by compressing feature maps, especially for ReLU-convolution and ReLU-pooling layers, as well as lower-precision representations (FP8/10/16). Likewise, FlashNeuron exploits sparse matrix representations such as CSR and FP16 representation on offloaded tensors to reduce the traffic to SSD devices.

6 Conclusion

With a relentless pursuit of higher accuracy, DNNs are continuously getting deeper and wider. One significant constraint in scaling trainable DNNs is the limited capacity of the GPU memory. This problem is exacerbated by emerging DNN applications required to handle large inputs. There have been previous attempts to overcome this GPU memory capacity wall through the use of host memory as a buffer for intermediate data generated during the forward pass of DNN training for reuse during the backward pass. However, these approaches experience substantial performance degradation as the host CPU contends for the limited host memory bandwidth. Thus, we propose FlashNeuron, the first buffering-on-SSD approach to offload intermediate data to high-performance NVMe SSDs instead of the host DRAM. FlashNeuron enables large-batch training of very deep and wide neural networks of today and the future to achieve high training throughput. Furthermore, it flexibly supports both SSD and memory offloading to provide excellent performance isolation between GPU training jobs and a wide range of co-located CPU workloads, including memory- and I/O-intensive ones.

Acknowledgments

We extend our thanks to Randal Burns for shepherding this paper. We also thank Jin-Soo Kim and Jaehoon Sim for valuable discussions and their help with P2P-DSA in an early phase of this work. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) (NRF-2020R1A2C3010663) and Samsung Electronics. The source code is available at https://github.com/SNU-ARC/flashneuron.git. Jae W. Lee is the corresponding author.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283. USENIX Association, 2016.

[2] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Vaino Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. In *Proceedings of the 33rd International Conference on Machine Learning*, pages 173–182. PMLR, 2016.

[3] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys*, 52(4), 2019.

[4] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: Seamless operating system integration of peer-to-peer DMA between SSDs and GPUs. In *Proceedings of the 2017 USENIX Annual Technical Conference*, pages 167–179. USENIX Association, 2017.

[5] Alexander Buslaev, Vladimir I. Iglovikov, Eugene Khvedchenya, Alex Parinov, Mikhail Druzhinin, and Alexandr A. Kalinin. Albumentations: Fast and flexible image augmentations. *Information*, 11(2), 2020.

[6] Zydan Bybin, Mohammed Khandaker, Monika Sane, and Graham Hill. Over-provisioning NAND-based intel SSDs for better endurance. *Intel White Paper*, 2019.

[7] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich Haratsch, Adrian Cristal, Osman Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *Proceedings of the 2012 IEEE 30th International Conference on Computer Design*, pages 94–101. IEEE, 2012.

[8] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174v2*, 2016.

[9] Xiaoming Chen, Danny Chen, and Xiaobo S. Hu. moDNN: Memory optimal DNN training on GPUs. In *Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition*, pages 13–18, 2018.

[10] Ekin D. Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V. Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, June 2020.

[11] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988. Association for Computational Linguistics, 2019.

[12] Jia Deng, Wei Dong, Richard Socher, Li jia Li, Kai Li, and Li Fei-fei. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2009.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[14] NVIDIA GDRCopy: A low-latency GPU memory copy library based on GPUDirect RDMA. https://github.com/NVIDIA/gdrcopy.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[16] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677v2*, 2018.

[17] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[18] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture*, pages 620–629. IEEE, 2018.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778. IEEE, 2016.

[20] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: Closing the generalization gap in large batch training of neural networks. In *Proceedings of the Advances in Neural Information Processing Systems 30*, pages 1731–1741. Curran Associates, Inc., 2017.

[21] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of the International Systems and Storage Conference*, pages 10:1–10:9. ACM, 2009.

[22] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Weinberger. Densely connected convolutional networks. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2261–2269. IEEE, 2017.

[23] Yermalayeu Ihar, Antonenka Mikhail, Radchenko Andrey, Dmitry Fedorov, Kirill Matsaberydze, Artur Voronkov, and Facundo Galan. SIMD library. http://ermig1979.github.io/Simd/.

[24] Yuki Ito, Ryo Matsumiya, and Toshio Endo. ooc_cuDNN: Accommodating convolutional neural networks over GPU memory capacity. In *Proceedings of the 2017 IEEE International Conference on Big Data*, pages 183–192. IEEE, 2017.

[25] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, pages 776–789, 2018.

[26] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 61–74. USENIX Association, 2014.

[27] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019.

[28] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 463–479. USENIX Association, November 2020.

[29] McCallum John C. Price and performance changes of computer technology with time. http://www.jcmit.net/.

[30] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *Proceedings of the 5th International Conference on Learning Representations*, 2017.

[31] Tushar Khot, Ashish Sabharwal, and Peter Clark. SciTaiL: A textual entailment dataset from science question answering. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, pages 5189–5197, 2018.

[32] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, pages 283–296. USENIX Association, February 2020.

[33] Youngeun Kwon and Minsoo Rhu. Beyond the memory wall: A case for memory-centric HPC system for deep learning. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 148–161. IEEE, 2018.

[34] Yonggang Li, Guosheng Hu, Yongtao Wang, Timothy Hospedales, Neil M. Robertson, and Yongxin Yang. DADA: Differentiable automatic data augmentation. *arXiv preprint arXiv:2003.03780*, 2020.

[35] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E Gonzalez. Train large, then compress: Rethinking model size for efficient training and inference of transformers. *arXiv preprint arXiv:2002.11794*, 2020.

[36] Xiangru Lian and Ji Liu. Revisit batch normalization: New understanding and refinement via composition optimization. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, pages 3254–3263, 2019.

[37] Sungbin Lim, Ildoo Kim, Taesup Kim, Chiheon Kim, and Sungwoong Kim. Fast autoaugment. In *Proceedings of the Advances in Neural Information Processing Systems 32*, pages 6665–6675. Curran Associates, Inc., 2019.

[38] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens van der Maaten. Exploring the limits of weakly supervised pretraining. *arXiv preprint arXiv:1805.00932*, 2018.

[39] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. DRAGON: Breaking GPU memory capacity limits with direct NVM access. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, pages 32:1–32:13. IEEE, 2018.

[40] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

[41] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *arXiv preprint arXiv:2007.06775*, 2020.

[42] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. Unified memory in CUDA 6.0: a brief overview of related data access and transfer issues. *Tech. Rep. TR-2014–09, University of Wisconsin-Madison*, 2014.

[43] Newegg.com. https://www.newegg.com/.

[44] NVIDIA. Training with mixed precision. https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html.

[45] Intel Optane SSD 905P series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series/optane-ssd-905p-series.html.

[46] Yongjin Park and Manolis Kellis. Deep learning for regulatory genomics. *Nature Biotechnology*, 33(8):825, 2015.

[47] David A. Patterson. Lecture 20: Domain-specific architectures and the google TPU, UC Berkeley CS152 Computer Architecture and Engineering. http://www-inst.eecs.berkeley.edu/~cs152/sp19, 2019.

[48] PCI-SIG. PCI-SIG® member companies announce support for the PCI express® 5.0 specification. https://pcisig.com.

[49] Samsung PM1725b NVMe SSD. http://image-us.samsung.com/SamsungUS/PIM/Samsung_1725b_Product.pdf.

[50] PyTorch. https://pytorch.org.

[51] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 2019.

[52] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250v3*, 2016.

[53] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, volume 33, page 4780–4789. Association for the Advancement of Artificial Intelligence, 2019.

[54] Jerome Revaud, Minhyeok Heo, Rafael S. Rezende, Chanmi You, and Seong-Gyun Jeong. Did it change? Learning to detect point-of-interest changes for proactive map updates. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4081–4090. IEEE, 2019.

[55] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 18:1–18:13. IEEE, 2016.

[56] Samsung Semiconductor. http://www.samsung.com/semiconductor/.

[57] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, 2019.

[58] Intel storage performance development kit. http://www.spdk.io/.

[59] Peng Sun, Yonggang Wen, Ruobing Han, Wansen Feng, and Shengen Yan. GradientFlow: Optimizing network performance for large-scale distributed DNN training. *IEEE Transactions on Big Data*, pages 1–1, 2019.

[60] Aarne Talman, Anssi Yli-Jyrä, and Jörg Tiedemann. Sentence embeddings in NLI with iterative refinement encoders. *Natural Language Engineering*, 25(4):467–482, 2019.

[61] TensorFlow. Data augmentation. https://www.tensorflow.org/tutorials/images/data_augmentation.

[62] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. SuperNeurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 41–53. ACM, 2018.

[63] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, page 7686–7695. Curran Associates Inc., 2018.

[64] Han Xiao. Bert-as-service. https://github.com/hanxiao/bert-as-service, 2018.

[65] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. Scaling for edge inference of deep neural networks. *Nature Electronics*, 1(4):216–222, 2018.

[66] Chih-Chieh Yang and Guojing Cong. Accelerating data loading in deep neural network training. In *Proceedings of the 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics*, pages 235–245. IEEE Press, 2019.

[67] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. XLNet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237v2*, 2020.

[68] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large-batch training for LSTM and beyond. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2019.

[69] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 2018.

[70] Samsung Z-SSD SZ985. https://www.samsung.com/semiconductor/global.semi.static/Brochure_Samsung_S-ZZD_SZ985_1804.pdf.

[71] Jie Zhang, David Donofrio, John Shalf, Mahmut T. Kandemir, and Myoungsoo Jung. NVMMU: A non-volatile memory management unit for heterogeneous GPU-SSD architectures. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation Techniques*, pages 13–24. IEEE, 2015.

[72] Bojian Zheng, Abhishek Tiwari, Nandita Vijaykumar, and Gennady Pekhimenko. Echo: Compiler-based GPU memory footprint reduction for LSTM RNN training. *arXiv preprint arXiv:1805.08899v5*, 2019.

[73] Jingbo Zhou, Qi Guo, H. V. Jagadish, Lubos Krcal, Siyuan Liu, Wenhao Luan, Anthony Tung, Yueji Yang, and Yuxin Zheng. A generic inverted index framework for similarity search on the GPU. In *Proceedings of the 2018 IEEE 34th International Conference on Data Engineering*, pages 893–904. IEEE, 2018.

[74] Ligeng Zhu, Ruizhi Deng, Michael Maire, Zhiwei Deng, Greg Mori, and Ping Tan. Sparsely aggregated convolutional networks. In *Proceedings of the European Conference on Computer Vision*, pages 186–201, 2018.

D2FQ: Device-Direct Fair Queueing for NVMe SSDs

Jiwon Woo, Minwoo Ahn, Gyusun Lee, Jinkyu Jeong
Sungkyunkwan University
{jiwon.woo, minwoo.ahn, gyusun.lee}@csi.skku.edu, jinkyu@skku.edu

Abstract

With modern high-performance SSDs that can handle parallel I/O requests from multiple tenants, fair sharing of block I/O is an essential requirement for performance isolation. Typical block I/O schedulers take three steps (submit-arbitrate-dispatch) to transfer an I/O request to a device, and the three steps incur high overheads in terms of CPU utilization, scalability and block I/O performance. This motivates us to offload the I/O scheduling function to a device. If so, the three steps can be reduced to one step (submit=dispatch), thereby saving CPU cycles and improving the I/O performance.

To this end, we propose D2FQ, a fair-queueing I/O scheduler that exploits the NVMe weighted round-robin (WRR) arbitration, a device-side I/O scheduling feature. D2FQ abstracts the three classes of command queues in WRR as three queues with different I/O processing speeds. Then, for every I/O submission D2FQ selects and dispatches an I/O request to one of three queues immediately while satisfying fairness. This avoids time-consuming I/O scheduling operations, thereby saving CPU cycles and improving the block I/O performance. The prototype is implemented in the Linux kernel and evaluated with various workloads. With synthetic workloads, D2FQ provides fairness while saving CPU cycles by up to 45% as compared to MQFQ, a state-of-the-art fair queueing I/O scheduler.

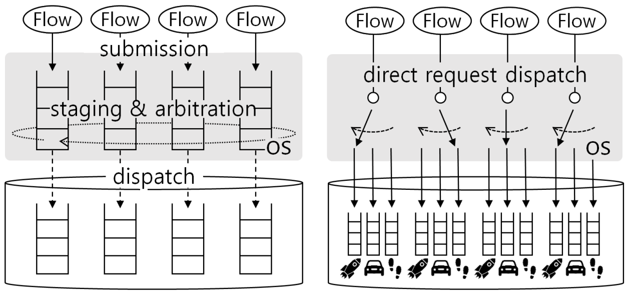
1 Introduction

Modern high-performance solid-state drives (SSDs) can deliver one million I/O operations per second (e.g., Samsung 980 Pro [1]). Such SSDs are also equipped with multiple I/O command queues to enable parallel I/O processing on multi-core processors. Thus, SSDs can accommodate multiple independent I/O flows in multi-tenant computing environments such as cloud data centers. In such an environment, fair sharing of the SSD performance is important to provide performance isolation between multiple applications or tenants.

A fair-share I/O scheduler [3, 8, 9, 29, 30, 34, 35] distributes storage performance proportionally to the weight of the applications. And, it is usually implemented at the block layer of the I/O stack. The typical block I/O scheduler takes three steps (submit-arbitrate-dispatch) during I/O processing (Figure 1a). When applications submit I/O requests, the I/O scheduling layer arbitrates and stages the I/O requests in the layer. Whenever an I/O scheduling condition is met (e.g., fairness), some staged I/O requests are eventually dispatched to the storage device. A problem is that these three-stage operations incur high CPU overhead, long I/O latency, and low I/O performance on high-performance SSDs. Since modern high-performance SSDs are shifting the bottleneck from I/O to CPU, many applications are changing their algorithms and/or data structures to adapt to the bottleneck changes [10, 17, 21]. With considering these efforts in reducing CPU overheads, reducing the CPU overheads associated with block I/O scheduling is also an important issue.

Offloading the I/O scheduling function to a device is an attractive approach to reducing the CPU overhead while preserving fairness. This scheduling offloading is already widely used in the domain of network packet scheduling [7,23,31,32] since many network interface cards have device-side I/O scheduling features, such as round-robin scheduling. Fortunately, modern storage devices are now having a device-side I/O scheduling feature called NVMe weighted round-robin (WRR) queue arbitration. It provides three priority classes of I/O command queues, each with a configurable weight, and applies the weighted round-robin queue arbitration during I/O processing by the SSD firmware. However, a challenge is that the basic NVMe WRR is too simple to properly schedule I/O requests from multiple tenants with various I/O characteristics, such as a varied number of threads, different I/O request rates, and various request sizes.

This paper proposes D2FQ, a **d**evice-**d**irect **f**air **q**ueueing scheme for NVMe SSDs. D2FQ leverages the NVMe WRR feature but does not use it as it is. It abstracts the three queue classes as three-class queues with different I/O processing speeds. Then, for every I/O request submission, D2FQ selects an I/O command queue and dispatches an I/O request to the queue immediately (Figure 1b). The queue selection policy



(a) Submit-arbitrate-dispatch **(b)** Submit=dispatch

Figure 1: Typical I/O scheduling in the block layer (a) and the proposed I/O scheduling (b).

is carefully designed to provide fairness while reducing tail latency as much as possible. Since the arbitration step is removed and the submission and dispatch steps are unified in the block layer, D2FQ can minimize the CPU overhead during I/O scheduling and improve the I/O performance.

D2FQ also leverages a scalable yet sloppy minimum value tracking method. Similar to other fair-share I/O schedulers, D2FQ is virtual time-based. In these schedulers, it is important to track the minimum virtual time in a scalable way [11]. The proposed minimum tracking method tracks the minimum value almost always while having a small window of tracking a non-minimum value. However, this little possibility of incorrect tracking allows us to achieve scalability a lot as compared to a scalable minimum tracking object in the literature. [22]

D2FQ is implemented in the Linux kernel and evaluated with various workloads. Using the FIO benchmark with various workload configurations, our scheme provides fairness while reducing CPU utilization by up to 45% as compared to MQFQ [11], the state-of-the-art fair queueing scheme. When the storage device is not the bottleneck, D2FQ outperforms other schedulers in terms of I/O latency, CPU utilization, and reaches the maximum storage bandwidth faster than the other I/O schedulers. Since D2FQ unifies the I/O submission and dispatch steps into one, it can be integrated with the low-latency I/O stack, which has no I/O scheduling capability [19]. After the integration with the low-latency I/O stack, it outperforms other schemes, reducing the I/O latency by up to 35% and improving the I/O bandwidth by up to 54%.

This paper has the following contributions:

- We successfully demonstrate to build a fair-queueing I/O scheduler (D2FQ) on top of a simple yet efficient device-side scheduling feature (NVMe WRR). The only necessary abstraction is the device-side I/O queues with different I/O processing speeds.
- We propose a scalable yet sloppy minimum tracking method suitable for the virtual time-based fairness of D2FQ.
- We provide a detailed evaluation of the proposed fair-queuing I/O scheduler. The evaluation results demonstrate that D2FQ provides fairness, low CPU utilization, and high block I/O performance.

2 Background & Motivation

2.1 Fair Queueing for SSDs

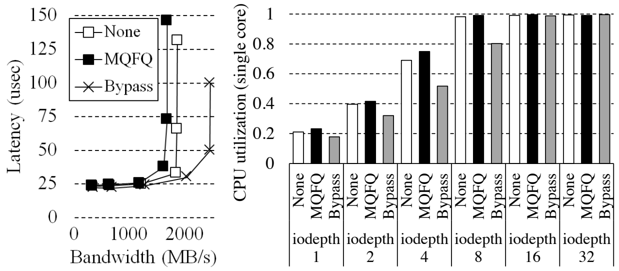
Modern high-performance SSDs are capable of accommodating parallel I/O requests from multiple tenants. For example, Samsung 980 Pro can perform at a million I/O operations per second [1]. This huge increase in the bandwidth and capacity of SSDs enable to service I/O requests from multiple independent workloads (or tenants) in a single storage device. Naturally, fair sharing of the SSD bandwidth is important to meet the service-level agreements of applications and to provide performance isolation between tenants. Among many proportional share I/O schedulers [3,8,9,29,30,34,35], virtual time-based fair queueing is an attractive solution for SSDs due to its work-conserving nature. They can maximize the SSD throughput while the bandwidth achieved by each tenant is proportional to the weight of the tenants.

An I/O flow is a stream of I/O requests issued by a resource principal [11] (e.g., virtual machines, Linux cgroups, thread groups), and the virtual time of a flow is the normalized accumulated I/O size serviced to the tenant. When a flow f is serviced an I/O request of length l, the virtual time vt_f of the flow is advanced by l/w_f where w_f is the weight of the flow.

Virtual time-based fair queueing I/O schedulers [9, 29] schedule I/O requests while minimizing the difference in virtual time between any flows. If all flows have the same virtual time, their I/O resource usages are proportional to their weights, and hence the fairness is satisfied. Accordingly, the goal of the schedulers is to minimize the virtual time gap between any flows. Hence, a flow with the minimum virtual time is only allowed to dispatch its I/O request since it is the flow with the lowest amount of I/O serviced. I/O requests from other flows are throttled by staging them in the I/O scheduler. This arbitration of request dispatching is denoted as I/O scheduling in the block layer, as shown in Figure 1a.

To maximize the performance of modern SSDs with internal parallelism, it is necessary to sustain a high number of in-flight requests. Accordingly, modern fair queueing I/O schedulers [11, 13] relax the strict fairness. Hence, flows whose virtual time is nearby the minimum virtual time are allowed to dispatch their I/O requests. This may allow a small amount of short-term unfairness but improves the overall I/O throughput by maximally utilizing the storage device.

Among fair queueing I/O schedulers, multi-queue fair queueing (MQFQ) [11] is the state-of-the-art approach for modern high-performance multi-queue SSDs. SSDs now have multiple command queues to utilize the internal parallelism of SSDs effectively. To scale with multiple command queues, MQFQ has request queues for each core in the I/O scheduler and employs scalable arbitration between the per-core I/O request queues. It employs two scalable objects: Mindicator [22] for scalable tracking of the minimum virtual time and a token tree [11] for scalable communication across cores.



(a) Latency/bandwidth **(b)** CPU (single core) utilization

Figure 2: (a) I/O latency and bandwidth and (b) CPU (single core) utilization of a single-thread 4 KB random read workload with varying I/O depth on Machine A in Table 1.

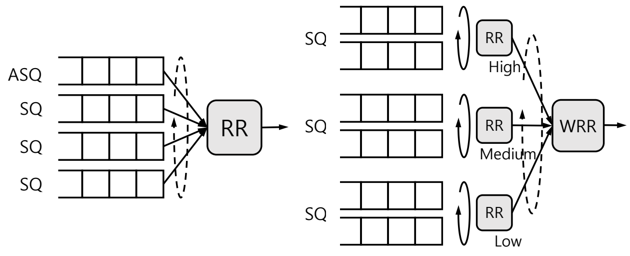
With these objects, MQFQ keeps the number of in-flight requests high to maximize the performance of multi-queue SSDs while not significantly violating the short-term fairness. The MQFQ is prototyped in the block layer of the operating system (OS) I/O stack.

2.2 I/O Scheduling Overheads in Software

The block layer in the OS I/O stack (e.g., *blk-mq* in Linux [5]) is the core of block I/O scheduling. The generic block layer provides merging, reordering, staging, and accounting of I/O requests. In addition, block I/O schedulers (e.g., BFQ [4], mq-deadline [25], kyber [18]) are implemented as a module of the block layer. The multi-queue block layer maintains I/O *request queues* to stage I/O requests for I/O scheduling. Without I/O scheduling, the submitted requests are immediately dispatched to I/O *command queues* of a storage device (e.g., NVMe submission queues). With I/O scheduling, the scheduler arbitrates dispatching of I/O requests by staging them inside the scheduler. If the scheduling condition satisfies after processing other requests, the staged I/O requests are finally dispatched to the device.

With high-performance SSDs, however, the block layer incurs overheads in terms of CPU cycles and I/O latency. Accordingly, many studies have proposed to bypass the block I/O layer to achieve low I/O latency [19, 38]. Figure 2 shows how the overhead of the block layer affects I/O latency, I/O bandwidth, and CPU utilization. We compared the vanilla Linux kernel without I/O scheduling (*None*), *MQFQ*, and the light-weight block layer (*Bypass*) [19], which bypasses the block layer and submits I/O requests directly to the device's command queues. MQFQ shows the highest I/O latency and lowest I/O bandwidth in Figure 2a because the CPU is saturated earlier than the other schemes. None shows moderate performance, and Bypass shows the lowest I/O latency, highest I/O bandwidth, and lowest CPU utilization; it delays the saturation point further than the other schemes because of its lowest CPU overhead of block I/O service.

The high CPU cost of the block I/O scheduling can exacerbate the problem of CPU bottleneck in modern data-intensive applications with fast SSDs. With the introduction of low-latency SSDs, the performance bottleneck is moving from an I/O device to CPU [10, 17, 21]. This incurs the need for lowering the CPU contention by changing data structures and/or algorithms of applications. Hence, the CPU overheads caused by block I/O scheduling can also be addressed to fully harness the performance potential of high-performance SSDs today.



(a) Round robin **(b)** Weighted round robin

Figure 3: Two NVMe queue arbitration policies: (a) round-robin and (b) weighted round-robin.

The high overhead of I/O scheduling can be alleviated by offloading I/O scheduling function to devices. Network interface cards (NICs) have experienced the era of microsecond-scale I/O latency earlier than SSDs. Many approaches have proposed to offload packet scheduling to NIC and succeeded in lowering the CPU utilization [7, 31, 32, 36]. Similarly, the block I/O scheduling can be offloaded to SSDs having device-side I/O scheduling features [14, 15, 26, 27, 33], and therefore the cost of the block I/O scheduling can be reduced.

2.3 Weighted round-robin in NVMe Protocol

Non-volatile memory express (NVMe) [26] is the de-facto standard interface bridging computer systems with storage devices due to its simplicity, efficiency, and scalability. The protocol also has a block I/O scheduling feature called weighted round-robin (WRR) queue arbitration [26]. The default I/O command scheduling policy of the protocol is round-robin; hence processing I/O commands one by one across command queues as shown in Figure 3a. If the WRR feature is enabled, the SSD firmware fetches I/O commands in a weighed round-robin fashion as shown in Figure 3b. With WRR enabled, command queues are classified into three priority classes (low, medium, and high)[1], and queues in each priority class are assigned a *queue weight* (1 – 256); hence queues in the same priority class share a queue weight. With WRR enabled, if queue weights are 1, 2, and 3 for the low, medium, and high queues, respectively, the SSD controller fetches three I/O commands from the high queues, then fetches two commands from the medium queues and then fetches one from the

[1] The NVMe WRR also supports another queue priority class called urgent priority but our scheme does not consider the use of the class

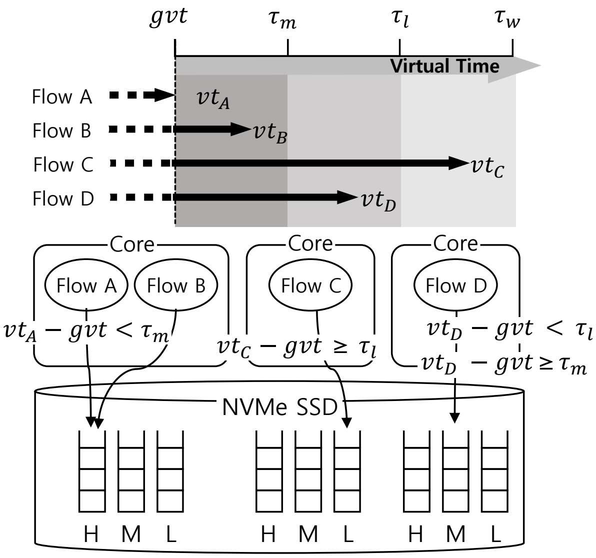


Figure 4: Overview of D2FQ.

low queues. Queues in the same priority class are accessed in a round-robin fashion.

The NVMe WRR feature can be easily implemented inside the SSD because of its simplicity. However, applying this to fair queueing has many challenges to be resolved. First, NVMe WRR has only three priority classes (i.e., low, medium and high), whereas the number of tenants can be higher. Second, its queue arbitration does not consider I/O sizes. Finally, the weight ratio between any two queues could not directly match the ratio of I/O commands serviced from the two queues. This is because the number of I/Os actually processed can vary depending on the utilization of the command queues. Consequently, it is necessary to bridge the gap between the requirement of fair queueing and the simple yet uncertain performance characteristic of NVMe WRR.

3 Device-Direct Fair Queuing

3.1 Overview

This paper proposes a fair queueing scheme called device-direct fair queueing (D2FQ) for NVMe SSDs. D2FQ offloads the I/O scheduling functionality to an SSD by exploiting the NVMe WRR feature. Accordingly, the CPU overheads and I/O latency associated software I/O scheduling can be reduced. Figure 4 shows the overview of D2FQ.

D2FQ is a virtual time-based fair queueing scheme. It manages the virtual time of each flow and the global virtual time (gvt), the minimum virtual time among active flows. An *active* flow is a flow with any pending I/O requests to be served. As other fair queueing schemes do, D2FQ provides fairness between only active flows.

D2FQ throttles a flow if its virtual time is far ahead of gvt. Throttling is done not by the block layer of the I/O stack but by exploiting the NVMe WRR feature. In addition, D2FQ does not establish any fixed mapping between flows and I/O command queues. Instead, our scheme abstracts the three classes of queues as three different queues with different I/O processing speeds (fast, moderate and slow). Then, whenever a flow submits an I/O request, our scheduling policy immediately selects a queue of the desired speed and dispatch the request to the queue (Figure 1b). As a result, slow flows in the virtual time domain are enforced to use the fast queues to catch up the virtual time of other flows, and fast flows are throttled by using the slow or moderate queues.

D2FQ maintains three threshold values: τ_m, τ_l and τ_w; the former two thresholds are used during the queue selection, and the latter is used to detect unfairness which is explained later in Section 3.2. When a flow f issues an I/O request, the gap between its virtual time and gvt (i.e., $vt_f - gvt$) is compared with the two threshold values to select the class of command queue (SQ) for I/O dispatching as follows:

$$SQ = \begin{cases} Q_{high} & \text{if } vt_f - gvt < \tau_m \\ Q_{mid} & \text{else if } vt_f - gvt < \tau_l \\ Q_{low} & \text{otherwise} \end{cases} \quad (1)$$

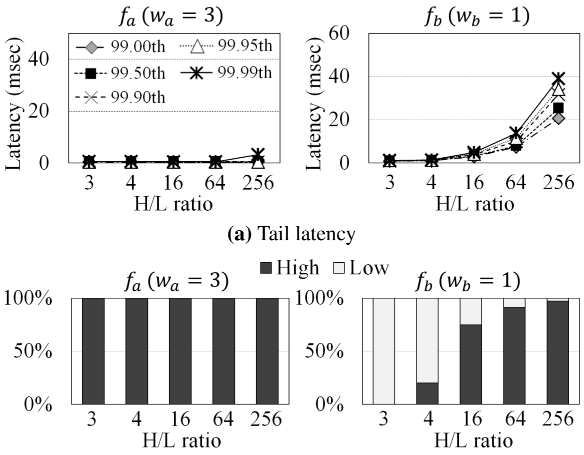
Hence, if the virtual time of a flow is not far from gvt (vt_B in Figure 4), its I/O requests are queued to high queues; hence the flow is not throttled. If a flow is far ahead of gvt (vt_C in the figure), its I/O requests are queued to low queues; hence the flow is throttled. Please note that our scheme assumes all cores are having their own queue set (three queues of each priority class).

Example walkthrough. Let us assume two flows f_a and f_b and their weights $w_a = 3$ and $w_b = 1$. Both flows issue 4 KB I/O requests with high I/O depth. If the weight of high queues is 3 and the weight of low queues is 1, both flows can fairly share the bandwidth by making flow f_a use the high queues and flow f_b use the low queues. However, our scheme does not statically map any flow to any queue but establishes the mapping dynamically. Indeed, at the beginning, both flows use the high queues together because their virtual time gap is zero. Then, vt_a advances by $4\ KB/3$ while vt_b advances by $4\ KB/1$ on each I/O completion; consequently, vt_b advances 3 times faster than vt_a. In the end, $vt_b - gvt (= vt_a)$ exceeds τ_l, and flow f_b begins to use the low queues. After that, both flows have the same virtual time progress rate.

We define the term *H/L ratio* as the ratio of the weight of high queues over the weight of low queues. The H/L ratio is the most important factor in satisfying the I/O fairness. It determines the maximum speed difference the high and low queues can produce if I/O sizes are identical and the queues are fully utilized. Hence, it determines the maximum weight ratio our scheme can cover with fairness.

A small H/L ratio cannot meet the fairness requirement. In the previous example, if the H/L ratio is 2, the two flows f_a and f_b cannot fairly share the I/O bandwidth.

Meanwhile, a high H/L ratio has a wide coverage of weight



(a) Tail latency

(b) Queue usage

Figure 5: Effect of the H/L ratio to tail latency and queue usage.

ratios between any flows. In the above example, if the H/L ratio is 6, three fifth of I/O requests from f_b need to use the high queues, and then the two flows can meet the fair bandwidth distribution. Since the low queues are $1/6$ times slower than the high queues, which is more than necessary, f_b needs to use the high queues to compensate the penalty caused by the use of the low queues.

The use of high H/L ratio seems appropriate. However, this has a side-effect of increasing the tail of I/O latency. Figure 5 shows the tail latency and queue utilization of flow f_a and f_b with varying the H/L ratio from 3 to 256. As shown in the figure, the H/L ratio of 3 shows the lowest tail latency for f_b. In that configuration, f_b uses the low queues only while the f_a uses high queues only. However, with high H/L ratios, flow f_b shows high tail latency while increasing its usage portion of the high queues. The flow f_b needs to be throttled but the use of the low queues with high H/L ratio gives higher penalty than necessary. This results in the increase of tail latency and the increase of the high queue usage.

Although the above examples show a simple workload having only two flows with a fixed I/O size and high I/O submission rate. However, real-world workloads may have a various number of flows with any number of threads, I/O submission rates and I/O sizes. With these realistic and unknown I/O characteristics, it is challenging to find the proper H/L ratio to make queues with sufficient I/O processing speed difference.

3.2 Dynamic H/L Ratio Adjustment

D2FQ finds proper weights of the three queue classes to meet the two goals: providing fairness and taming tail latency. As explained above, the H/L ratio is the most important factor since D2FQ needs to satisfy fairness. In this regard, our scheme finds a proper H/L ratio first and then sets the weight of medium queue as the square root of the H/L ratio. Hence, the speed ratio between high and medium queues is equal to that between medium and low queues.

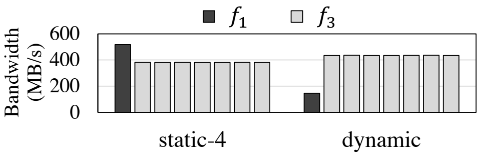


Figure 6: Effect of the dynamic H/L ratio adjustment.

D2FQ collects information of the virtual time of all flows and their I/O usage statistics, such as queues and I/O sizes. It uses the collected information to find the appropriate H/L ratio periodically as follows.

3.2.1 Increasing H/L Ratio

The H/L ratio needs to increase when fairness is not satisfied. Recall that our scheme maintains three thresholds, and the third one τ_w is the threshold to detect unfairness and trigger the process of finding a proper H/L ratio. Hence, if a flow with the largest virtual time is a far ahead of gvt by τ_w, D2FQ finds a new H/L ratio that is suitable in providing fairness.

To this end, D2FQ keeps track of two flows, one with the largest virtual time (denoted as f_{max}) and the other with the smallest virtual time (denoted as f_{min} whose virtual time $vt_{f_{min}}$ is equal to gvt). Then, it calculates the delta of virtual time increase in the last information collection period. Hence, Δvt_{max} is the virtual time increase rate in the last period by f_{max}, and Δvt_{min} is the virtual time increase rate last period by f_{min}. Then, the next H/L ratio is calculated by using the following formula:

$$\text{H/L ratio}_{next} = \lfloor \frac{\Delta vt_{max}}{\Delta vt_{min}} \times \text{H/L ratio}_{prev} \rfloor + 1 \qquad (2)$$

The term $\frac{\Delta vt_{max}}{\Delta vt_{min}}$ is the ratio of widening virtual time gap between f_{max} and f_{min}, and this has happened under the previous H/L ratio. Accordingly, the next H/L ratio should be the product of the widening ratio and the previous H/L ratio. The next H/L ratio is ensured to have a higher value by one than the proper H/L ratio to make the gap narrowed down next.

The use of high queues does not guarantee that I/O requests in the high queues are processed faster than those in the low queues. However, our dynamic weight adjustment finds out a proper H/L ratio to meet the fairness. Figure 6 shows the bandwidth distribution of eight flows, one with weight 1 (f_1) the other seven flows with weight 3 (f_3). When the H/L ratio is fixed to 4 (*static-4*), f_1 uses the low queues and seven f_3 flows use the high queues. In this case, all the flows are not allocated fair amount of I/O resource due to the contention in the high queues. However, if our *dynamic* H/L ratio adjustment is applied, the H/L ratio becomes 22 using Equation 2 and all the flows meet the fair bandwidth distribution; the required effective queue weight ratio is 1:21 (1:3 weight ratio with 1:7 ratio of the number of flows) and one is incremented using the equation.

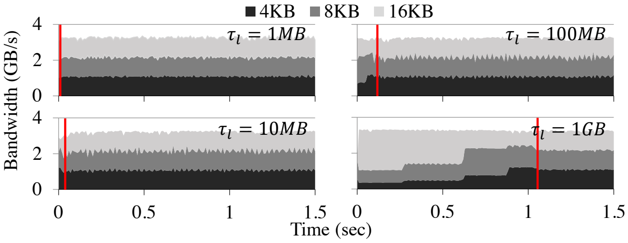


Figure 7: Time-series bandwidth of three flows with different I/O sizes (4 KB, 8 KB and 16 KB) with varying the threshold τ_l from 1 MB to 1 GB.

3.2.2 Decreasing H/L Ratio

As explained in Section 3.1, an unnecessarily high H/L ratio may increase the tail latency of flows requiring throttling. Hence, it is necessary to decrease the H/L ratio if the current H/L ratio is too high. The condition to decrease the H/L ratio is when the maximum virtual time gap is below τ_w. In this case, D2FQ calculates the *virtual slowdown* of each flow using the I/O statistics in the last statistics collection period. The virtual slowdown is an estimated value of how the I/O requests of this flow are slowed down by not using the high queues. The virtual slowdown of flow f is calculated by using the following formula where $\sum l_{f,x}$ is the total amount of I/O submitted to the queue class x by flow f in the last period and $\frac{p_x}{p_y}$ is the weight ratio of two queue classes x and y:

$$\text{slowdown}(f) = \frac{\sum l_{f,h} * \frac{p_h}{p_h} + \sum l_{f,m} * \frac{p_h}{p_m} + \sum l_{f,l} * \frac{p_h}{p_l}}{\sum l_{f,h} + \sum l_{f,m} + \sum l_{f,l}} \quad (3)$$

Then, D2FQ chooses the maximum virtual slowdown among all active flows in the system and sets the next H/L ratio as the maximum value.

3.3 Determining Thresholds

D2FQ regulates the fairness by throttling fast flows in virtual time (i.e., flows with low weight values). The two thresholds (τ_m and τ_l) are the criteria of when to throttle such fast flows.

Large threshold values allow a huge virtual time interval between any flows and *gvt*. Hence, it determines the allowed unfairness in virtual time.

Figure 7 shows the time-series bandwidth of three flows with three I/O sizes: 4 KB, 8 KB and 16 KB, respectively. The vertical line in each figure indicates the point in time starting fair bandwidth sharing. As shown in the figure, with a small threshold (τ_l = 1 MB[2]), the three flows equally share the I/O bandwidth from the beginning. That point is delayed to after 1 second with a large threshold value (1 GB). However, after that point, the flows equally share the I/O bandwidth.

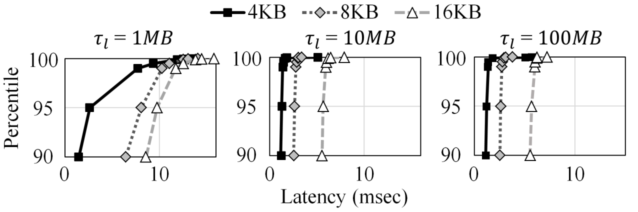


Figure 8: Tail latency of the three flows with varying the threshold τ_l: 1 MB, 10 MB and 100 MB.

In our scheme, the virtual time progression is controlled by making the flow crossing the threshold boundary back and forth. Thus, a flow could not get proper throttling until the flow hits thresholds in the virtual time domain. Hence, the threshold values only determine when this control begins.

On the other hand, small threshold values may unintentionally increase the tail latency of flows, especially those who stay back in the virtual time domain. Such flows are intended to use the high queues only. However, a small increase in virtual time can make such flows use the medium or low queues due to crossing the thresholds. This may exacerbate the tail latency of all flows because other flows can unintentionally use the high queues and be throttled to offset the benefits of using the high queues.

Figure 8 compares the tail latency of the three flows with varying threshold values: 1 MB, 10 MB, and 100 MB. As shown in the figure, with the 10 or 100 MB threshold values, the three flows show no significant increase in the tail latency. However, with the 1 MB threshold value, the three flows show up to 3.7 times long tail latency.

Consequently, there is a trade-off between short-term fairness and tail latency in setting the threshold value. Depending on whether a user focuses on tail latency or short-term fairness, the user can adjust the appropriate threshold value.

The characteristics of workloads, especially I/O size and weight of flows, impact on the selection of the threshold values because the I/O size and weight determine the stride of virtual time increase. Our scheme uses a proper τ_l that is empirically found to work with our tested workloads. We leave the fine-tuning of the threshold values to the users or system administrators.

τ_m also affects tail latency. However, its latency impact is not significant as compared to that of τ_l since the medium queues are faster than the low queues. We empirically found that it is suitable to set $\tau_m = \tau_l/2$.

3.4 Global Virtual Time Tracking

The value *gvt* is frequently accessed during I/O submission and completion. Accordingly, it is important to track *gvt* in a scalable way.

Tracking *gvt* is equal to tracking the minimum among a set of values where each value changes simultaneously. One coarse-grained approach is to inspect all the values for every

[2] τ_l = 1 MB indicates that a flow with weight 8 can cross the threshold boundary after it is serviced 1 MB I/O size. If its weight is 1, the flow can meet the threshold only after 128 KB I/O size serviced (one eighth of 1 MB).

```
 1 struct vt {
 2   u64 id : 16 bits // id of a flow
 3   u64 vt : 48 bits // virtual time of a flow
 4 } gvt; // global virtual time
 5
 6 void update_gvt (vt my)
 7   while (true) {
 8     vt old = gvt
 9     if ((old.id == NO_HOLDER)
10     || (old.id == my.id && old.vt < my.vt)
11     || (old.id != my.id && old.vt > my.vt)) {
12       if ( CAS(&gvt, old, my) == SUCCESS )
13         return
14     } else
15       return
16   }
17
18 void release_gvt (vt my)
19   while (true) {
20     vt new, old = gvt
21     if (old.id == my.id) {
22       new.id = NO_HOLDER; new.vt = old.vt
23       if ( CAS(&gvt, old, new) == SUCCESS )
24         return
25     } else
26       return
27   }
```

Figure 9: Pseudocode of tracking the global virtual time.

query of the minimum. An alternative is to use a scalable minimum tracking object such as Mindicator [22].

In D2FQ, we take yet another approach of tracking *gvt* in a sloppy way. We consider that it is not always necessary to retrieve the true minimum value among the virtual time of flows. The goal of tracking *gvt* is not to minimize the virtual time gap between any flows. It is to make the pace of virtual time progression of any flows at a similar rate. If the value of *gvt* is not far from the true minimum value, the sloppy management hardly affects the policy of low or medium queue selection since the use of the thresholds gives tolerance to the queue selection policy.

In this regard, our scheme maintains the *gvt holder*, the flow owning *gvt*, and allows only the *gvt* holder to be able to increase *gvt* (line 9–13 in Figure 9). Other flows can also update *gvt* but only when their virtual time is smaller than *gvt* (line 11). A little inaccuracy can happen when the *gvt* holder increases *gvt* and it now overtakes the virtual time of other flows, hence violating that *gvt* is not the minimum. However, this little inaccuracy comes with the simplification of the *gvt* update operation; otherwise, every *gvt* update needs to inspect the virtual time of all the flows.

The function `update_gvt()` is called when I/O completion happens. When the *gvt* holder becomes inactive, it calls `release_gvt()`, and any flow can become the *gvt* holder. We use the atomic instruction `compare_and_swap` (`CAS`) and the while loop to secure minimal serialization between concurrent *gvt* updates.

3.5 Implementation

D2FQ is implemented in the multi-queue block layer [5] of the Linux kernel. Figure 10 represents the high-level pseudo code of D2FQ. Each core has three submission queues (high, medium and low) (line 2). Each flow has virtual time, the number of in-flight requests and its weight value (line 4–7).

```
 1 per-CPU structures:
 2     high/medium/low class SQ
 3
 4 per-flow structures:
 5     vt             // virtual time
 6     nr_inflight    // # of in-flight requests
 7     weight         // I/O weight of this flow
 8
 9 void dispatch_request (request R, flow F)
10     if (F->active == false)
11         F->active = true; F->vt = gvt
12     F->nr_inflight += 1
13     vt_gap = F->vt - gvt
14     if (vt_gap > threshold_low)
15         R->dispatch_Q = low class SQ
16     else if (vt_gap > threshold_medium)
17         R->dispatch_Q = medium class SQ
18     else
19         R->dispatch_Q = high class SQ
20
21 void complete_request (request R, flow F)
22     F->vt += R->length / F->weight
23     F->nr_inflight -= 1
24     if (F->nr_inflight == 0)
25         enter_grace_period(F)
26     update_gvt()
```

Figure 10: Pseudocode for D2FQ working flow.

The function `dispatch_request()` (line 9) is the core function that selects the queue to dispatch an I/O request. It is invoked in the block layer function `blk_mq_start_request()`. However, D2FQ is independent to the block layer since it has no staging operation. Accordingly, it can also be invoked elsewhere before request dispatching, such as `nvme_queue_rq()`.

The number of in-flight requests is used to detect the activeness of flows. If it becomes zero, a grace period is given, and after that the flow becomes idle (line 24). The use of the grace period is to avoid the deceptive idleness [12].

The function `complete_request()` is invoked whenever a request is completed. In our implementation, it is called from the block layer function `blk_mq_finish_request()`. It is also independent to the block layer so it can be invoked elsewhere after request completion.

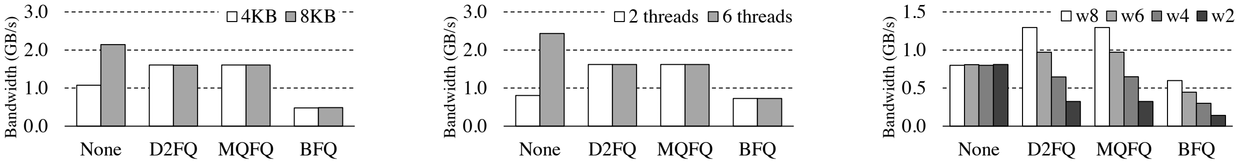
4 Evaluation

4.1 Methodology

Table 1 shows our experimental configuration. We used Samsung Z-SSD as the main storage device because it supports the NVMe WRR feature. The NVMf on ramdisk is used only for the scalability test due to lack of WRR support.

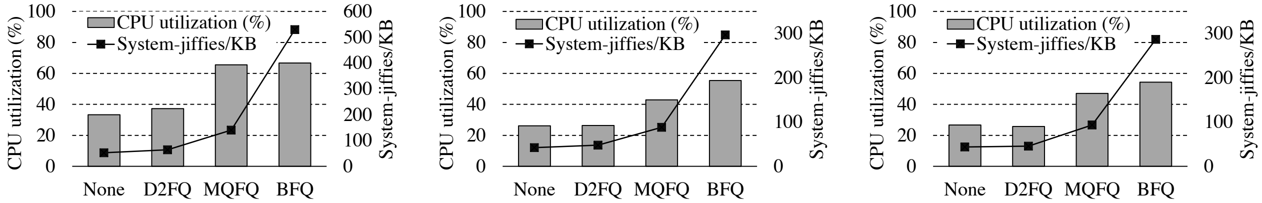
We evaluated the following four schedulers:

- **None** performs no I/O scheduling in the block layer.
- **D2FQ** is the prototype of our scheme which is based on None as explained in Section 3.5. The dynamic H/L ratio adjustment is enabled. The H/L ratio is initially 256. The



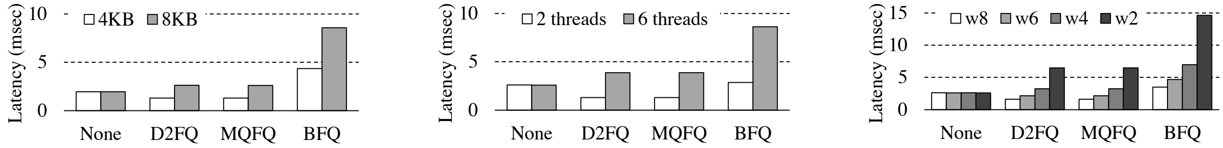
(a) I/O request size (4 KB vs 8 KB) **(b)** # of threads (2 vs 6) **(c)** Weight ratio 8:6:4:2

Figure 11: Fairness measurement with varying (a) I/O request size, (b) the number of threads and (c) the weight of each flow.



(a) I/O request size (4 KB vs 8 KB) **(b)** # of threads (2 vs 6) **(c)** Weight ratio 8:6:4:2

Figure 12: CPU utilization and I/O processing cost of the workloads in Figure 11.



(a) I/O request size (4 KB vs 8 KB) **(b)** # of threads (2 vs 6) **(c)** Weight ratio 8:6:4:2

Figure 13: Average I/O latency of the workloads in Figure 11.

Hardware configuration	**Machine A**	
	CPU	Intel Xeon Gold 5112 3.60 GHz 8 cores
	Memory	DDR4 192 GB
	Storage	Samsung Z-SSD 800 GB
	Machine B	
	CPU	Intel Xeon Gold 6226 2.70 GHz 24 cores
	Memory	DDR4 192 GB
	Storage	Samsung Z-SSD 800 GB NVMf ramdisk 64 GB
	Network	Mellanox Connect X-4 56 Gbps
Software configuration	**OS**	Ubuntu 18.04.4
	Kernel	Linux version 5.3.10
	FIO	libaio, random read, direct I/O
	YCSB	Uniform request distribution

Table 1: Experimental configuration.

threshold τ_l is set to 8 MB for flows with weight 8 which is 1 MB for flows with weight 1; then the rest of the thresholds are automatically set $\tau_m = \tau_l/2$, $\tau_w = 2 \times \tau_l$. The period of the H/L ratio adjustment is set to one second.

- **MQFQ** is the state-of-the-art fair-queueing I/O scheduler. Unfortunately the source code is unavailable. So we made our own implementation of MQFQ, which faithfully follows the design described in the MQFQ paper [11]. We ported the Mindicator written in C++ [24] to C for the integration with the Linux kernel. MQFQ has two parameters: D is 64 and T is 45 KB in our setting.
- **BFQ** is the time slice-based proportional share I/O scheduler in Linux [4]. We set `max_budget` to 256.

The four schedulers use `ionice()` to set the weight of each flow. The weight values range from 1 to 8 in the experiment. Unless specified, the default weight value is 8.

The evaluation is organized to answer the three questions: (1) whether our scheme provides fairness when the storage device is saturated, (2) how well our scheduler shows good I/O performance when the storage device is unsaturated, and (3) whether our scheme provides fairness with realistic workload.

4.2 Fairness

Providing fairness is the primary goal of fair queueing when multiple flows contend on a storage device. We build three workloads with varying the following factors: I/O request size, the number of threads and the weight of flows. Then, we measure the bandwidth of each flow in Figure 11. Unless specified, the I/O depth is 128 by default in each thread.

I/O Request Size. Figure 11a shows the bandwidth distribution of two I/O flows with different I/O request sizes: 4 KB vs 8 KB. Each flow has 4 threads. Since the two flows have the same weight, the SSD bandwidth should be fairly distributed to the two flows, and D2FQ, MQFQ and BFQ provide the fairness. The total bandwidth is identical across the three schedulers: None, D2FQ and MQFQ.

Thread Count. Figure 11b shows the bandwidth distribution of two I/O flows with different number of threads: 2 threads vs 6 threads. The I/O request size is 8 KB in both flows. In this experiment, both flows need to evenly share the SSD bandwidth and D2FQ, MQFQ and BFQ achieve this while None does not.

Weight. Figure 11c shows the bandwidth consumed by four I/O flows with different weight values: 8, 6, 4 and 2; each

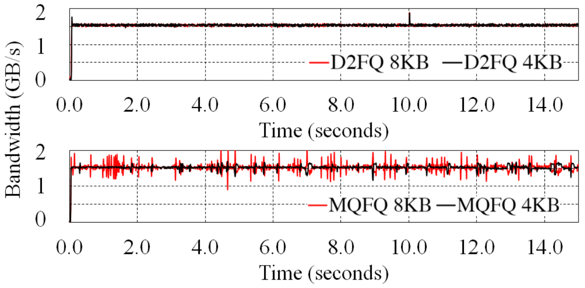


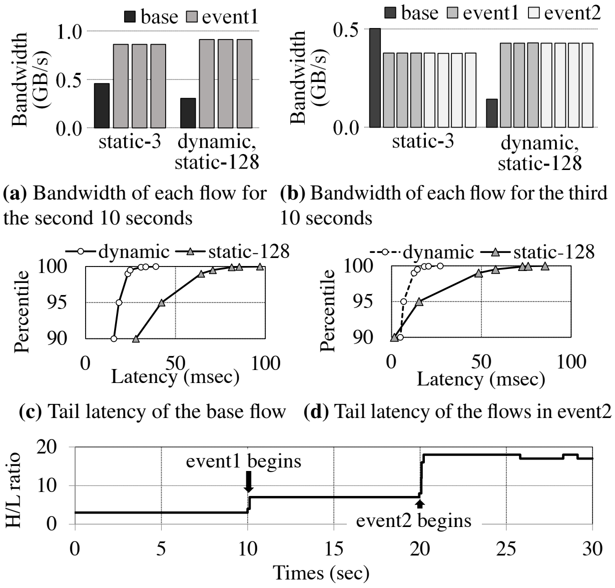
Figure 14: Bandwidth fluctuation test between fair I/O schedulers (same configuration as figure 11a).

flow has two threads, and the request size is 8 KB. Hence, the bandwidth ratio of the four flows needs to be identical to the ratio of weights. As shown in the figure, D2FQ and MQFQ show correct bandwidth distribution ratio while BFQ shows a slightly poor bandwidth distribution. In the three workloads, the total bandwidth of BFQ is lower than the others. Please note that, in the above three experiments with D2FQ, the H/L ratio is 256 at the beginning and converges to 3, 3 and 6, respectively, by our dynamic H/L ratio adjustment.

Cost of Fairness. The cost of I/O scheduling is CPU cycles and D2FQ minimizes them by avoiding arbitration including request staging in the block layer. This effect results in the reduction in CPU cycles or CPU overhead for each I/O request handling. Figure 12 shows the CPU utilization and system jiffies per 1 KB I/O of the three workloads in Figure 11. As shown in the figure, BFQ shows the highest CPU utilization and also fails to fully utilize the SSD. MQFQ shows similar CPU utilization to BFQ due to its computation for request arbitration. D2FQ consume slightly more CPU cycles than None since D2FQ needs a few CPU cycles to select queue during dispatch and maintain scheduler statistics, such as virtual time. D2FQ reduces the CPU utilization by up to 45% as compared to MQFQ in Figure 12a. If the metric is CPU cycles per I/O (i.e., system-jiffies/KB), None and D2FQ show lower per-request CPU cost than MQFQ and BFQ. In summary, D2FQ provides fairness with minimal CPU cycles and the saved CPU cycles may be able to be used more usefully for applications.

Latency. Figure 13 shows the average I/O latency of the three workloads in Figure 11. With the fair I/O schedulers, the I/O latency is largely affected by I/O throttling; the throttled flows show longer I/O latency than the others. D2FQ and MQFQ show similar latency results. However, BFQ shows longer latency than the two schedulers. None shows the shortest latency in all the cases but fails to provide the fairness.

Short-term Fairness. While other fair queueing schedulers [9, 11, 13] have theoretical upper-bound in unfairness, D2FQ has none. This leads us to measure how much short-term unfairness occurs at least empirically. To this end, we measure the bandwidth of each flow of the request size experiment (Figure 11a) every short time interval (10 ms) and depicts the time-series bandwidth in Figure 14. Interestingly, our scheme shows very stable bandwidth distribution in both flows. This indicates that both flows almost evenly share the SSD bandwidth in every short time period. Meanwhile, MQFQ shows fluctuation of bandwidth in both flows; this phenomenon is due to frequent exchange of dispatch slots across the cores/sockets.



(a) Bandwidth of each flow for the second 10 seconds **(b)** Bandwidth of each flow for the third 10 seconds

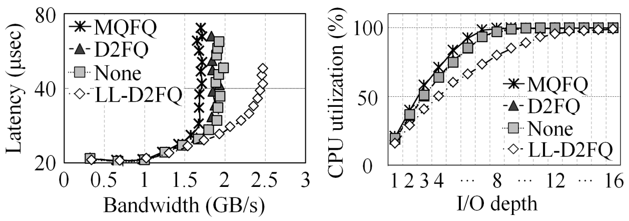
(c) Tail latency of the base flow **(d)** Tail latency of the flows in event2

(e) Dynamic H/L ratio change over time

Figure 15: Effect of the dynamic H/L ratio adjustment.

Dynamic H/L Ratio Adjustment. To test the effect of our dynamic H/L ratio adjustment, we build a synthetic workload with three flow groups with different lifetimes: (1) *base* that contains one weight-1 flow and runs from the beginning to the end, (2) *event1* that contains three weight-3 flows and runs from the 10-second point in time to the end, and (3) *event2* that contains four weight-3 flows and runs from the 20-second point to the end. Each flow has 2 threads and issues 4 KB I/O requests with 128 I/O queue depth. We run the workload with the three H/L ratio configurations: *static-3*, *static-128*, and *dynamic*, and depict the bandwidth, latency and H/L ratio change (dynamic only) in Figure 15.

Static-3 fails to provide the bandwidth fairness in Figure 15a and 15b; the bandwidth ratio between base and the rest should be 1:3. Static-128 and dynamic achieve the target bandwidth ratio because their H/L ratios are high enough to satisfy the fairness. However, as shown in Figure 15c and 15d, static-128 shows long tail latency because the H/L ratio of 128 is too high and flows using the low queues experience long I/O delays. Our dynamic H/L ratio adjustment adaptively sets the H/L ratio properly based on the I/O patterns of the flows. The H/L ratio changes over time as shown in Figure 15e; it is set to 7 when event1 begins and to 17–18 when event2 begins. Please note that the H/L ratio is not 22 as in Section 3.2.1 because the use of the medium queues gives additional fairness control over the low queues.



(a) Bandwidth and latency **(b)** Single-core utilization

Figure 16: Latency, bandwidth and CPU (single core) utilization with a single-thread flow with varying I/O depths.

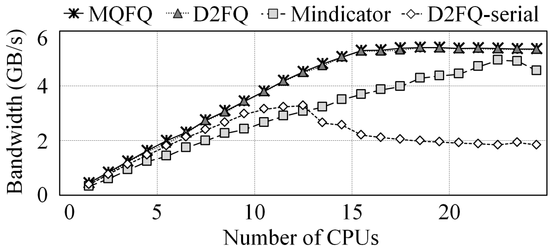


Figure 17: Scalability test of the global virtual time tracking schemes.

4.3 I/O Performance

The I/O scheduler performs not only when the storage device is saturated but also when the device is unsaturated. When the device is unsaturated, the performance of I/O scheduler is the major factor to the I/O performance. In this regard, we measure the I/O performance under low I/O contention.

In this experiment, we add another scheme to demonstrate that D2FQ is independent of the multi-queue block layer in Linux. A low-latency I/O stack [19] achieves low I/O latency by its submit=dispatch characteristic. Unfortunately, it lacks I/O scheduling support, which can be complemented by D2FQ; *LL-D2FQ* is the low-latency I/O stack with D2FQ, thereby having the I/O scheduling capability.

Figure 16 shows the latency, bandwidth and CPU utilization of each I/O scheduler. For the workload, we run a single-thread FIO performing 8 KB random read with varying its I/O depth from 1 to 16.

Basically, the increase in I/O depth results in the increase of bandwidth as well as latency in all the schedulers as shown in Figure 16a. Unless the CPU is saturated, increasing the I/O depth increases the CPU utilization due to handling more I/O requests as shown in Figure 16b.

More importantly, the overhead of I/O scheduler significantly affects the I/O latency, I/O bandwidth and the CPU utilization. As shown in the figure, MQFQ shows the lowest performance in terms of the three metrics. D2FQ and None show similar position in performance since they exclude the I/O scheduler in the block layer. Finally, LL-D2FQ outperforms the others by up to 35% in latency and 54% in bandwidth due to its low overhead in I/O request handling. It delays the CPU saturation point to the I/O depth of 14 whereas the other schedulers saturate the CPU at the I/O depth of 9 in Figure 16b.

Scalability. We test the scalability of D2FQ with varying the *gvt* tracking methodology. *None* performs no *gvt* tracking. *D2FQ* uses our *gvt* tracking method in Section 3.4. *Mindicator* tracks *gvt* using Mindicator [22] as in MQFQ [11]. *D2FQ-serial* tracks *gvt* by inspecting all flows every time of accessing *gvt*.

As shown in Figure 17, with increasing the number of cores (i.e., flows), D2FQ-serial does not scale after 12 cores due to cross-socket communication [6]. Mindicator scales well, but its tree-based data management incurs overheads with high core counts. D2FQ shows identical scaling to None.

4.4 Realistic workload

Finally, we measure the impact of D2FQ on realistic workloads. We run two flows: one flow of YCSB workloads [37] on the RocksDB and the other contending flow of the FIO benchmark. Since we run a realistic workload, YCSB saturates CPU first in Machine A. So, we use Machine B in this experiment. This machine has another difficulty in queue assignment since 24 cores need 72 command queues (three queues for each core), but Z-SSD provides only 32 command queues. To resolve this issue, we group three queues of three adjacent cores and make the three cores share the three queues as the three-class queues in this experiment.

The FIO workload issues 4 KB random read using 4 threads with 128 I/O depth. In the YCSB workload performs 64 million operations on 64 GB key-value dataset with 1 KB value size. The physical memory is reduced to 16 GB.

Figure 18 shows the bandwidth and CPU utilization of the workload with the four schedulers. As shown in the figure, None cannot fairly distribute the SSD bandwidth to the two flows. YCSB consumes lower bandwidth than FIO since FIO is more bandwidth hungry. With fair queueing schedulers, the bandwidth is fairly distributed across the two flows; D2FQ shows 1.00 – 1.05 bandwidth ratio (YCSB bandwidth over FIO bandwidth) and MQFQ shows 1.00 – 1.08 bandwidth ratio. None shows lower CPU utilization than D2FQ and MQFQ because YCSB, which consumes more CPU cycles per I/O than FIO, takes a smaller portion in total bandwidth than D2FQ and MQFQ.

D2FQ shows a slightly higher total bandwidth than MQFQ by up to 1.83% on average. This is due to the true work-conserving characteristic of D2FQ; all submitted requests are dispatched to the device queues. MQFQ rarely fails to maximally utilize the device bandwidth due to the exchange of request dispatch slots between cores and sockets.

Figure 19 shows the average I/O latency of each workload normalized to the result of None. The YCSB workload reports the latency and number of operations for each operation type, so we calculate the weighted average latency in that case. As shown in the figure, basically flows showing higher I/O bandwidth achieve shorter I/O latency. With None, FIO

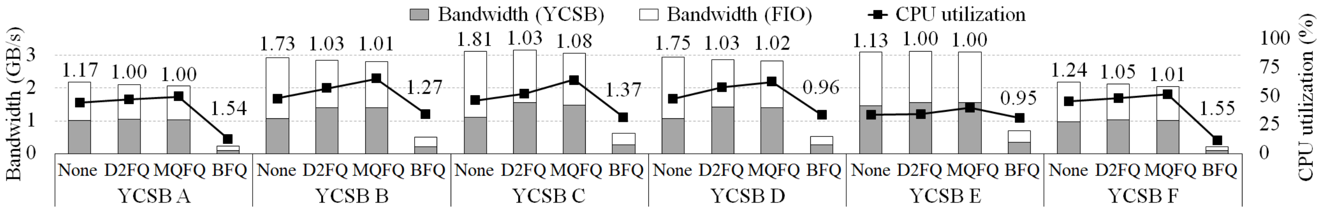


Figure 18: I/O bandwidth and CPU utilization of the realistic workload (YCSB with FIO). The number above each histogram shows the bandwidth ratio (the FIO bandwidth over the YCSB bandwidth).

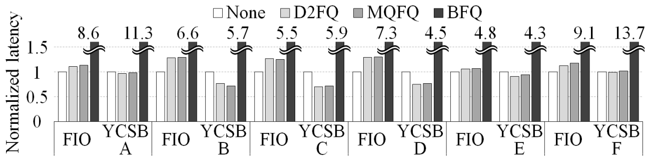


Figure 19: Normalized latency of the realistic workload (YCSB with FIO).

achieves high I/O bandwidth and low I/O latency whereas YCSB shows low I/O bandwidth and high I/O latency. On the contrary, D2FQ and MQFQ show smaller FIO bandwidths and larger YCSB bandwidths than None. Accordingly, both show longer I/O latency with FIO and shorter I/O latency with YCSB. BFQ shows very long I/O latency due to its inefficiency in I/O scheduling.

5 Related Work

Fair-share I/O Schedulers. Fair resource sharing is one of important goals of I/O resource sharing. Linux employs time slice-based fair schedulers, such as CFQ [3], BFQ [34], Argon [35] and FIOS [30]. Time slice-based schedulers are non-work conserving: I/O resources can remain unused while requests are available. Ahn et al. [2] proposed a budget-based fair share I/O scheduler implemented in Linux cgroup layer; it is also a non-work conserving scheduler. H-BFQ [28] has expanded the original BFQ to hierarchical cgroup structure. Fair queueing I/O schedulers [8, 9, 29] including D2FQ are work-conserving so they always try to keep I/O resource busy whenever requests exist. Fair queueing schedulers provide fairness using virtual time [8, 9, 29], and they controls the order of I/O requests to minimize the virtual time gap between any flows. With the advance in storage performance, it is necessary to dispatch multiple requests to a device to maximize the I/O performance. This relaxes the requirement of minimizing the virtual time gap between any flows. SFQ(D) [13] allows at most D outstanding requests. MQFQ [11] relaxes the requirement further to enable parallel dispatch with a little communication across cores. D2FQ also relaxes the assumption which is determined by the two thresholds (τ_m and τ_l) for a different reason: too small thresholds increases the tail latency by unnecessarily throttling requests.

Other I/O Schedulers. Lee et al. [20] isolate queues to prioritize reads over writes. Kyber [18] prioritize synchronous I/Os over asynchronous ones to foreground performance. Kim et al. [16] prioritize requests from foreground context holistically throughout the I/O stack. These schedulers, however, do not provide fair I/O resource management.

Scheduling Offloading to Device. FLIN [33] implements fair-share scheduler in the SSD firmware and identifies and considers the major sources of performance interference in a flash-based SSD. Joshi et al. [14] enlightens the use of NVMe WRR in Linux for SSD resource fairness. The use of NVMe WRR to the block cgroup is later implemented in the mainline Linux [27]. None of these work consider the sharing of queues with multiple flows, which is necessary when the number of flows exceeds the number of queues.

6 Conclusion

This paper proposes D2FQ, a low overhead high-performance fair-queueing I/O scheduler. D2FQ is carefully designed to implement the sophisticated fair-queueing I/O scheduler on top of the simple device-side I/O scheduling feature (i.e., NVMe WRR). Therefore, the CPU overhead associated with scheduling decision is minimized, thereby saving CPU cycles and improving I/O performance. Modern high-performance SSDs are changing the paradigm that the bottleneck is no longer I/O but CPU. We expect our light-weight fair-queueing scheme will help reduce the contention on the CPU and allow applications to use more CPU cycles for a useful way. We plan to extend our scheme to leverage the urgent priority class of the NVMe WRR for better quality of service of block I/O scheduling.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Janki Bhimani, for their valuable comments. This work was supported partly by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2020R1A2C2102406), by the MSIT (Ministry of Science and ICT), Korea, under the ICT Creative Consilience program (IITP-2020-0-01821) and by Samsung Electronics.

Availability

The source code is available at https://github.com/skkucsl/d2fq

References

[1] Samsung 980 Pro. https://news.samsung.com/global/samsung-delivers-next-level-ssd-performance-with-980-pro-for-gaming-and-high-end-pc-applications.

[2] Sungyong Ahn, Kwanghyun La, and Jihong Kim. Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-Core Systems. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'16, page 111–115, USA, 2016. USENIX Association.

[3] Jens Axboe. Linux block IO—present and future. In *Ottawa Linux Symp*, pages 51–61, 2004.

[4] Budget Fair Queueing. https://lwn.net/Articles/784267/, 2019.

[5] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, New York, NY, USA, 2013. Association for Computing Machinery.

[6] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 33–48, New York, NY, USA, 2013. Association for Computing Machinery.

[7] Data center bridging task group. http://www.ieee802.org/1/pages/dcbridges.html.

[8] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, August 1989.

[9] Pawan Goyal, Harrick M. Vin, and Haichen Chen. Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. *SIGCOMM Comput. Commun. Rev.*, 26(4):157–168, August 1996.

[10] Jun He, Kan Wu, Sudarsun Kannan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Read as Needed: Building WiSER, a Flash-Optimized Search Engine. In *18th USENIX Conference on File and Storage Technologies*, FAST'20, pages 59–73, Santa Clara, CA, February 2020. USENIX Association.

[11] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. Multi-Queue Fair Queueing. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 301–314, USA, 2019. USENIX Association.

[12] Sitaram Iyer and Peter Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 117–130, New York, NY, USA, 2001. Association for Computing Machinery.

[13] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed Proportional Sharing for a Storage Service Utility. *SIGMETRICS Perform. Eval. Rev.*, 32(1):37–48, June 2004.

[14] Kanchan Joshi, Praval Choudhary, and Kaushal Yadav. Enabling NVMe WRR Support in Linux Block Layer. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'17, page 22, USA, 2017. USENIX Association.

[15] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T. Kandemir. HIOS: A Host Interface I/O Scheduler for Solid State Disks. *SIGARCH Comput. Archit. News*, 42(3):289–300, June 2014.

[16] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, page 345–358, USA, 2017. USENIX Association.

[17] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the Performance of Fast NVM Storage with Udepot. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 1–15, USA, 2019. USENIX Association.

[18] Kyber multi-queue i/o scheduler. https://lwn.net/Articles/720071/, 2017.

[19] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O Stack: A Low-Latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 603–616, USA, 2019. USENIX Association.

[20] Minkyeong Lee, Dong Hyun Kang, Minho Lee, and Young Ik Eom. Improving Read Performance by Isolating Multiple Queues in NVMe SSDs. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, IMCOM '17, New York, NY, USA, 2017. Association for Computing Machinery.

[21] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*,

SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Yujie Liu, Victor Luchangco, and Michael Spear. Mindicators: A Scalable Approach to Quiescence. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, page 206–215, USA, 2013. IEEE Computer Society.

[23] MELLANOX TECHNOLOGIES. ConnectX-4 VPI. https://www.mellanox.com/files/doc-2020/pb-connectx-4-vpi-card.pdf.

[24] Mindicator. https://github.com/mfs409/nonblocking/tree/master/tsx_acceleration/mindicator.

[25] Multi-queue deadline I/O Scheduler. https://lwn.net/Articles/767987/, 2016.

[26] NVM express specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4a-2020.03.09-Ratified.pdf.

[27] Add support Weighted Round Robin for blkcg and nvme. https://lwn.net/Articles/810726/.

[28] Kwonje Oh, Jonggyu Park, and Young Ik Eom. H-BFQ: Supporting Multi-Level Hierarchical Cgroup in BFQ Scheduler. In *2020 IEEE International Conference on Big Data and Smart Computing (BigComp 20)*, pages 366–369. IEEE, 2020.

[29] Abhay Kumar Parekh. *A generalized processor sharing approach to flow control in integrated services networks*. PhD thesis, Massachusetts Institute of Technology, 1992.

[30] Stan Park and Kai Shen. FIOS: A Fair, Efficient Flash I/O Scheduler. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 13, USA, 2012. USENIX Association.

[31] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for End-Host Rate Limiting. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, page 475–488, USA, 2014. USENIX Association.

[32] Brent Stephens, Aditya Akella, and Michael M. Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 33–46, USA, 2019. USENIX Association.

[33] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 397–410. IEEE Press, 2018.

[34] Paolo Valente and Arianna Avanzini. Evolution of the BFQ Storage-I/O scheduler. In *2015 Mobile Systems Technologies Workshop*, pages 15–20. IEEE, 2015.

[35] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, page 5, USA, 2007. USENIX Association.

[36] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[37] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB.

[38] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. Flashshare: Punching through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 477–492, USA, 2018. USENIX Association.

An In-Depth Study of Correlated Failures in Production SSD-Based Data Centers

Shujie Han[1], Patrick P. C. Lee[1], Fan Xu[2], Yi Liu[2], Cheng He[2], and Jiongzhou Liu[2]
[1]The Chinese University of Hong Kong [2]Alibaba Group

Abstract

Flash-based solid-state drives (SSDs) are increasingly adopted as the mainstream storage media in modern data centers. However, little is known about how SSD failures in the field are correlated, both spatially and temporally. We argue that characterizing correlated failures of SSDs is critical, especially for guiding the design of redundancy protection for high storage reliability. We present an in-depth data-driven analysis on the correlated failures in the SSD-based data centers at Alibaba. We study nearly one million SSDs of 11 drive models based on a dataset of SMART logs, trouble tickets, physical locations, and applications. We show that correlated failures in the same node or rack are common, and study the possible impacting factors on those correlated failures. We also evaluate via trace-driven simulation how various redundancy schemes affect the storage reliability under correlated failures. To this end, we report 15 findings. Our dataset and source code are now released for public use.

1 Introduction

Maintaining high storage reliability is undoubtedly important for modern data centers, yet it is often challenged by *correlated failures*, such as bursts of latent sector errors [25], correlated disk failures [5, 26], co-occurring node failures [5, 8, 11, 19], or correlated crashes of data and protocols [1]. Correlated failures complicate the design of redundancy protection schemes, which may be sufficient for tolerating independent failures but not correlated failures [19].

Modern data centers now increasingly build on flash-based solid-state drives (SSDs), and their storage reliability guarantees critically depend on the reliability of SSDs. Several field studies have characterized SSD failures in production environments, including Facebook [16], Google [2, 27], Microsoft [18], Alibaba [30], and NetApp [15] (see §6 for details). However, some of the studies [2, 15, 16, 27] analyze the proprietary customized attributes that are inapplicable for general production environments; others [18, 30] leverage the SMART (Self-Monitoring, Analysis and Reporting Technology) attributes that are known to provide statistical details for disk drive failure symptoms, yet SMART attributes do not provide the location details of how multiple failures manifest across storage *scopes* (e.g., nodes and racks). Although correlated failures are reportedly found in SSD-based data centers [15, 16], little is known about the characteristics of correlated failures and their implications on storage reliability in production environments.

To elaborate, the following questions on correlated failures remain unexplored: (i) How far are SSD failures spaced apart across different scopes in large-scale data centers? (ii) How likely does an SSD fail after another failure occurs in the same scope? (iii) How long is the time interval between two consecutive SSD failures in the same scope? (iv) Do SSD failures that are close in space imply that they are also close in time? (v) What are the factors that affect the correlated failures? (vi) What should be the proper redundancy protection schemes in the presence of correlated failures? The answers to these questions can provide insights into achieving high storage reliability in production environments.

In this paper, we present an in-depth data-driven analysis on the correlated failures, from both spatial and temporal perspectives, of SSD-based data centers at Alibaba, one of the largest Internet companies in the world. We present an extensive study on the correlated failures of nearly one million SSDs, belonging to 11 drive models from three vendors, over a span of two years. Our dataset covers the SMART logs, trouble tickets, physical locations of SSDs (e.g., nodes and racks), and the applications hosted by the underlying SSDs. Our analysis makes the following findings:

- We characterize two main types of correlated failures in the same node and rack that occur within a short time (e.g., 30 minutes), referred to as *intra-node failures* and *intra-rack failures*, respectively. We observe a non-negligible fraction of intra-node and intra-rack failures, implying the existence of strong spatial and temporal correlations of SSD failures.
- We analyze four impacting factors of drive characteristics on the correlated failures: drive models, lithography, age, and capacity. We show that such factors pose different effects on the spatial and temporal correlations of SSD failures. In particular, intra-node (intra-rack) failures likely occur in the nodes (racks) that are attached by many SSDs of the same drive model. Both intra-node and intra-rack failures of aged SSDs tend to occur within a short time.
- We analyze the impact of SMART attributes and applications on both intra-node and intra-rack failures. We find that SMART attributes have limited correlations with both intra-node and intra-rack failures and are not good indicators for detecting the existence of intra-node and intra-rack failures. Also, write-dominant applications lead to more intra-node and intra-rack failures than read-dominant ones.
- We conduct trace-driven simulation using our dataset on the impact of different redundancy protection schemes on

storage reliability. We show that redundancy schemes with high fault tolerance are critical to storage reliability under correlated failures.

We release our dataset, including the SMART logs of all failed SSDs, trouble tickets, locations, and applications, for the 11 drive models at **https://github.com/alibaba-edu/dcbrain/tree/master/ssd_open_data**. The community can leverage our dataset and findings to design effective reliability solutions in production environments. We also open-source our analysis scripts and simulator prototype at **http://adslab.cse.cuhk.edu.hk/software/ssdanalysis**.

2 Dataset

In this section, we introduce the dataset for our analysis. We describe our data collection methodology (§2.1) and study the drive population and characteristics of our dataset (§2.2). We also discuss the limitations of our dataset (§2.3).

2.1 Data Collection

We collected data from multiple SSD-based data centers at Alibaba. Each data center comprises multiple *racks*, each of which holds multiple machines called *nodes*. Each node is further attached with one or multiple SSDs.

Our dataset spans two years from January 2018 to December 2019. It covers a population of nearly one million SSDs of 11 drive models from three vendors. The SSDs are deployed in 200 K nodes of 30 K racks. Note that the SSDs of the same drive model were typically purchased from multiple batches at different times, and the SSDs attached to each node may be heterogeneous in terms of vendors, models, capacities, and deployment times. However, among the nodes with at least two SSDs, 88.6% of them are attached to the SSDs of the same drive model.

Our dataset includes multiple data types: SMART logs, trouble tickets, locations, and applications.

SMART logs. SMART is a widely adopted tool for monitoring disk drive status. It periodically reports the numerical values of the performance and reliability statistics on different dimensions, called *attributes*. Each SMART attribute includes both the raw and normalized values. Our dataset contains daily collected SMART logs over the two-year span, and its collected SMART attributes are summarized in Table 1. Since the definitions of SMART attributes vary across vendors, for easy comparison, we focus on the SMART attributes that are reported by more than half of SSDs (shown in the "Reported%" column). We categorize the SMART attributes by their monitoring types into five groups, namely internal errors, spare blocks, wearout degree, workload, and power. Some SMART attributes have identical meanings but are assigned different SMART IDs by vendors (e.g., S170/S180, S171/S181, and S172/S182). Also, some SMART attributes have vendor-specific raw values (marked with an asterisk '*' in Table 1), so we only consider their normalized values.

Category	**ID**	**Attribute name**	**Reported%**
Internal errors	S5	Reallocated sector count	100.0%
	S183	SATA downshift error count	96.5%
	S184	End-to-end errors	100.0%
	S187	Reported uncorrectable errors	100.0%
	S195	Hardware ECC recovered	55.4%
	S197	Current pending sector count	87.5%
	S199	UltraDMA CRC error count	100.0%
	S171/S181	Program failed count	100.0%
	S172/S182	Erase failed count	100.0%
Spare blocks*	S170/S180	Available reserved blocks	100.0%
Wearout degree*	S173	Wear leveling count	100.0%
	S177	Wear range delta	
	S233	Media wearout indicator	
Workload	S241	Number of blocks written	68.8%
	S242	Number of blocks read	56.3%
Power	S9	Power on hours	100.0%
	S12	Power cycle count	99.1%
	S174	Unexpected power loss count	78.5%
	S175*	Power loss protection failure	57.0%

Table 1: Overview of SMART attributes in our dataset. "Reported%" is the percentage of SSDs with the corresponding SMART attribute. Only the normalized values are considered for the vendor-specific SMART attributes marked by an asterisk '*'.

Trouble tickets. Each node runs a background monitoring daemon that periodically collects SMART statistics and system-level logs/alerts from its attached SSDs and sends the collected data to a centralized maintenance system that monitors failures. The maintenance system applies rule-based detection, defined by administrators, to detect and report any failure behavior in the form of *trouble tickets*. Each trouble ticket records the node ID, drive ID, timestamp, and failure description. Administrators further manually validate each trouble ticket to confirm the failure status. We use the trouble tickets as the ground-truths for our failure analysis. Throughout the two-year span, we collected about 19 K trouble tickets (i.e., 19 K failed SSDs in total).

Our trouble tickets cover two main types of SSD failures: (i) *whole drive failures*, in which an SSD either cannot be accessed or loses all data that is unrecoverable; and (ii) *partial drive failure*, in which part of the data in an SSD either cannot be accessed and is unrecoverable.

Locations. Our dataset records the physical location of each SSD, including the machine room ID, rack ID, node ID, drive ID, and slot number. In particular, we can correlate an SSD to the SMART logs and trouble tickets by its drive ID.

Applications. Our dataset covers hundreds of applications, including both internal (e.g., resource management, development, testing, etc.) and external services (e.g., web services, data analytics, etc.). Each node is configured to serve a single application (note that the applications within a rack may be different) and distributes a set of tasks to the attached SSDs as evenly as possible. We can correlate an SSD to its hosted

Applications	Total%	Failures%
Web service management (WSM)	39.4%	48.5%
Resource management (RM)	19.1%	16.4%
Web proxy services (WPS)	4.6%	2.9%
SQL services (SS)	3.4%	1.0%
Database (DB)	2.8%	1.1%
Web services (WS)	1.8%	1.3%
Data analytics engine (DAE)	1.7%	6.6%
Network attached storage (NAS)	1.5%	2.9%

Table 2: Overview of the top eight most widely used applications with more than hundreds of failed SSDs, including the percentage of deployed SSDs in the whole population ("Total%") and the percentage of SSD failures in the failed SSD population ("Failures%"). Note that SS and DB are two similar applications, but belong to different business units.

application via its node ID. Table 2 shows the top eight most widely used applications, each of which contains hundreds of failed SSDs in our dataset. Specifically, WSM covers 39.4% of all SSDs and 48.5% of all failed SSDs. WPS, SS, and DB cover 10.8% of all SSDs, while covering only 5.0% of all failed SSDs. DAE and NAS have 3.2% of all SSDs, while covering 9.5% of all failed SSDs. We will give a detailed analysis on the relationships between the failure patterns and workload distributions of the eight applications (§4.4).

2.2 Summary of Statistics

We first analyze the basic statistics and SSD characteristics in our dataset, as shown in Table 3.

Population statistics. We consider 11 drive models from three vendors. Each drive model is denoted by "Vendor""*k*", where "Vendor" is represented by a letter ('A', 'B', and 'C') for each of the three vendors, and "*k*" (1 to 6) refers to the *k*-th most numerous model in the same vendor. The first three columns in Table 3 show the percentages of each drive model in the same vendor and the whole population. The 11 drive models together cover nearly one million SSDs.

Drive characteristics. The fourth to sixth columns in Table 3 describe the key drive characteristics, including the flash technology, lithography, and capacity. All 11 drive models use the SATA interface. The drive models in vendors A and B build on enterprise-class MLC NAND cells, while those in vendor C build on 3D-TLC flash. These drive models have different lithography parameters (bill-of-material (BOM) revision for 3D-TLC) and capacities (ranging from 240 to 1920 GB).

Usage. The seventh to ninth columns in Table 3 show the statistical summaries of SSD usage, including the over-provisioning (OP) factor (i.e., the fraction of dedicatedly reserved space in SSDs for internal garbage collection), the average power-on years computed from S9 (Table 1), and the mean of rated life used (i.e., the percentage of erase cycles over the erase cycle limit) computed from the SMART attributes related to the wearout degree (Table 1).

Reliability. The last three columns in Table 3 show three reliability metrics, including the mean percentage of spare blocks used, the mean number of bad sectors, and the annualized failure rate (AFR). We compute the percentage of spare blocks using the SMART attributes related to spare blocks (S170/S180), and the number of bad sectors using S5 in Table 1. We define the AFR by the following formula [13, 18]:

$$AFR(\%) = \frac{f}{n_1 + n_2 + \ldots + n_{two-year}} \times 365 \times 100,$$

where f is the total number of failed SSDs reported in our trouble tickets and n_i is the number of operational SSDs on day i over the two-year span. The overall AFR of all MLC SSDs (A1 to A6 and B1 to B3) is 0.55%, and their AFRs range from 0.16% to 2.52%, slightly lower than those reported for SSDs in Google's data centers (1-2.5%) [27]. In contrast, the AFRs of 3D-TLC SSDs (C1 and C2) are higher than 3%. The overall AFR of all SSDs in our dataset is 1.16%.

2.3 Limitations

Our analysis has the following limitations, mainly due to the unavailable information in our dataset.

Data missing. We expect that the SMART logs contain daily statistics without loss, yet our dataset indeed contains incomplete SMART data over time in both failed and healthy SSDs. Reasons of such data missing include network failures, software maintenance or upgrades, system crashes, etc. In this work, we mainly focus on analyzing the correlations of SSD failures via trouble tickets, rather than the correlations of SMART attributes over time. Thus, the data missing in the SMART logs does not compromise our analysis.

Failure symptoms. Our dataset reports SSD failures via trouble tickets, but does not cover the failure symptoms at the operating system level. Such failure symptoms can be found in kernel syslogs, which are not collected in our dataset.

Drive repair. Our dataset does not include the repair details for failed SSDs. In practice, how long the data in a failed SSD is recovered depends on the importance of its stored data to the upper-level applications. Administrators may not immediately repair the failed SSDs that store less critical data to save operational overhead [2, 30]. Due to limited details, we assume that all SSDs store data with the same importance, and the repair time depends on the amount of data to be reconstructed (§5).

Redundancy protection. Production storage systems use erasure coding for redundancy protection against failures [8, 12, 17]. In Alibaba production, 3-way replication is the commonly used redundancy mechanism [30]. However, the redundancy parameters may also vary across applications and we do not have access to the redundancy parameters for each application. In this work, we assume that all applications adopt the same redundancy parameters to drive our reliability analysis (§5).

Population statistics			Drive characteristics			Usage			Reliability		
Model	Vendor%	Total%	Flash Tech.	Lithography	Capacity	OP	Power-on years	Rated life used (%)	Spare blocks used (%)	# bad sectors	AFR (%)
A1	52.3%	29.8%	MLC	20 nm	480 GB	7%	4.6	17.8 (±0.067)	0.18 (±0.0080)	9.3 (±0.60)	0.16%
A2	21.8%	12.4%	MLC	20 nm	800 GB	28%	4.5	17.2 (± 0.15)	0.19 (±0.013)	12.5 (±1.3)	0.46%
A3	7.9%	4.5%	MLC	20 nm	480 GB	7%	5.5	25.9 (±0.41)	0.022 (±0.012)	12.4 (±2.4)	2.36%
A4	7.2%	4.1%	MLC	16 nm	240 GB	7%	3.2	8.8 (±0.074)	0.064 (±0.013)	2.4 (±0.72)	0.64%
A5	5.7%	3.3%	MLC	16 nm	480 GB	7%	3.2	27.0 (±0.28)	0.087 (±0.015)	5.0 (±1.2)	0.45%
A6	5.1%	2.9%	MLC	20 nm	800 GB	28%	4.6	24.7 (±0.44)	0.018 (±0.013)	13.7 (±2.9)	0.49%
B1	51.5%	10.3%	MLC	21 nm	480 GB	7%	3.8	6.4 (±0.029)	0.0063 (±0.0010)	0.036 (±0.024)	0.21%
B2	25.5%	5.1%	MLC	19 nm	1920 GB	7%	3.3	2.0 (±0.014)	0.086 (±0.0092)	12.2 (±1.4)	0.71%
B3	23.0%	4.6%	MLC	24 nm	1920 GB	7%	2.1	3.6 (±0.028)	0.021 (±0.0041)	0.50 (±0.25)	2.52%
C1	89.3%	20.6%	3D-TLC	V1	1920 GB	7%	2.0	4.3 (±0.022)	0.064 (±0.0054)	10.1 (±0.74)	3.29%
C2	10.7%	2.5%	3D-TLC	V1	960 GB	7%	1.4	2.0 (±0.062)	0.0049 (±0.0047)	0.67 (±0.37)	3.92%

Table 3: Summary of statistics of collected dataset. The population statistics include the percentage of drives in the same vendor ("Vendor%") and the percentage of drives in the whole drive population in the dataset ("Total%"). For the "Rated life used", "Spare blocks used", and "# bad sectors" columns, each value in brackets denotes the 95% confidence interval.

3 Overview of Analysis Methodology

Our analysis studies the correlated failures of SSDs in our dataset, and focuses on several dimensions.

Spatial and temporal properties. We study how SSD failures manifest within a scope, either a node or a rack, within a certain time period. We consider both *intra-node failures* and *intra-rack failures* to refer to the failures co-occurring within a node and a rack, respectively. We also define the *intra-node (intra-rack) failure time interval* as the time interval between two consecutive failures that co-occur within the same node (rack). We refer to a failure as an intra-node (intra-rack) failure if its intra-node (intra-rack) failure time interval with its preceding or following failure in the same node (rack) is smaller than a pre-specified threshold. Here, we set a default threshold as 30 minutes, assuming that this is the minimum time for a failure to be detected before it is repaired [12]. In other words, a node (rack) may contain more than one active failure at a time under intra-node (intra-rack) failures. We define the *intra-node (intra-rack) failure group* as a sequence of intra-node (intra-rack) failures starting from an intra-node (intra-rack) failure without a preceding one until an intra-node (intra-rack) failure without a following one. We also vary the thresholds of the intra-node and intra-rack failure time interval in our analysis.

Correlation properties. We use the Spearman's Rank Correlation Coefficient (SRCC) [29] to measure the correlation of two variables. For example, to measure the correlation between an SSD failure and a SMART attribute using the SRCC, we use an indicator variable to represent if an SSD is failed (i.e., 1 means failed; or 0 otherwise), and a numerical variable to represent the value of a SMART attribute. The SRCC calculates the Pearson Correlation Coefficient [21] between the rank values of two variables to measure their monotonic relationships. The SRCC ranges from -1 (i.e., high negative correlation) to +1 (i.e., high positive correlation); a zero SRCC means that the two variables are independent.

4 Correlation Analysis

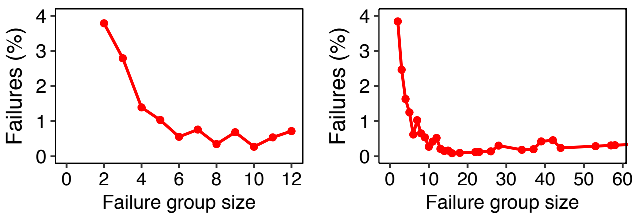
We analyze the correlated failures of SSDs in our dataset in four aspects: (i) spatial and temporal correlations among failures (§4.1), (ii) the impacting factors on correlated failures, including the drive models, lithography, age, and capacity (§4.2), (iii) the impact of SMART attributes on correlated failures (§4.3), and (iv) the impact of applications on correlated failures (§4.4). Finally, we discuss the implications of our findings (§4.5).

4.1 Correlations among Failures

We first examine the severity of correlated failures by the intra-node and intra-rack failure group sizes (i.e., by counting the number of failures within a group). Figure 1 shows the percentage of failures versus the intra-node or intra-rack failure group sizes; note that for Figure 1(b), we omit the plots for the intra-rack failure group sizes that exceed 60 (the maximum is 89) due to the sparseness. We see that a non-negligible fraction of SSD failures belong to intra-node and intra-rack failures. In particular, 12.9% (18.3%) of failures are intra-node (intra-rack) failures. Also, the intra-node and intra-rack failure group size can exceed the tolerable limit of some typical redundancy protection schemes (e.g., four failures) (see §5 for details).

Finding 1. *A non-negligible fraction of SSD failures belong to intra-node and intra-rack failures (12.9% and 18.3% in our dataset, respectively). Also, the intra-node and intra-rack failure group size can exceed the tolerable limit of some typical redundancy protection schemes.*

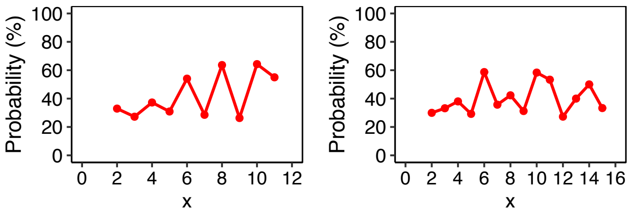
We further check whether the likelihood of an SSD failure depends on the already existing SSD failures among the intra-node and intra-rack failures. Borrowing the idea by Mesa et al. [16], we compute the conditional probability of having an additional SSD failure per intra-node (intra-rack) failure group given the existing intra-node (intra-rack) failures, by dividing the number of intra-node (intra-rack) failure groups



(a) Intra-node failures

(b) Intra-rack failures

Figure 1: Finding 1. Percentages of failures for different intra-node and intra-rack failure group sizes.



(a) Intra-node failures

(b) Intra-rack failures

Figure 2: Finding 2. Conditional probabilities of having an additional SSD failure for different failure group sizes per intra-node or intra-rack failure group.

with a failure group size of $x+1$ by the number of intra-node (intra-rack) failure groups with a failure group size of x or $x+1$.

Figure 2 shows that the conditional probability of an additional SSD failure depends on the already existing SSD failures among the intra-node and intra-rack failures. The conditional probability of having an additional SSD failure in an intra-node (intra-rack) failure group ranges from 26.3% to 64.3% as x ranges from 2 to 11 (from 27.3% to 58.7% as x ranges from 2 to 88); note that we omit the plots for the intra-rack failure group size that exceeds 16 in Figure 2(b) due to the sparseness. If there is no correlation among intra-node (intra-rack) failures and the SSD failures are uniformly distributed on nodes (racks), the conditional probability of having an additional SSD failure given the existing intra-node (intra-rack) failures is similar to the AFR [16].

Finding 2. *The likelihood of having an additional intra-node (intra-rack) failure in an intra-node (intra-rack) failure group depends on the already existing intra-node (intra-rack) failures.*

We examine how the percentages of intra-node and intra-rack failures are affected by various thresholds of the intra-node and intra-rack failure time intervals, respectively. Figure 3 shows that a non-negligible fraction of intra-node and intra-rack failures occur within a short period of time. The intra-node (intra-rack) failures with one month as the threshold of the failure time interval account for 29.2% (63.0%). When the threshold of the failure time interval falls in one minute, the intra-node (intra-rack) failures still account for 10.0% (14.4%).

Finding 3. *A non-negligible fraction of intra-node and intra-rack failures occur within a short period of time, even within one minute.*

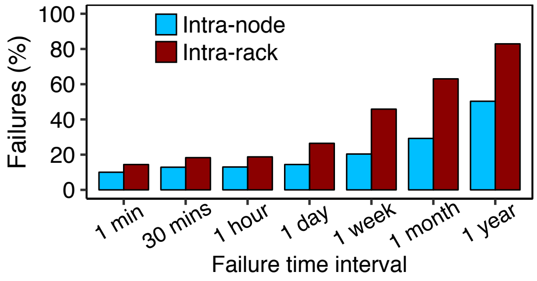


Figure 3: Finding 3. Percentages of intra-node (intra-rack) failures broken down by different thresholds of the intra-node (intra-rack) failure time intervals.

4.2 Impacting Factors on Correlated Failures

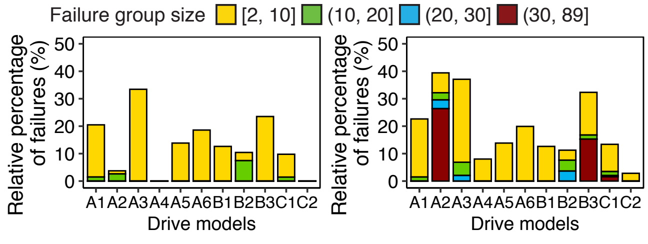
We next study how various factors affect the spatial and temporal correlations of failures.

4.2.1 Drive Models

We analyze the impact of drive models on correlated failures. Figure 4 shows that the relative percentages of failures (over all SSD failures of the same drive model) for different sets of intra-node and intra-rack failure group sizes vary highly across the drive models. In particular, the relative percentages of intra-node (intra-rack) failures range from 0% to 33.4% (from 2.8% to 39.4%). Interestingly, A2 has only 3.7% of intra-node failures, but has 39.4% of intra-rack failures, among which 26.4% reside in the intra-rack failure groups of sizes larger than 30.

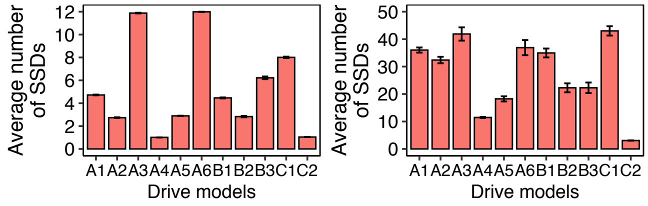
We next examine the reason of high percentages of intra-node and intra-rack failures of some drive models, by examining the average numbers of SSDs per node or rack for different drive models. Figure 5 shows the distribution of the average number of SSDs per node and rack (each error bar shows the 95% confidence interval). In general, putting more SSDs from the same drive model in the same nodes (racks) leads to a higher percentage of intra-node (intra-rack) failures.

However, we observe some exceptions. A3 and A6 have the same average number of SSDs per node (i.e., 12.0), but the relative percentage of intra-node failures for A3 is higher than that for A6 by 14.8%. One possible reason is that the AFR of A3 (2.36%) is 5× that of A6 (0.49%). Note that the AFR is not always the root cause of leading to high relative percentages of intra-node and intra-rack failures. For example, one exception is that C1 has more average number of SSDs per node (rack) than B3 by 1.8 (20.7), but the relative percentage of intra-node (intra-rack) failures for C1 is lower than that for B3 by 13.7% (18.9%). Similar exceptions include the intra-rack failures for A2 and A3. However, the AFR of B3 (A2) is lower than that of C1 (A3) by 0.77% (1.9%). We further examine the machine rooms where intra-rack failures reside for A2 and B3. We observe that the relative percentages of intra-rack failures



(a) Intra-node failures (b) Intra-rack failures

Figure 4: Finding 4. Relative percentages of failures for different sets of intra-node or intra-rack failure group sizes across the drive models.



(a) Per node (b) Per rack

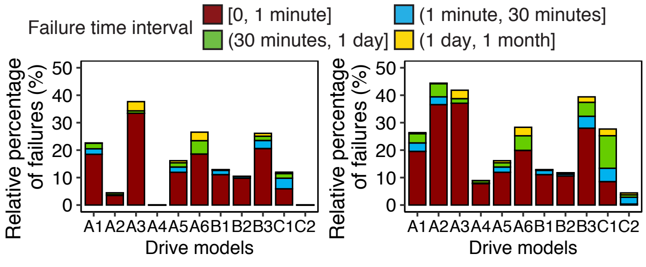
Figure 5: Finding 4. Average numbers of SSDs per node or rack for the drive models (with 95% confidence intervals as error bars).

from two machine rooms account for 29.6% and 15.3% for A2 and B3, respectively, and the intra-rack failure group sizes are larger than 20 and 30 for A2 and B3, respectively. Thus, the high relative percentages of intra-rack failures may also be attributed to the machine rooms (e.g., high temperature in a machine room can lead to more SSD failures [30]).

Finding 4. *The relative percentages of intra-node and intra-rack failures vary across drive models. Putting too many SSDs from the same drive model in the same nodes (racks) leads to a high percentage of intra-node (intra-rack) failures. Also, the AFR and environmental factors (e.g., temperature) affect the relative percentages of intra-node and intra-rack failures.*

We vary the thresholds of the intra-node and intra-rack failure time intervals, broken down by the drive models. Figure 6 shows that the intra-node and intra-rack failures with a short failure time interval account for non-negligible percentages for most drive models. In particular, the relative percentages of intra-node (intra-rack) failures with a threshold of one day range from 4.4 to 34.3% (from 11.8 to 44.2%), except for A4 and C2 due to their limited numbers of SSDs per node or rack. The relative percentages of intra-node (intra-rack) failures with a threshold of one minute still account for 3.5-33.4% (7.8-37.1%) except for A4 and C2 (C2).

Finding 5. *There exist non-negligible fractions of intra-node and intra-rack failures with a short failure time interval for most drive models (e.g., up to 33.4% and 37.1% with a failure time interval of within one minute in our dataset, respectively)* .



(a) Intra-node failures (b) Intra-rack failures

Figure 6: Finding 5. Relative percentages of failures for different thresholds of intra-node or intra-rack failure time intervals across the drive models.

4.2.2 Lithography

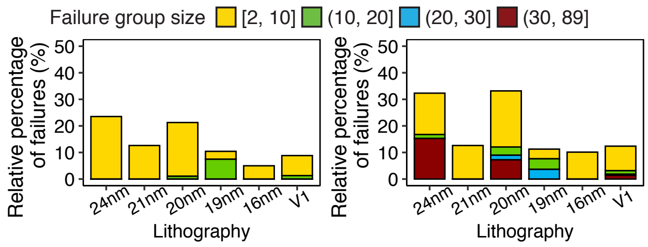
We analyze the impact of lithography on correlated failures. For MLC SSDs, a smaller lithography implies that the SSDs have a higher density. Also, 3D-TLC SSDs (C1 and C2) have higher densities than those of the MLC SSDs. Figure 7 shows that the SSDs of a smaller lithography (i.e., a higher density) generally have lower relative percentages of intra-node and intra-rack failures (over all SSD failures of the same lithography). In particular, for MLC SSDs, the relative percentages of intra-node (intra-rack) failures decrease from 23.5% to 5.0% (from 32.3% to 10.1%) from 24 nm to 16 nm. An exception is 21 nm SSDs, due to its limited number of failures. For 3D-TLC SSDs, the relative percentages of both intra-node and intra-rack failures are close to 19 nm MLC SSDs.

We also vary the thresholds of the intra-node and intra-rack failure time intervals, broken down by the lithography. Figure 8 shows that the relative percentages of intra-node and intra-rack failures for different thresholds decrease generally with a smaller lithography for MLC SSDs. In particular, the relative percentages of intra-node (intra-rack) failures with a threshold of one minute increase from 20.6% to 4.3% (28.0% to 9.3%) from 24 nm to 16 nm except for 21 nm SSDs due to few failures. The intra-rack failures with a threshold of one minute for 20 nm and 24 nm SSDs account for higher percentages than other MLC SSDs by 18.7-22.4%, since they include the intra-rack failures from A2 and B3, respectively (Figure 4(b)).

Finding 6. *MLC SSDs with higher densities generally have lower relative percentages of intra-node and intra-rack failures.*

4.2.3 Age

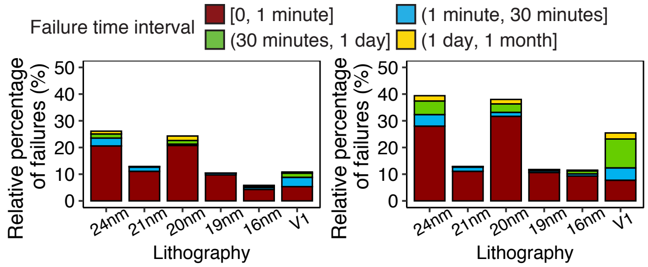
We analyze the impact of the age of a failed SSD (e.g., the power-on years until the failure occurs) on correlated failures. Figure 9 shows that the relative percentages of intra-node (intra-rack) failures (over all SSD failures of the same age group) for different sets of intra-node (intra-rack) group sizes increase with age in general. In particular, the relative percentages of intra-node (intra-rack) failures of each age group increase from 6.8% to 33.2% (from 11.0% to 37.6%) from



(a) Intra-node failures

(b) Intra-rack failures

Figure 7: Finding 6. Relative percentages of failures for different sets of intra-node or intra-rack failure group sizes across the lithography.



(a) Intra-node failures

(b) Intra-rack failures

Figure 8: Finding 6. Relative percentages of failures for different thresholds of intra-node or intra-rack failure time intervals across the lithography.

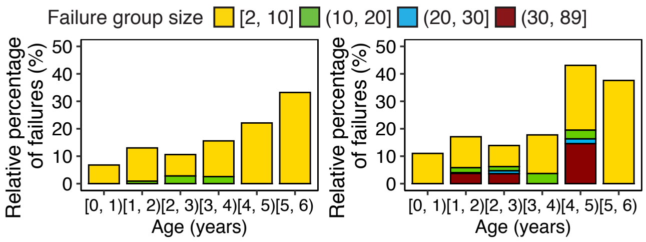
zero to six years old. Also, the relative percentage of intra-node (intra-rack) failures for 1-2 years old is slightly higher than that for 2-3 years old by 2.4% (3.2%). One possible reason is that the infant mortality of SSD failures can last for more than a year [15].

Figure 10 shows a noticeable trend that the intra-node and intra-rack failures at an older age are more likely to occur within a short time. In particular, the relative percentages of intra-node (intra-rack) failures with a threshold of one minute increase from 3.1% to 32.5% (from 5.2% to 36.9%) from zero to six years old. We also examine the average rated life used for intra-node and intra-rack failures at different ages (not shown in plots). The rated life used for intra-node (intra-rack) failures (with the default threshold of 30 minutes) increases from 1.6% (1.4%) for 0-1 year old to 67.5% (68.4%) for 5-6 years old on average, showing that a longer rated life used increases the likelihood of intra-node and intra-rack failures.

Finding 7. *The relative percentages of intra-node and intra-rack failures increase with age. The intra-node and intra-rack failures at an older age are more likely to occur within a short time due to the increasing rated life used.*

4.2.4 Capacity

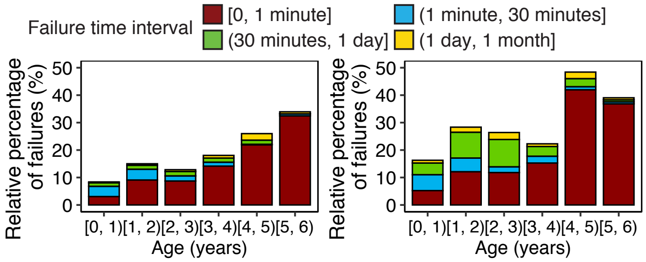
We examine the impact of the capacity on correlated failures. Figure 11 shows that the relative percentages of intra-node (intra-rack) failures (over all SSD failures of the same capacity) for different sets of intra-node (intra-rack) failure group sizes vary significantly across the capacity. Specifically, the



(a) Intra-node failures

(b) Intra-rack failures

Figure 9: Finding 7. Relative percentages of failures for different sets of intra-node or intra-rack failure group sizes across the age.



(a) Intra-node failures

(b) Intra-rack failures

Figure 10: Finding 7. Relative percentages of failures for different thresholds of intra-node or intra-rack failure time intervals across the age.

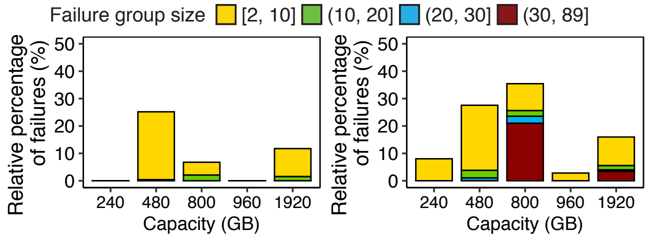
relative percentages of intra-node (intra-rack) failures for each capacity range from 0% to 25.2% (from 2.9% to 35.4%). As the SSDs with capacities of 480 GB, 800 GB, and 1920 GB cover more failures (Table 3), they have higher relative percentages of intra-node and intra-rack failures.

We next vary the thresholds of the intra-node and intra-rack failure time intervals, broken down by the capacity. Figure 12 shows no clear trend between the relative percentages of intra-node or intra-rack failures and the capacity. In particular, the 480 GB SSDs have the highest relative percentage of intra-node failures with a threshold of one minute, since they cover A3 with 34.4% of intra-node failures (Figure 6(a)), while the 800 GB SSDs have the highest relative percentage of intra-rack failures with a threshold of one minute, since they cover A2 with 36.6% of intra-rack failures (Figure 6(b)).

Finding 8. *The relative percentages of intra-node and intra-rack failures vary significantly across the capacity. There is no clear trend between the relative percentages of intra-node (or intra-rack) failures for different thresholds of failure time intervals and the capacity.*

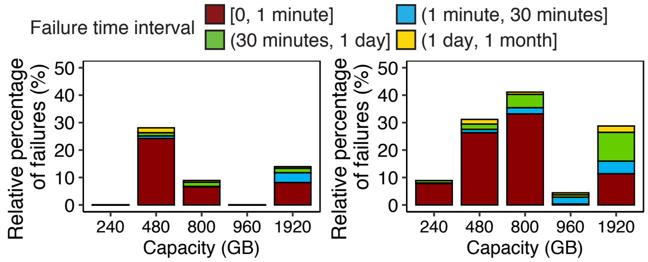
4.3 Impact of SMART Attributes

We analyze how SMART attributes are correlated with intra-node and intra-rack failures. We use the SRCC [29] (§3) to examine which SMART attributes are correlated with intra-node and intra-rack failures. Figure 13 shows that the SMART attributes have limited correlations with intra-node or intra-rack failures, and the differences of the absolute values of SRCC between intra-node and intra-rack failures are very



(a) Intra-node failures (b) Intra-rack failures

Figure 11: Finding 8. Relative percentages of failures for different sets of intra-node or intra-rack failure group sizes across the capacity.



(a) Intra-node failures (b) Intra-rack failures

Figure 12: Finding 8. Relative percentages of failures for different thresholds of intra-node or intra-rack failure time intervals across the capacity.

small. In particular, the SMART attributes related to internal errors (e.g., S187) are more correlated with intra-node and intra-rack failures, yet the highest SRCC values are only 0.23 for both intra-node and intra-rack failures. This implies that SMART attributes are not good indicators for detecting the existence of intra-node and intra-rack failures. Furthermore, the differences of the absolute values of SRCC between intra-node and intra-rack failures are very small and less than 0.02.

Finding 9. *The SMART attributes have limited correlations with intra-node and intra-rack failures, and the highest SRCC values (from S187) are only 0.23 for both intra-node and intra-rack failures. Thus, SMART attributes are not good indicators for detecting the existence of intra-node and intra-rack failures. Also, intra-node and intra-rack failures have no significant difference of the absolute values of SRCC for each SMART attribute.*

4.4 Impact of Applications

We analyze the relationships between the failure patterns and workload distributions of the eight applications (Table 2), and study the impact of applications on correlated failures.

We first examine the relationships between the AFRs and workload distributions of the eight applications. In particular, we use the raw values of SMART attributes S241 and S242 to calculate the percentage of writes among the total workloads of reads and writes, and determine if each SSD is read-dominant (i.e., more reads than writes) or write-dominant (i.e., more writes than reads). Figure 14(a) shows the average percentages of writes per SSD for the eight applications (each error bar shows the 95% confidence interval). Reads are dominant for WPS, SS, and DB, while writes are dominant for the remaining five applications. Figure 14(b) shows the AFRs for the applications. The AFRs of write-dominant applications in general are higher than those of read-dominant applications. This implies that write-dominant workloads lead to more SSD failures overall, conforming to prior findings [18].

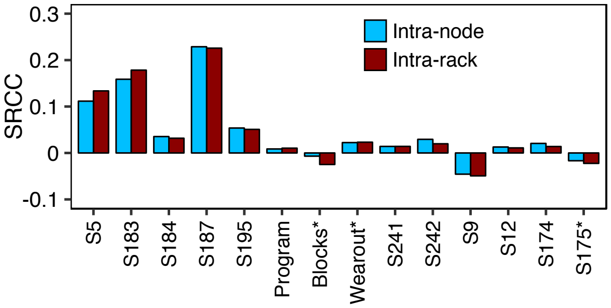
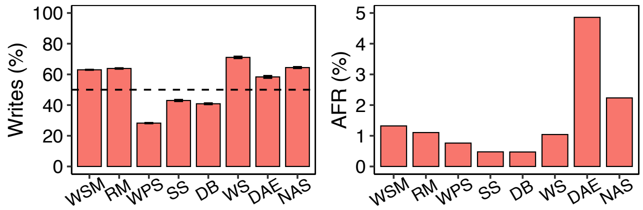


Figure 13: Finding 9. SRCC values between each SMART attribute and intra-node or intra-rack failures. Note that we omit three SMART attributes, including S197, S199, and erase failed counts, since their absolute SRCC values are less than 0.01 for both intra-node and intra-rack failures.



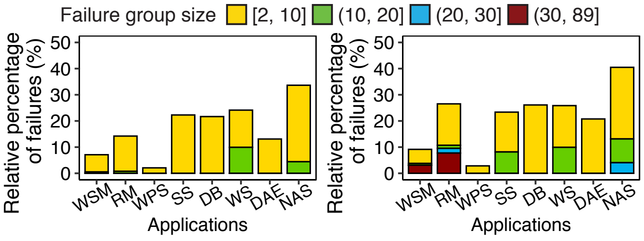
(a) Average percentages of writes per SSD (b) AFRs

Figure 14: Finding 10. Average percentages of writes per SSD (with 95% confidence intervals as error bars) and AFRs for the applications.

However, write-dominant workloads are not the only impacting factor on the AFRs. We see that DAE has the highest AFR (i.e., 4.9%), and it is mainly hosted on the drive model C1, which has a high AFR (3.29% in Table 3). Also, WPS has a higher AFR than SS and DB by 0.29%, although it has a lower percentage of writes than SS and DB. The reason is that C1 is mainly used in WPS, while A1, which has a low AFR (0.16% in Table 3), is the drive model mainly used in SS and DB.

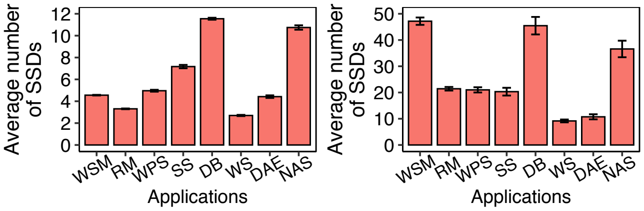
Finding 10. *Write-dominant workloads lead to more SSD failures overall, but are not the only impacting factor on the AFRs. Other factors (e.g., drive models) can affect the AFRs.*

We analyze the impact of applications on correlated failures. Figure 15 shows that the relative percentages of intra-node (intra-rack) failures (over all SSD failures of the same application) for different sets of intra-node (intra-rack) failure group sizes vary across the applications. In particular, the relative percentages of intra-node (intra-rack) failures for the applications range from 2.1% to 33.6% (from 2.8% to 40.5%).



(a) Intra-node failures (b) Intra-rack failures

Figure 15: Finding 11. Relative percentages of failures for different sets of intra-node or intra-rack failure group sizes across the applications.



(a) Per node (b) Per rack

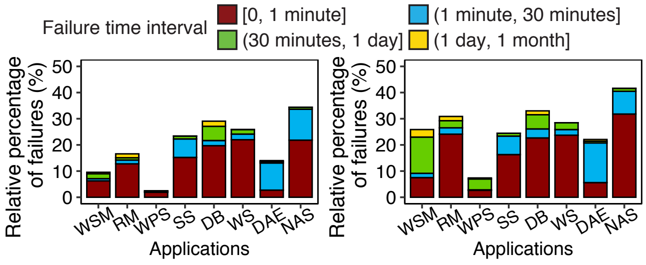
Figure 16: Finding 11. Average numbers of SSDs per node or rack for the applications (with 95% confidence intervals as error bars).

To explain these differences across the applications, we examine the average number of SSDs per node or rack for each application. Figure 16 shows that attaching more SSDs on nodes and racks for applications tends to have a high percentage of intra-node (intra-rack) failures. However, there are some exceptions. The average number of SSDs per node for WSM (4.6) is close to that of WPS (5.0), yet the relative percentage of intra-node failures for WSM is higher than that of WPS by 5.0%. The reason is that WPS has read-dominant workloads, while WSM has write-dominant workloads that lead to more failures (Figure 14(a)). Similar observations also hold for intra-rack failures. The average number of SSDs per rack for DAE (10.7) is much less than that for WPS (21.0), yet the relative percentage of intra-rack failures of DAE is higher than that of WPS by 17.9%.

Finding 11. *The applications with more SSDs per node (rack) and write-dominant workloads tend to have a high percentage of intra-node (intra-rack) failures.*

We further examine the impact of applications on correlated failures by varying the thresholds of the intra-node and intra-rack failure time intervals. Figure 17 shows that the relative percentages of intra-node and intra-rack failures for different thresholds of the failure time intervals vary across the applications. In particular, the relative percentages of intra-node (intra-rack) failures with a threshold of one minute account for 1.9-22.0% (2.6-31.8%).

To explain these differences among the applications, we examine the average ages of intra-node and intra-rack failures for the applications (not shown in plots). The average ages of intra-node (intra-rack) failures with a threshold of one minute for RM, SS, DB, and WS range from 3.2 to 3.9 years old (from 3.2 to 4.1 years old), which are older than those of the remaining applications, i.e., from 1.3 to 2.5 years old (from 1.2 to 2.2 years old). This conforms to Finding 7. However, there are two exceptions: (i) The average ages of intra-node and intra-rack failures for WS are younger than those for SS and DB by 0.48-0.65 years, while the relative percentage of intra-node (intra-rack) failures with a threshold of one minute for WS is higher than those for SS and DB by 2.3-4.5% (1.0-7.4%). (ii) The average age of intra-node (intra-rack) failures for NAS is younger than that for WPS by 1.0 (0.79) years, while the relative percentage of intra-node (intra-rack) failures with a threshold of one minute for NAS is higher than that for WPS by 19.9% (29.1%). The reasons for these exceptions are due to more write-dominant workloads for WS and NAS (Figure 14(a)).



(a) Intra-node failures (b) Intra-rack failures

Figure 17: Finding 12. Relative percentages of failures for different thresholds of intra-node or intra-rack failure time intervals across the applications.

Finding 12. *Among individual applications, the intra-node and intra-rack failures at an older age and with more write-dominant workloads tend to occur in a short time.*

4.5 Discussion

We highlight the findings in the correlation analysis:

- Intra-node and intra-rack failures commonly exist in SSD failures. Even worse, a non-negligible fraction of intra-node and intra-rack failures occur within a short time. In the presence of intra-node and intra-rack failures, it is critical to deploy the redundancy protection schemes with high fault tolerance to cope with such correlated failures.

- We analyze the effects of the four impacting factors, namely drive models, lithography, age, and capacity, on intra-node and intra-rack failures. We find that drive models and age have larger impacts on correlated failures than lithography and capacity. Also, intra-node (intra-rack) failures tend to occur with many SSDs from the same drive model on the same node (rack), and the intra-node and intra-rack failures of aged SSDs are more likely to occur within a short time. System operators should avoid putting such SSDs in the same scope to limit the occurrences of correlated failures.

- Intra-node and intra-rack failures have limited correlations with the SMART attributes and have no significant differences of correlations with each SMART attribute. Thus, the SMART attributes are not good indicators for detecting the existence of intra-node and intra-rack failures in practice. Other data sources, such as system logs, may be useful to detect any potential correlated failures.

- In addition to SSD characteristics, applications also play a role in the behavior of correlated failures. Intra-node and intra-rack failures are more likely to occur in write-dominant applications than read-dominant ones. Thus, high fault-tolerance protection schemes are more essential for write-dominant applications.

5 Case Study: Redundancy Protection

In this section, we present a trace-driven simulation analysis on how redundancy schemes affect the storage reliability in the face of correlated failures using our dataset.

5.1 Simulation Methodology

Redundancy schemes. Replication and erasure coding are two widely adopted redundancy approaches to provide fault tolerance in modern data centers. Our analysis considers three redundancy schemes:

- *r-way replication* (Rep(r)): For each data chunk, it makes $r > 1$ exact chunk copies to tolerate any $r-1$ chunk failures. We consider Rep(2) and Rep(3), where Rep(3) is used by traditional distributed file systems [6, 9].
- *Reed-Solomon coding* [23] (RS(k,m)): For every coding group of k data chunks, it encodes them into m parity chunks, such that any k out of $k+m$ data/parity chunks (i.e., any m chunk failures can be tolerated). We consider RS(6,3) (used by Google Colossus [7] and Quantcast File System [20]), RS(10,4) (used by Facebook [17]), and RS(12,4) (the same redundancy as in Azure [12]).
- *Local Reconstruction Coding* [12] (LRC(k,l,g)): For every coding group of k data chunks, it encodes each subgroup of k/l data chunks into a local parity chunk, and encodes all k data chunks into g global parity chunks. Thus, each single chunk failure can be reconstructed from any k/l non-failed chunks, while tolerating any $g+1$ chunk failures. We consider LRC(12,2,2), as used by Azure [12]. Note that it has the same redundancy as RS(12,4), but can only tolerate any three chunk failures and some of the four chunk failures (but not all four chunk failures as in RS(12,4)).

Replication is simple to implement, but incurs high storage overhead. Reed-Solomon coding incurs much lower storage overhead than replication, but incurs high *repair bandwidth* since any lost chunk needs to be reconstructed by accessing k non-failed chunks. Local Reconstruction Coding mitigates the repair bandwidth as any lost chunk can now be reconstructed by k/l non-failed chunks.

To mitigate repair bandwidth, we also consider *lazy recovery* [28], which triggers a repair operation only when more than one chunk fails (in Reed-Solomon coding, all data chunks remain available if no more than m chunks fail). This is in contrast to *eager recovery*, which triggers a repair operation immediately when there exists any failed chunk.

Simulator. We extend the C++ discrete-event simulator SimEDC [31] to support the reliability evaluation on our dataset. Our simulator runs multiple iterations. In each iteration, it initializes the data center topology, redundancy scheme, and chunk placement. It issues the failure events based on the chronological failure patterns in our dataset. It also generates the repair events, whose repair durations depend on the amount of repair bandwidth and the available data center capacity; for lazy recovery, the repair events are triggered only when a threshold number of failures occurs. Each iteration runs over a mission time. To generate randomness across iterations, we configure random chunk placements (see details below). We report the averaged results over all iterations.

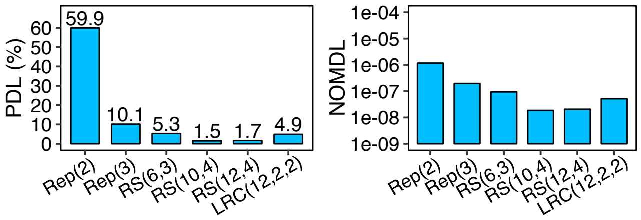
Metrics. We measure the reliability with the following metrics over the mission time:

- *Probability of data loss (PDL)*. It measures the likelihood that (unrecoverable) data loss occurs in a data center (i.e., the number of chunk failures in a coding group exceeds the tolerable limit).
- *Normalized magnitude of data loss (NOMDL)* [10]. It measures the amount of (unrecoverable) data loss (in bytes) normalized to the storage capacity.

Simulator setup. We configure the chunks in a coding group to be stored on different racks (one chunk per rack), so as to provide both node-level and rack-level fault tolerance. However, in our dataset, the number of racks varies highly across the clusters. Thus, we focus on the clusters that have at least 16 racks to support all redundancy schemes that we consider (the maximum number of chunks in a coding group is 16, for RS(12,4) and LRC(12,2,2)). To this end, we select 128 clusters from our dataset for evaluation. Due to the varying SSD capacity in our dataset, we fix the capacities of all SSDs as 512 GiB for simplicity. We set the chunk size as 256 MiB, the default chunk size in Facebook [24]. We also fix the same percentage of used storage capacity for data chunks as 50% for each redundancy scheme setting. We set the network link capacity for repair as 1 Gb/s, the parameter used for measuring the repair performance in erasure-coded storage [12, 24]. Furthermore, we set the mission time as ten years and run a sufficient number of iterations for each cluster until the relative error of PDL is less than 20% [31]. As our dataset spans only two years, we replay the dataset from beginning to end repeatedly in each iteration.

5.2 Simulation Results

We first evaluate the reliability of different redundancy schemes based on the SSD failure patterns in our dataset. Figure 18 shows that erasure coding achieves lower PDL and



(a) PDL (b) NOMDL

Figure 18: Finding 13. Comparison of the redundancy schemes.

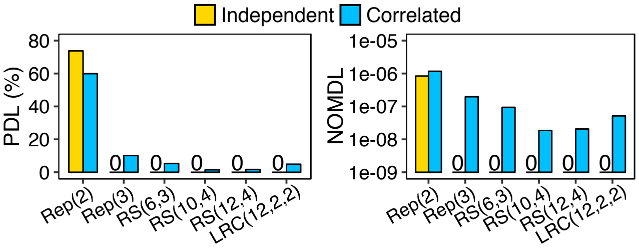
NOMDL (i.e., higher reliability) than replication. In particular, Rep(2) has the highest PDL (59.9%), indicating that two chunk copies are insufficient to tolerate failures. Also, Rep(3) is not good enough with a PDL of 10.1%. In contrast, RS(10,4) has the lowest PDL and NOMDL among all RS codes, since it tolerates more failures than RS(6,3) and has less repair bandwidth than RS(12,4). LRC(12,2,2) has slightly higher PDL and NOMDL than RS(12,4), since it cannot tolerate four chunks at any time.

Finding 13. *Erasure coding shows higher reliability than replication based on the failure patterns in our dataset.*

We claim that the redundancy schemes that are sufficient for tolerating independent failures may be insufficient for correlated failures. To justify this claim, we examine the reliability under only independent failures (generated from a mathematical failure model) and under the failure patterns in our dataset (including both independent and correlated ones). Specifically, we generate independent SSD failures following an exponential distribution with the mean time between failures (i.e., the number of hours in a year over the overall AFR in §2.2) in our dataset as the rate parameter, i.e., $\frac{8760}{1.16\%}$.

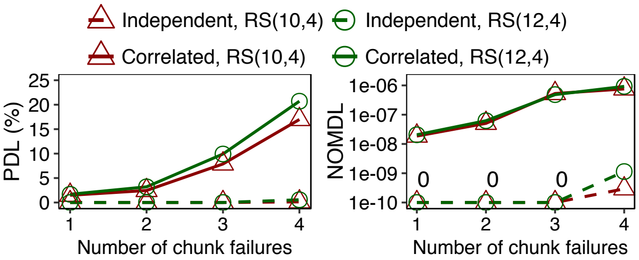
Figure 19 shows the results of the PDL and NOMDL for eager recovery under only independent failures and the failure patterns in our dataset. The PDL and NOMDL under only independent failures for Rep(3), RS(6,3), RS(10,4), RS(12,4), and LRC(12,2,2) are zero. However, the reliability of these redundancy schemes degrades under the failure patterns in our dataset. The reason is that some correlated failures occur within a short time period (Finding 3) and additional failures are likely to occur in a short time with the existing correlated failures on the same node or rack (Finding 2), leading to the competition for network bandwidth resources and a slowdown of the repair process. This increases the likelihood of data loss. In addition, the PDL under only independent failures for Rep(2) is higher than that under the failures in our dataset by 13.8%. The reason is that the number of failures generated by the mathematical failure model may be more than that in our dataset for some clusters, leading to more failed chunks that exceed the tolerable limit of Rep(2). This implies that Rep(2) is still insufficient under only independent failures.

Finding 14. *Redundancy schemes that are sufficient for tolerating independent failures may be insufficient for tolerating the correlated failures as shown in our dataset.*



(a) PDL (b) NOMDL

Figure 19: Finding 14. Comparison of the PDL and NOMDL of eager recovery under independent failures ("Independent") and the failure patterns in our dataset ("Correlated").



(a) PDL (b) NOMDL

Figure 20: Finding 15. Comparison of the PDL and NOMDL for the threshold number of chunk failures for recovery under only independent failures ("Independent") and the failure patterns in our dataset ("Correlated").

We next evaluate the reliability of lazy recovery under only independent failures derived from the mathematical failure model and under the failure patterns in our dataset. For lazy recovery, we vary the threshold of triggering recovery from one to four failed chunks (note that four is the tolerable limit for RS(10,4) and RS(12,4)); a threshold of one implies eager recovery.

Figure 20 shows that RS(10,4) and RS(12,4) achieve a high reliability under only independent failures, but their reliability degrades under the failure patterns in our dataset as the threshold increases. In particular, under only independent failures, RS(10,4) and RS(12,4) can achieve a high reliability without data loss with a threshold of one to three failed chunks, conforming to the prior work [18]. They have a small PDL (0.14-0.56%) with a threshold of four failed chunks since having any additional failed chunk will lead to data loss. However, under the failure patterns in our dataset, the PDL values for RS(10,4) and RS(12,4) increase by 0.98-1.5% when the threshold increases from one to two failed chunks, and continue to increase by more than 10% from two to four failed chunks. The reason of the reliability degradation of lazy recovery under the failures in our dataset is that when the number of failed chunks reaches a larger threshold of chunk failures, additional correlated failures are also more likely to occur in a short time (Findings 2 and 3). Thus, the most proper threshold number of chunk failures is one, i.e., eager recovery, under the failure patterns in our dataset.

Finding 15. *Lazy recovery is less suitable than eager recovery for tolerating correlated failures in our dataset.*

6 Related Work

SSD measurement. Field studies have analyzed the reliability of SSDs and characterized the correlations between SSD failures and their symptoms [2,15,16,18,27,30]. For example, some studies [16,18,27] analyze the symptoms (e.g., uncorrectable errors) reported by proprietary customized attributes and SMART attributes in SSD failures. Xu *et al.* [30] investigate the effects of system-level symptoms on SSD failures. Alter *et al.* [2] exploit the failure patterns from the symptoms to predict future SSD failures. Maneas *et al.* [15] analyze how SSD replacements and other factors affect the replacement rates within a RAID system. Although some studies [15,16] report the existence of correlated failures in SSD-based storage systems, they do not cover the location details of SSD failures due to the limited information in their datasets. In general, the above studies mainly focus on how SSD failures are correlated with different factors, while our work focuses on the correlations among the SSD failures. In particular, we characterize the correlated failures within a node or a rack. We study the impact of different factors on correlated failures, and the implications on storage reliability under correlated failures in SSD-based data centers.

HDD measurement. Field studies have analyzed the reliability of hard disk drives (HDDs) in production environments. Pinheiro *et al.* [22] analyze different factors that are correlated with HDD failures based on SMART logs at Google. Schroeder *et al.* [26] characterize the HDD replacement rates statistically. Also, prior studies present the patterns of latent sector errors [4,25] and data corruptions [3] at NetApp. In the literature, Lu *et al.* [14] leverage the locations of HDDs to predict HDD failures. Instead, our work uses the locations to study correlated failures of SSDs.

Correlated failures. Prior studies have characterized the correlated failures on various storage scopes. Chun *et al.* [5] and Nath *et al.* [19] investigate the correlated failures that threaten the durability and availability of storage systems. Schroeder *et al.* [25,26] provide a statistical analysis on correlated failures of hard disks and the bursts of latent sector errors in disks. Ford *et al.* [8] characterize the statistical behavior of correlated node failures. In contrast, we focus on characterizing the correlated failures in SSD-based data centers in a more comprehensive manner.

7 Conclusion

We present an in-depth analysis on correlated failures of SSDs based on the large-scale dataset at Alibaba. Our analysis includes spatial and temporal correlations of SSD failures and the impact of different factors on correlated failures. We also evaluate the reliability of various redundancy schemes under correlated failures via trace-driven simulation. We report 15 findings, and release our dataset and source code for public validation.

Acknowledgement

We thank our shepherd, Jiri Schindler, and the anonymous reviewers for their comments. We also thank Qiuping Wang and Jinhong Li for their feedback. This work was supported in part by Alibaba Group via the Alibaba Innovation Research (AIR) program and the Research Grants Council of Hong Kong (AoE/P-404/18).

References

[1] R. Alagappan, A. Ganesan, Y. Patel, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Correlated crash vulnerabilities. In *Proc. of USENIX OSDI*, 2016.

[2] J. Alter, J. Xue, A. Dimnaku, and E. Smirni. SSD failures in the field: Symptoms, causes, and prediction models. In *Proc. of ACM/IEEE SC*, 2019.

[3] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. *ACM Trans. on Storage*, 4(3):8, Nov 2008.

[4] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *Proc. of ACM SIGMETRICS*, 2007.

[5] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. T. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of USENIX NSDI*, 2006.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of ACM SOSP*, 2007.

[7] A. Fikes. Storage architecture and challenges. *Talk at the Google Faculty Summit*, 2010.

[8] D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, 2010.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. of ACM SOSP*, 2003.

[10] K. M. Greenan, J. S. Plank, J. J. Wylie, et al. Mean time to meaningless: MTTDL, Markov models, and storage system reliability. In *Proc. of USENIX HotStorage*, 2010.

[11] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of USENIX NSDI*, 2005.

[12] 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 ATC*, 2012.

[13] S. Kadekodi, K. Rashmi, and G. R. Ganger. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. In *Proc. of USENIX FAST*, 2019.

[14] S. Lu, B. Luo, T. Patel, Y. Yao, D. Tiwari, and W. Shi. Making disk failure predictions SMARTer! In *Proc. of USENIX FAST*, 2020.

[15] S. Maneas, K. Mahdaviani, T. Emami, and B. Schroeder. A study of SSD reliability in large scale enterprise storage deployments. In *Proc. of USENIX FAST*, 2020.

[16] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. A large-scale study of flash memory failures in the field. In *Proc. of ACM SIGMETRICS*, 2015.

[17] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm BLOB storage system. In *Proc. of USENIX OSDI*, 2014.

[18] I. Narayanan, D. Wang, M. Jeon, B. Sharma, L. Caulfield, A. Sivasubramaniam, B. Cutler, J. Liu, B. Khessib, and K. Vaid. SSD failures in datacenters: What? when? and why? In *Proc. of ACM SYSTOR*, 2016.

[19] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proc. of USENIX NSDI*, 2006.

[20] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast file system. In *Proc. of the VLDB Endowment*, 2013.

[21] K. Pearson. Vii. note on regression and inheritance in the case of two parents. *Proc. of the Royal Society of London*, 58(347-352):240–242, 1895.

[22] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proc. of USENIX FAST*, 2007.

[23] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[24] 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*, 2013.

[25] B. Schroeder, S. Damouras, and P. Gill. Understanding latent sector errors and how to protect against them. *ACM Trans. on Storage*, 6(3):1–23, 2010.

[26] B. Schroeder and G. A. Gibson. Understanding disk failure rates: What does an MTTF of 1,000,000 hours mean to you? *ACM Trans. on Storage (TOS)*, 3(3):8–es, 2007.

[27] B. Schroeder, R. Lagisetty, and A. Merchant. Flash reliability in production: The expected and the unexpected. In *Proc. of USENIX FAST*, 2016.

[28] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proc. of ACM SYSTOR*, 2014.

[29] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 100(3/4):441–471, 1987.

[30] E. Xu, M. Zheng, F. Qin, Y. Xu, and J. Wu. Lessons and actions: What we learned from 10K SSD-related storage system failures. In *Proc. of USENIX ATC*, 2019.

[31] M. Zhang, S. Han, and P. P. Lee. SimEDC: A simulator for the reliability analysis of erasure-coded data centers. *IEEE Trans. on Parallel and Distributed Systems*, 30(12):2836–2848, 2019.