usenix ASSOCIATION

conference

proceedings

12th USENIX Conference on File and Storage Technologies

Santa Clara, CA, USA
February 17–20, 2014

Proceedings of the 12th USENIX Conference on File and Storage Technologies

Santa Clara, CA, USA February 17–20, 2014

Sponsored by

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

In cooperation with ACM SIGOPS

Thanks to Our FAST '14 Sponsors

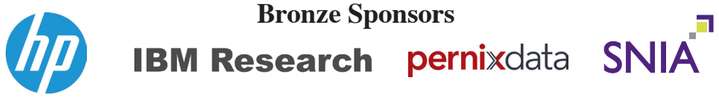
Open Access Sponsor

Platinum Sponsor

Gold Sponsors

Silver Sponsor

Bronze Sponsors



General Sponsors

Media Sponsors and Industry Partners

ACM *Queue*
ADMIN magazine
Distributed Management Task Force (DMTF)
EnterpriseTech
HPCwire
InfoSec News
Linux Pro Magazine
LXer
No Starch Press
O'Reilly Media
Raspberry Pi Geek
UserFriendly.org

ISBN 978-1-931971-08-9

Thanks to Our USENIX and LISA SIG Supporters

USENIX Patrons

Google Microsoft Research NetApp VMware

USENIX Benefactors

Akamai Citrix Facebook *Linux Pro Magazine* Puppet Labs

USENIX and LISA Partners

Cambridge Computer Google

USENIX Partners

EMC Meraki

USENIX Association

Proceedings of the 12th USENIX Conference on File and Storage Technologies

February 17–20, 2014
Santa Clara, CA

Conference Organizers

Program Co-Chairs

Bianca Schroeder, *University of Toronto*

Eno Thereska, *Microsoft Research*

Program Committee

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

Andre Brinkmann, *Universität Mainz*

Landon Cox, *Duke University*

Angela Demke-Brown, *University of Toronto*

Jason Flinn, *University of Michigan*

Garth Gibson, *Carnegie Mellon University and Panasas*

Steven Hand, *University of Cambridge*

Randy Katz, *University of California, Berkeley*

Kimberly Keeton, *HP Labs*

Jay Lorch, *Microsoft Research*

C.S. Lui, *The Chinese University of Hong Kong*

Arif Merchant, *Google*

Ethan Miller, *University of California, Santa Cruz*

Brian Noble, *University of Michigan*

Sam H. Noh, *Hongik University*

James Plank, *University of Tennesee*

Florentina Popovici, *Google*

Raju Rangaswami, *Florida International University*

Erik Riedel, *EMC*

Jiri Schindler, *NetApp*

Anand Sivasubramaniam, *Pennsylvania State University*

Steve Swanson, *University of California, San Diego*

Tom Talpey, *Microsoft*

Andrew Warfield, *University of British Columbia and Coho Data*

Hakim Weatherspoon, *Cornell University*

Erez Zadok, *Stony Brook University*

Xiaodong Zhang, *Ohio State University*

Zheng Zhang, *Microsoft Research Beijing*

Steering Committee

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

William J. Bolosky, *Microsoft Research*

Randal Burns, *Johns Hopkins University*

Jason Flinn, *University of Michigan*

Greg Ganger, *Carnegie Mellon University*

Garth Gibson, *Carnegie Mellon University and Panasas*

Casey Henderson, *USENIX Association*

Kimberly Keeton, *HP Labs*

Darrell Long, *University of California, Santa Cruz*

Jai Menon, *Dell*

Erik Riedel, *EMC*

Margo Seltzer, *Harvard School of Engineering and Applied Sciences and Oracle*

Keith A. Smith, *NetApp*

Ric Wheeler, *Red Hat*

John Wilkes, *Google*

Yuanyuan Zhou, *University of California, San Diego*

Tutorial Coordinator

John Strunk, *NetApp*

External Reviewers

Rachit Agarwal
Ganesh Ananthanarayanan
Christos Gkantsidis
Jacob Gorm Hansen
Cheng Huang
Qiao Lian
K. Shankari
Shivaram Venkataraman
Neeraja Yadwadkar

12th USENIX Conference on File and Storage Technologies
February 17–20, 2014
Santa Clara, CA

Tuesday, February 18, 2014

Big Memory

Flash and SSDs

Personal and Mobile

(Tuesday, February 18, continues on p. iv)

RAID and Erasure Codes

Wednesday, February 19, 2014

Experience from Real Systems

Performance and Efficiency

Thursday, February 20, 2014

OS and Storage Interactions

OS and Peripherals

Message from the 12th USENIX Conference on File and Storage Technologies Program Co-Chairs

Welcome to the 12th USENIX Conference on File and Storage Technologies. This year's conference continues the FAST tradition of bringing together researchers and practitioners from both industry and academia for a program of innovative and rigorous storage-related research. We are pleased to present a diverse set of papers on topics such as personal and mobile storage, RAID and erasure codes, experiences from building and running real systems, flash and SSD, performance, reliability and efficiency of storage systems, and interactions between operating and storage system. Our authors hail from seven countries on three continents and represent both academia and industry. Many of our papers are the fruits of collaboration between the two.

FAST '14 received 133 submissions, nearly equalling the record number of submissions (137) from FAST '12. Of these, we selected 24, for an acceptance rate of 18%. Six accepted papers have Program Committee authors. The Program Committee used a two-round online review process, and then met in person to select the final program. In the first round, each paper received three reviews. For the second round, 64 papers received two or more additional reviews. The Program Committee discussed 54 papers in an all-day meeting on December 6, 2013, in Toronto, Canada. We used Eddie Kohler's excellent HotCRP software to manage all stages of the review process, from submission to author notification.

As in the previous two years, we have again included a category of short papers in the program. Short papers provide a vehicle for presenting research ideas that do not require a full-length paper to describe and evaluate. In judging short papers, we applied the same standards as for full-length submissions. 32 of our submissions were short papers, of which we accepted three.

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 research to FAST '14. We had a wide range of high-quality work from which to choose our program. We would also like to thank the attendees of FAST '14 and future readers of these papers. Together with the authors, you form the FAST community and make storage research vibrant and fun. We also extend our thanks to the staff of USENIX, who have provided outstanding support throughout the planning and organizing of this conference. They gave advice, anticipated our needs, and guided us through the logistics of planning a large conference with professionalism and good humor. Most importantly, they handled all of the behind-the-scenes work that makes this conference actually happen. Thanks go also to the members of the FAST Steering Committee who provided invaluable advice and feedback. Thanks!

Finally, we wish to thank our Program Committee for their many hours of hard work in reviewing and discussing the submissions. We were privileged to work with this knowledgeable and dedicated group of researchers. Together with our external reviewers, they wrote over 500 thoughtful and meticulous reviews. Their reviews, and their thorough and conscientious deliberations at the PC meeting, contributed significantly to the quality of our decisions. We also thank the three student volunteers, Nosayba El-Sayed, Andy Hwang and Ioan Stefanovici, who helped us organize the PC meeting.

We look forward to an interesting and enjoyable conference!

Bianca Schroeder, ***University of Toronto***
Eno Thereska, ***Microsoft Research***
FAST '14 Program Co-Chairs

Log-structured Memory for DRAM-based Storage

Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout
{rumble, ankitak, ouster}@cs.stanford.edu
Stanford University

Abstract

Traditional memory allocation mechanisms are not suitable for new DRAM-based storage systems because they use memory inefficiently, particularly under changing access patterns. In contrast, a log-structured approach to memory management allows 80-90% memory utilization while offering high performance. The RAMCloud storage system implements a unified log-structured mechanism both for active information in memory and backup data on disk. The RAMCloud implementation of log-structured memory uses a two-level cleaning policy, which conserves disk bandwidth and improves performance up to 6x at high memory utilization. The cleaner runs concurrently with normal operations and employs multiple threads to hide most of the cost of cleaning.

1 Introduction

In recent years a new class of storage systems has arisen in which all data is stored in DRAM. Examples include memcached [2], Redis [3], RAMCloud [30], and Spark [38]. Because of the relatively high cost of DRAM, it is important for these systems to use their memory efficiently. Unfortunately, efficient memory usage is not possible with existing general-purpose storage allocators: they can easily waste half or more of memory, particularly in the face of changing access patterns.

In this paper we show how a log-structured approach to memory management (treating memory as a sequentially-written log) supports memory utilizations of 80-90% while providing high performance. In comparison to non-copying allocators such as malloc, the log-structured approach allows data to be copied to eliminate fragmentation. Copying allows the system to make a fundamental space-time trade-off: for the price of additional CPU cycles and memory bandwidth, copying allows for more efficient use of storage space in DRAM. In comparison to copying garbage collectors, which eventually require a global scan of all data, the log-structured approach provides garbage collection that is more incremental. This results in more efficient collection, which enables higher memory utilization.

We have implemented log-structured memory in the RAMCloud storage system, using a unified approach that handles both information in memory and backup replicas stored on disk or flash memory. The overall architecture is similar to that of a log-structured file system [32], but with several novel aspects:

- In contrast to log-structured file systems, log-structured memory is simpler because it stores very little metadata in the log. The only metadata consists of *log digests* to enable log reassembly after crashes, and *tombstones* to prevent the resurrection of deleted objects.
- RAMCloud uses a *two-level* approach to cleaning, with different policies for cleaning data in memory versus secondary storage. This maximizes DRAM utilization while minimizing disk and network bandwidth usage.
- Since log data is immutable once appended, the log cleaner can run concurrently with normal read and write operations. Furthermore, multiple cleaners can run in separate threads. As a result, *parallel cleaning* hides most of the cost of garbage collection.

Performance measurements of log-structured memory in RAMCloud show that it enables high client throughput at 80-90% memory utilization, even with artificially stressful workloads. In the most stressful workload, a single RAMCloud server can support 270,000-410,000 durable 100-byte writes per second at 90% memory utilization. The two-level approach to cleaning improves performance by up to 6x over a single-level approach at high memory utilization, and reduces disk bandwidth overhead by 7-87x for medium-sized objects (1 to 10 KB). Parallel cleaning effectively hides the cost of cleaning: an active cleaner adds only about 2% to the latency of typical client write requests.

2 Why Not Use Malloc?

An off-the-shelf memory allocator such as the C library's malloc function might seem like a natural choice for an in-memory storage system. However, existing allocators are not able to use memory efficiently, particularly in the face of changing access patterns. We measured a variety of allocators under synthetic workloads and found that all of them waste at least 50% of memory under conditions that seem plausible for a storage system.

Memory allocators fall into two general classes: non-copying allocators and copying allocators. *Non-copying* allocators such as malloc cannot move an object once it has been allocated, so they are vulnerable to fragmentation. Non-copying allocators work well for individual applications with a consistent distribution of object sizes, but Figure 1 shows that they can easily waste half of memory when allocation patterns change. For example, every allocator we measured performed poorly when 10 GB of small objects were mostly deleted, then replaced with 10 GB of much larger objects.

Changes in size distributions may be rare in individual

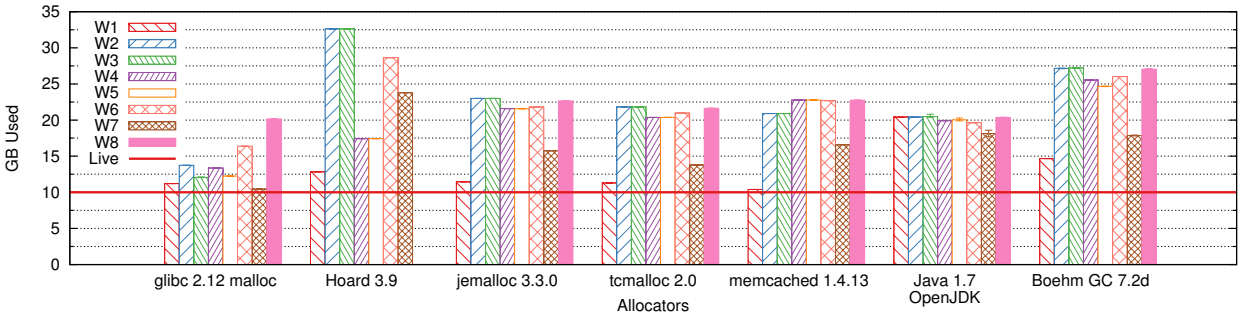


Figure 1: Total memory needed by allocators to support 10 GB of live data under the changing workloads described in Table 1 (average of 5 runs). "Live" indicates the amount of live data, and represents an optimal result. "glibc" is the allocator typically used by C and C++ applications on Linux. "Hoard" [10], "jemalloc" [19], and "tcmalloc" [1] are non-copying allocators designed for speed and multiprocessor scalability. "Memcached" is the slab-based allocator used in the memcached [2] object caching system. "Java" is the JVM's default parallel scavenging collector with no maximum heap size restriction (it ran out of memory if given less than 16 GB of total space). "Boehm GC" is a non-copying garbage collector for C and C++. Hoard could not complete the W8 workload (it overburdened the kernel by *mmap*ing each large allocation separately).

Workload	Before	Delete	After
W1	Fixed 100 Bytes	N/A	N/A
W2	Fixed 100 Bytes	0%	Fixed 130 Bytes
W3	Fixed 100 Bytes	90%	Fixed 130 Bytes
W4	Uniform 100 - 150 Bytes	0%	Uniform 200 - 250 Bytes
W5	Uniform 100 - 150 Bytes	90%	Uniform 200 - 250 Bytes
W6	Uniform 100 - 200 Bytes	50%	Uniform 1,000 - 2,000 Bytes
W7	Uniform 1,000 - 2,000 Bytes	90%	Uniform 1,500 - 2,500 Bytes
W8	Uniform 50 - 150 Bytes	90%	Uniform 5,000 - 15,000 Bytes

Table 1: Summary of workloads used in Figure 1. The workloads were not intended to be representative of actual application behavior, but rather to illustrate plausible workload changes that might occur in a shared storage system. Each workload consists of three phases. First, the workload allocates 50 GB of memory using objects from a particular size distribution; it deletes existing objects at random in order to keep the amount of live data from exceeding 10 GB. In the second phase the workload deletes a fraction of the existing objects at random. The third phase is identical to the first except that it uses a different size distribution (objects from the new distribution gradually displace those from the old distribution). Two size distributions were used: "Fixed" means all objects had the same size, and "Uniform" means objects were chosen uniform randomly over a range (non-uniform distributions yielded similar results). All workloads were single-threaded and ran on a Xeon E5-2670 system with Linux 2.6.32.

applications, but they are more likely in storage systems that serve many applications over a long period of time. Such shifts can be caused by changes in the set of applications using the system (adding new ones and/or removing old ones), by changes in application phases (switching from map to reduce), or by application upgrades that increase the size of common records (to include additional fields for new features). For example, workload W2 in Figure 1 models the case where the records of a table are expanded from 100 bytes to 130 bytes. Facebook encountered distribution changes like this in its memcached storage systems and was forced to introduce special-purpose cache eviction code for specific situations [28]. Non-copying allocators will work well in many cases, but they are unstable: a small application change could dramatically change the efficiency of the storage system. Unless excess memory is retained to handle the worst-case change, an application could suddenly find itself unable to make progress.

The second class of memory allocators consists of those that can move objects after they have been created, such as copying garbage collectors. In principle, garbage collectors can solve the fragmentation problem by moving live data to coalesce free heap space. However, this comes with a trade-off: at some point all of these collectors (even those that label themselves as "incremental") must walk all live data, relocate it, and update references. This is an expensive operation that scales poorly, so garbage collectors delay global collections until a large amount of garbage has accumulated. As a result, they typically require 1.5-5x as much space as is actually used in order to maintain high performance [39, 23]. This erases any space savings gained by defragmenting memory.

Pause times are another concern with copying garbage collectors. At some point all collectors must halt the processes' threads to update references when objects are moved. Although there has been considerable work on real-time garbage collectors, even state-of-art solutions have maximum pause times of hundreds of microseconds, or even milliseconds [8, 13, 36] – this is 100 to 1,000 times longer than the round-trip time for a RAMCloud RPC. All of the standard Java collectors we measured exhibited pauses of 3 to 4 seconds by default (2-4 times longer than it takes RAMCloud to detect a failed server and reconstitute 64 GB of lost data [29]). We experimented with features of the JVM collectors that re-

duce pause times, but memory consumption increased by an additional 30% and we still experienced occasional pauses of one second or more.

An ideal memory allocator for a DRAM-based storage system such as RAMCloud should have two properties. First, it must be able to copy objects in order to eliminate fragmentation. Second, it must not require a global scan of memory: instead, it must be able to perform the copying *incrementally*, garbage collecting small regions of memory independently with cost proportional to the size of a region. Among other advantages, the incremental approach allows the garbage collector to focus on regions with the most free space. In the rest of this paper we will show how a log-structured approach to memory management achieves these properties.

In order for incremental garbage collection to work, it must be possible to find the pointers to an object without scanning all of memory. Fortunately, storage systems typically have this property: pointers are confined to index structures where they can be located easily. Traditional storage allocators work in a harsher environment where the allocator has no control over pointers; the log-structured approach could not work in such environments.

3 RAMCloud Overview

Our need for a memory allocator arose in the context of RAMCloud. This section summarizes the features of RAMCloud that relate to its mechanisms for storage management, and motivates why we used log-structured memory instead of a traditional allocator.

RAMCloud is a storage system that stores data in the DRAM of hundreds or thousands of servers within a datacenter, as shown in Figure 2. It takes advantage of low-latency networks to offer remote read times of 5μs and write times of 16μs (for small objects). Each storage server contains two components. A *master* module manages the main memory of the server to store RAMCloud objects; it handles read and write requests from clients. A *backup* module uses local disk or flash memory to store backup copies of data owned by masters on other servers. The masters and backups are managed by a central *coordinator* that handles configuration-related issues such as cluster membership and the distribution of data among the servers. The coordinator is not normally involved in common operations such as reads and writes. All RAMCloud data is present in DRAM at all times; secondary storage is used only to hold duplicate copies for crash recovery.

RAMCloud provides a simple key-value data model consisting of uninterpreted data blobs called *objects* that are named by variable-length *keys*. Objects are grouped into *tables* that may span one or more servers in the cluster. Objects must be read or written in their entirety. RAMCloud is optimized for small objects – a few hundred bytes or less – but supports objects up to 1 MB.

Each master's memory contains a collection of objects stored in DRAM and a hash table (see Figure 3). The

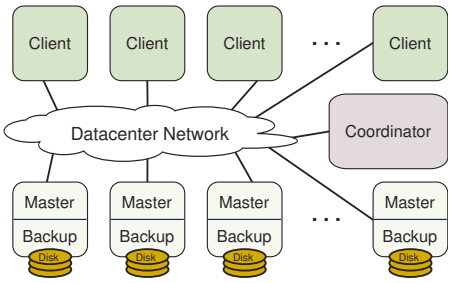


Figure 2: RAMCloud cluster architecture.

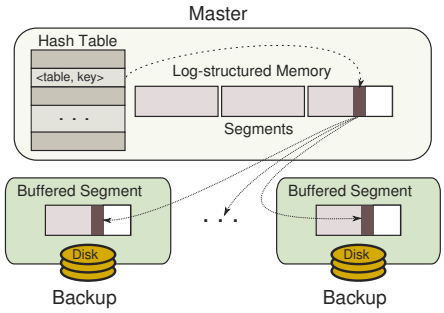


Figure 3: Master servers consist primarily of a hash table and an in-memory log, which is replicated across several backups for durability.

hash table contains one entry for each object stored on that master; it allows any object to be located quickly, given its table and key. Each live object has exactly one pointer, which is stored in its hash table entry.

In order to ensure data durability in the face of server crashes and power failures, each master must keep backup copies of its objects on the secondary storage of other servers. The backup data is organized as a log for maximum efficiency. Each master has its own log, which is divided into 8 MB pieces called *segments*. Each segment is replicated on several backups (typically two or three). A master uses a different set of backups to replicate each segment, so that its segment replicas end up scattered across the entire cluster.

When a master receives a write request from a client, it adds the new object to its memory, then forwards information about that object to the backups for its current head segment. The backups append the new object to segment replicas stored in nonvolatile buffers; they respond to the master as soon as the object has been copied into their buffer, without issuing an I/O to secondary storage (backups must ensure that data in buffers can survive power failures). Once the master has received replies from all the backups, it responds to the client. Each backup accumulates data in its buffer until the segment is complete. At that point it writes the segment to secondary storage and reallocates the buffer for another segment. This approach has two performance advantages: writes complete without waiting for I/O to secondary storage, and backups use secondary storage bandwidth efficiently by performing I/O in large blocks, even if objects are small.

RAMCloud could have used a traditional storage allocator for the objects stored in a master's memory, but we chose instead to use the same log structure in DRAM that is used on disk. Thus a master's object storage consists of 8 MB segments that are identical to those on secondary storage. This approach has three advantages. First, it avoids the allocation inefficiencies described in Section 2. Second, it simplifies RAMCloud by using a single unified mechanism for information both in memory and on disk. Third, it saves memory: in order to perform log cleaning (described below), the master must enumerate all of the objects in a segment; if objects were stored in separately allocated areas, they would need to be linked together by segment, which would add an extra 8-byte pointer per object (an 8% memory overhead for 100-byte objects).

The segment replicas stored on backups are never read during normal operation; most are deleted before they have ever been read. Backup replicas are only read during crash recovery (for details, see [29]). Data is never read from secondary storage in small chunks; the only read operation is to read a master's entire log.

RAMCloud uses a *log cleaner* to reclaim free space that accumulates in the logs when objects are deleted or overwritten. Each master runs a separate cleaner, using a basic mechanism similar to that of LFS [32]:

- The cleaner selects several segments to clean, using the same cost-benefit approach as LFS (segments are chosen for cleaning based on the amount of free space and the age of the data).
- For each of these segments, the cleaner scans the segment stored in memory and copies any live objects to new *survivor segments*. Liveness is determined by checking for a reference to the object in the hash table. The live objects are sorted by age to improve the efficiency of cleaning in the future. Unlike LFS, RAMCloud need not read objects from secondary storage during cleaning.
- The cleaner makes the old segments' memory available for new segments, and it notifies the backups for those segments that they can reclaim the replicas' storage.

The logging approach meets the goals from Section 2: it copies data to eliminate fragmentation, and it operates incrementally, cleaning a few segments at a time. However, it introduces two additional issues. First, the log must contain metadata in addition to objects, in order to ensure safe crash recovery; this issue is addressed in Section 4. Second, log cleaning can be quite expensive at high memory utilization [34, 35]. RAMCloud uses two techniques to reduce the impact of log cleaning: two-level cleaning (Section 5) and parallel cleaning with multiple threads (Section 6).

4 Log Metadata

In log-structured file systems, the log contains a lot of indexing information in order to provide fast random access to data in the log. In contrast, RAMCloud has a separate hash table that provides fast access to information in memory. The on-disk log is never read during normal use; it is used only during recovery, at which point it is read in its entirety. As a result, RAMCloud requires only three kinds of metadata in its log, which are described below.

First, each object in the log must be self-identifying: it contains the table identifier, key, and version number for the object in addition to its value. When the log is scanned during crash recovery, this information allows RAMCloud to identify the most recent version of an object and reconstruct the hash table.

Second, each new log segment contains a *log digest* that describes the entire log. Every segment has a unique identifier, and the log digest is a list of identifiers for all the segments that currently belong to the log. Log digests avoid the need for a central repository of log information (which would create a scalability bottleneck and introduce other crash recovery problems). To replay a crashed master's log, RAMCloud locates the latest digest and loads each segment enumerated in it (see [29] for details).

The third kind of log metadata is *tombstones* that identify deleted objects. When an object is deleted or modified, RAMCloud does not modify the object's existing record in the log. Instead, it appends a *tombstone* record to the log. The tombstone contains the table identifier, key, and version number for the object that was deleted. Tombstones are ignored during normal operation, but they distinguish live objects from dead ones during crash recovery. Without tombstones, deleted objects would come back to life when logs are replayed during crash recovery.

Tombstones have proven to be a mixed blessing in RAMCloud: they provide a simple mechanism to prevent object resurrection, but they introduce additional problems of their own. One problem is tombstone garbage collection. Tombstones must eventually be removed from the log, but this is only safe if the corresponding objects have been cleaned (so they will never be seen during crash recovery). To enable tombstone deletion, each tombstone includes the identifier of the segment containing the obsolete object. When the cleaner encounters a tombstone in the log, it checks the segment referenced in the tombstone. If that segment is no longer part of the log, then it must have been cleaned, so the old object no longer exists and the tombstone can be deleted. If the segment still exists in the log, then the tombstone must be preserved.

5 Two-level Cleaning

Almost all of the overhead for log-structured memory is due to cleaning. Allocating new storage is trivial; new objects are simply appended at the end of the head segment. However, reclaiming free space is much more expensive. It requires running the log cleaner, which will have to copy live data out of the segments it chooses for cleaning as described in Section 3. Unfortunately, the cost of log cleaning rises rapidly as memory utilization in-

creases. For example, if segments are cleaned when 80% of their data are still live, the cleaner must copy 8 bytes of live data for every 2 bytes it frees. At 90% utilization, the cleaner must copy 9 bytes of live data for every 1 byte freed. Eventually the system will run out of bandwidth and write throughput will be limited by the speed of the cleaner. Techniques like cost-benefit segment selection [32] help by skewing the distribution of free space, so that segments chosen for cleaning have lower utilization than the overall average, but they cannot eliminate the fundamental tradeoff between utilization and cleaning cost. Any copying storage allocator will suffer from intolerable overheads as utilization approaches 100%.

Originally, disk and memory cleaning were tied together in RAMCloud: cleaning was first performed on segments in memory, then the results were reflected to the backup copies on disk. This made it impossible to achieve both high memory utilization and high write throughput. For example, if we used memory at high utilization (80-90%) write throughput would be severely limited by the cleaner's usage of disk bandwidth (see Section 8). On the other hand, we could have improved write bandwidth by increasing the size of the disk log to reduce its average utilization. For example, at 50% disk utilization we could achieve high write throughput. Furthermore, disks are cheap enough that the cost of the extra space would not be significant. However, disk and memory were fundamentally tied together: if we reduced the utilization of disk space, we would also have reduced the utilization of DRAM, which was unacceptable.

The solution is to clean the disk and memory logs independently – we call this *two-level cleaning*. With two-level cleaning, memory can be cleaned without reflecting the updates on backups. As a result, memory can have higher utilization than disk. The cleaning cost for memory will be high, but DRAM can easily provide the bandwidth required to clean at 90% utilization or higher. Disk cleaning happens less often. The disk log becomes larger than the in-memory log, so it has lower overall utilization, and this reduces the bandwidth required for cleaning.

The first level of cleaning, called *segment compaction*, operates only on the in-memory segments on masters and consumes no network or disk I/O. It compacts a single segment at a time, copying its live data into a smaller region of memory and freeing the original storage for new segments. Segment compaction maintains the same logical log in memory and on disk: each segment in memory still has a corresponding segment on disk. However, the segment in memory takes less space because deleted objects and obsolete tombstones were removed (Figure 4).

The second level of cleaning is just the mechanism described in Section 3. We call this *combined cleaning* because it cleans both disk and memory together. Segment compaction makes combined cleaning more efficient by postponing it. The effect of cleaning a segment later is that more objects have been deleted, so the segment's utilization will be lower. The result is that when combined cleaning does happen, less bandwidth is required to reclaim the same amount of free space. For example, if the disk log is allowed to grow until it consumes twice as much space as the log in memory, the utilization of segments cleaned on disk will never be greater than 50%, which makes cleaning relatively efficient.

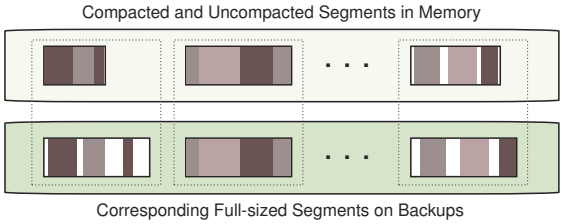


Figure 4: Compacted segments in memory have variable length because unneeded objects and tombstones have been removed, but the corresponding segments on disk remain full-size. As a result, the utilization of memory is higher than that of disk, and disk can be cleaned more efficiently.

Two-level cleaning leverages the strengths of memory and disk to compensate for their weaknesses. For memory, space is precious but bandwidth for cleaning is plentiful, so we use extra bandwidth to enable higher utilization. For disk, space is plentiful but bandwidth is precious, so we use extra space to save bandwidth.

5.1 Seglets

In the absence of segment compaction, all segments are the same size, which makes memory management simple. With compaction, however, segments in memory can have different sizes. One possible solution is to use a standard heap allocator to allocate segments, but this would result in the fragmentation problems described in Section 2. Instead, each RAMCloud master divides its log memory into fixed-size 64 KB *seglets*. A segment consists of a collection of seglets, and the number of seglets varies with the size of the segment. Because seglets are fixed-size, they introduce a small amount of internal fragmentation (one-half seglet for each segment, on average). In practice, fragmentation should be less than 1% of memory space, since we expect compacted segments to average at least half the length of a full-size segment. In addition, seglets require extra mechanism to handle log entries that span discontiguous seglets (before seglets, log entries were always contiguous).

5.2 When to Clean on Disk?

Two-level cleaning introduces a new policy question: when should the system choose memory compaction over combined cleaning, and vice-versa? This choice has an important impact on system performance because combined cleaning consumes precious disk and network I/O resources. However, as we explain below, memory compaction is not always more efficient. This section explains how these considerations resulted in RAMCloud's current

policy module; we refer to it as the *balancer*. For a more complete discussion of the balancer, see [33].

There is no point in running either cleaner until the system is running low on memory or disk space. The reason is that cleaning early is never cheaper than cleaning later on. The longer the system delays cleaning, the more time it has to accumulate dead objects, which lowers the fraction of live data in segments and makes them less expensive to clean.

The balancer determines that memory is running low as follows. Let L be the fraction of all memory occupied by live objects and F be the fraction of memory in unallocated seglets. One of the cleaners will run whenever $F \leq min(0.1, (1 - L)/2)$ In other words, cleaning occurs if the unallocated seglet pool has dropped to less than 10% of memory and at least half of the free memory is in active segments (vs. unallocated seglets). This formula represents a tradeoff: on the one hand, it delays cleaning to make it more efficient; on the other hand, it starts cleaning soon enough for the cleaner to collect free memory before the system runs out of unallocated seglets.

Given that the cleaner must run, the balancer must choose which cleaner to use. In general, compaction is preferred because it is more efficient, but there are two cases in which the balancer must choose combined cleaning. The first is when too many tombstones have accumulated. The problem with tombstones is that memory compaction alone cannot remove them: the combined cleaner must first remove dead objects from disk before their tombstones can be erased. As live tombstones pile up, segment utilizations increase and compaction becomes more and more expensive. Eventually, tombstones would eat up all free memory. Combined cleaning ensures that tombstones do not exhaust memory and makes future compactions more efficient.

The balancer detects tombstone accumulation as follows. Let T be the fraction of memory occupied by live tombstones, and L be the fraction of live objects (as above). Too many tombstones have accumulated once $T/(1 - L) \geq 40\%$. In other words, there are too many tombstones when they account for 40% of the freeable space in a master ($1 - L$; i.e., all tombstones and dead objects). The 40% value was chosen empirically based on measurements of different workloads, object sizes, and amounts of available disk bandwidth. This policy tends to run the combined cleaner more frequently under workloads that make heavy use of small objects (tombstone space accumulates more quickly as a fraction of freeable space, because tombstones are nearly as large as the objects they delete).

The second reason the combined cleaner must run is to bound the growth of the on-disk log. The size must be limited both to avoid running out of disk space and to keep crash recovery fast (since the entire log must be replayed, its size directly affects recovery speed). RAMCloud implements a configurable *disk expansion factor* that sets the maximum on-disk log size as a multiple of the in-memory log size. The combined cleaner runs when the on-disk log size exceeds 90% of this limit.

Finally, the balancer chooses memory compaction when unallocated memory is low and combined cleaning is not needed (disk space is not low and tombstones have not accumulated yet).

6 Parallel Cleaning

Two-level cleaning reduces the cost of combined cleaning, but it adds a significant new cost in the form of segment compaction. Fortunately, the cost of cleaning can be hidden by performing both combined cleaning and segment compaction concurrently with normal read and write requests. RAMCloud employs multiple cleaner threads simultaneously to take advantage of multi-core CPUs.

Parallel cleaning in RAMCloud is greatly simplified by the use of a log structure and simple metadata. For example, since segments are immutable after they are created, the cleaner need not worry about objects being modified while the cleaner is copying them. Furthermore, the hash table provides a simple way of redirecting references to objects that are relocated by the cleaner (all objects are accessed indirectly through it). This means that the basic cleaning mechanism is very straightforward: the cleaner copies live data to new segments, atomically updates references in the hash table, and frees the cleaned segments.

There are three points of contention between cleaner threads and service threads handling read and write requests. First, both cleaner and service threads need to add data at the head of the log. Second, the threads may conflict in updates to the hash table. Third, the cleaner must not free segments that are still in use by service threads. These issues and their solutions are discussed in the subsections below.

6.1 Concurrent Log Updates

The most obvious way to perform cleaning is to copy the live data to the head of the log. Unfortunately, this would create contention for the log head between cleaner threads and service threads that are writing new data.

RAMCloud's solution is for the cleaner to write survivor data to different segments than the log head. Each cleaner thread allocates a separate set of segments for its survivor data. Synchronization is required when allocating segments, but once segments are allocated, each cleaner thread can copy data to its own survivor segments without additional synchronization. Meanwhile, request-processing threads can write new data to the log head. Once a cleaner thread finishes a cleaning pass, it arranges for its survivor segments to be included in the next log digest, which inserts them into the log; it also arranges for the cleaned segments to be dropped from the next digest.

Using separate segments for survivor data has the additional benefit that the replicas for survivor segments will be stored on a different set of backups than the replicas

of the head segment. This allows the survivor segment replicas to be written in parallel with the log head replicas without contending for the same backup disks, which increases the total throughput for a single master.

6.2 Hash Table Contention

The main source of thread contention during cleaning is the hash table. This data structure is used both by service threads and cleaner threads, as it indicates which objects are alive and points to their current locations in the in-memory log. The cleaner uses the hash table to check whether an object is alive (by seeing if the hash table currently points to that exact object). If the object is alive, the cleaner copies it and updates the hash table to refer to the new location in a survivor segment. Meanwhile, service threads may be using the hash table to find objects during read requests and they may update the hash table during write or delete requests. To ensure consistency while reducing contention, RAMCloud currently uses fine-grained locks on individual hash table buckets. In the future we plan to explore lockless approaches to eliminate this overhead.

6.3 Freeing Segments in Memory

Once a cleaner thread has cleaned a segment, the segment's storage in memory can be freed for reuse. At this point, future service threads will not use data in the cleaned segment, because there are no hash table entries pointing into it. However, it could be that a service thread began using the data in the segment before the cleaner updated the hash table; if so, the cleaner must not free the segment until the service thread has finished using it.

To solve this problem, RAMCloud uses a simple mechanism similar to RCU's [27] *wait-for-readers* primitive and Tornado/K42's *generations* [6]: after a segment has been cleaned, the system will not free it until all RPCs currently being processed complete. At this point it is safe to reuse the segment's memory, since new RPCs cannot reference the segment. This approach has the advantage of not requiring additional locks for normal reads and writes.

6.4 Freeing Segments on Disk

Once a segment has been cleaned, its replicas on backups must also be freed. However, this must not be done until the corresponding survivor segments have been safely incorporated into the on-disk log. This takes two steps. First, the survivor segments must be fully replicated on backups. Survivor segments are transmitted to backups asynchronously during cleaning, so at the end of each cleaning pass the cleaner must wait for all of its survivor segments to be received by backups. Second, a new log digest must be written, which includes the survivor segments and excludes the cleaned segments. Once the digest has been durably written to backups, RPCs are issued to free the replicas for the cleaned segments.

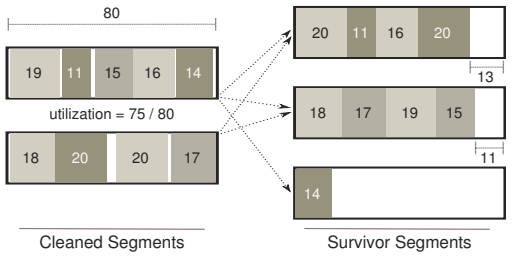


Figure 5: A simplified situation in which cleaning uses more space than it frees. Two 80-byte segments at about 94% utilization are cleaned: their objects are reordered by age (not depicted) and written to survivor segments. The label in each object indicates its size. Because of fragmentation, the last object (size 14) overflows into a third survivor segment.

7 Avoiding Cleaner Deadlock

Since log cleaning copies data before freeing it, the cleaner must have free memory space to work with before it can generate more. If there is no free memory, the cleaner cannot proceed and the system will deadlock. RAMCloud increases the risk of memory exhaustion by using memory at high utilization. Furthermore, it delays cleaning as long as possible in order to allow more objects to be deleted. Finally, two-level cleaning allows tombstones to accumulate, which consumes even more free space. This section describes how RAMCloud prevents cleaner deadlock while maximizing memory utilization.

The first step is to ensure that there are always free seglets for the cleaner to use. This is accomplished by reserving a special pool of seglets for the cleaner. When seglets are freed, they are used to replenish the cleaner pool before making space available for other uses.

The cleaner pool can only be maintained if each cleaning pass frees as much space as it uses; otherwise the cleaner could gradually consume its own reserve and then deadlock. However, RAMCloud does not allow objects to cross segment boundaries, which results in some wasted space at the end of each segment. When the cleaner reorganizes objects, it is possible for the survivor segments to have greater fragmentation than the original segments, and this could result in the survivors taking more total space than the original segments (see Figure 5).

To ensure that the cleaner always makes forward progress, it must produce at least enough free space to compensate for space lost to fragmentation. Suppose that N segments are cleaned in a particular pass and the fraction of free space in these segments is F; furthermore, let S be the size of a full segment and O the maximum object size. The cleaner will produce $NS(1-F)$ bytes of live data in this pass. Each survivor segment could contain as little as $S-O+1$ bytes of live data (if an object of size O couldn't quite fit at the end of the segment), so the maximum number of survivor segments will be $\lceil \frac{NS(1-F)}{S-O+1} \rceil$. The last seglet of each survivor segment could be empty except for a single byte, resulting in almost a full seglet of

CPU	Xeon X3470 (4x2.93 GHz cores, 3.6 GHz Turbo)
RAM	24 GB DDR3 at 800 MHz
Flash Disks	2x Crucial M4 SSDs CT128M4SSD2 (128 GB)
NIC	Mellanox ConnectX-2 Infiniband HCA
Switch	Mellanox SX6036 (4X FDR)

Table 2: The server hardware configuration used for benchmarking. All nodes ran Linux 2.6.32 and were connected to an Infiniband fabric.

fragmentation for each survivor segment. Thus, F must be large enough to produce a bit more than one seglet's worth of free data for each survivor segment generated. For RAMCloud, we conservatively require 2% of free space per cleaned segment, which is a bit more than two seglets. This number could be reduced by making seglets smaller.

There is one additional problem that could result in memory deadlock. Before freeing segments after cleaning, RAMCloud must write a new log digest to add the survivors to the log and remove the old segments. Writing a new log digest means writing a new log head segment (survivor segments do not contain digests). Unfortunately, this consumes yet another segment, which could contribute to memory exhaustion. Our initial solution was to require each cleaner pass to produce enough free space for the new log head segment, in addition to replacing the segments used for survivor data. However, it is hard to guarantee "better than break-even" cleaner performance when there is very little free space.

The current solution takes a different approach: it reserves two special *emergency head segments* that contain only log digests; no other data is permitted. If there is no free memory after cleaning, one of these segments is allocated for the head segment that will hold the new digest. Since the segment contains no objects or tombstones, it does not need to be cleaned; it is immediately freed when the next head segment is written (the emergency head is not included in the log digest for the next head segment). By keeping two emergency head segments in reserve, RAMCloud can alternate between them until a full segment's worth of space is freed and a proper log head can be allocated. As a result, each cleaner pass only needs to produce as much free space as it uses.

By combining these techniques, RAMCloud can guarantee deadlock-free cleaning with total memory utilization as high as 98%. When utilization reaches this limit, no new data (or tombstones) can be appended to the log until the cleaner has freed space. However, RAMCloud sets a lower utilization limit for writes, in order to reserve space for tombstones. Otherwise all available log space could be consumed with live data and there would be no way to add tombstones to delete objects.

8 Evaluation

All of the features described in the previous sections are implemented in RAMCloud version 1.0, which was released in January, 2014. This section describes a series of experiments we ran to evaluate log-structured memory and its implementation in RAMCloud. The key results are:

- RAMCloud supports memory utilizations of 80-90% without significant loss in performance.
- At high memory utilizations, two-level cleaning improves client throughput up to 6x over a single-level approach.
- Log-structured memory also makes sense for other DRAM-based storage systems, such as memcached.
- RAMCloud provides a better combination of durability and performance than other storage systems such as HyperDex and Redis.

Note: all plots in this section show the average of 3 or more runs, with error bars for minimum and maximum values.

8.1 Performance vs. Utilization

The most important metric for log-structured memory is how it performs at high memory utilization. In Section 2 we found that other allocators could not achieve high memory utilization in the face of changing workloads. With log-structured memory, we can choose any utilization up to the deadlock limit of about 98% described in Section 7. However, system performance will degrade as memory utilization increases; thus, the key question is how efficiently memory can be used before performance drops significantly. Our hope at the beginning of the project was that log-structured memory could support memory utilizations in the range of 80-90%.

The measurements in this section used an 80-node cluster of identical commodity servers (see Table 2). Our primary concern was the throughput of a single master, so we divided the cluster into groups of five servers and used different groups to measure different data points in parallel. Within each group, one node ran a master server, three nodes ran backups, and the last node ran the coordinator and client benchmark. This configuration provided each master with about 700 MB/s of back-end bandwidth. In an actual RAMCloud system the back-end bandwidth available to one master could be either more or less than this; we experimented with different back-end bandwidths and found that it did not change any of our conclusions. Each byte stored on a master was replicated to three different backups for durability.

All of our experiments used a maximum of two threads for cleaning. Our cluster machines have only four cores, and the main RAMCloud server requires two of them, so there were only two cores available for cleaning (we have not yet evaluated the effect of hyperthreading on RAMCloud's throughput or latency).

In each experiment, the master was given 16 GB of log space and the client created objects with sequential keys until it reached a target memory utilization; then it over-

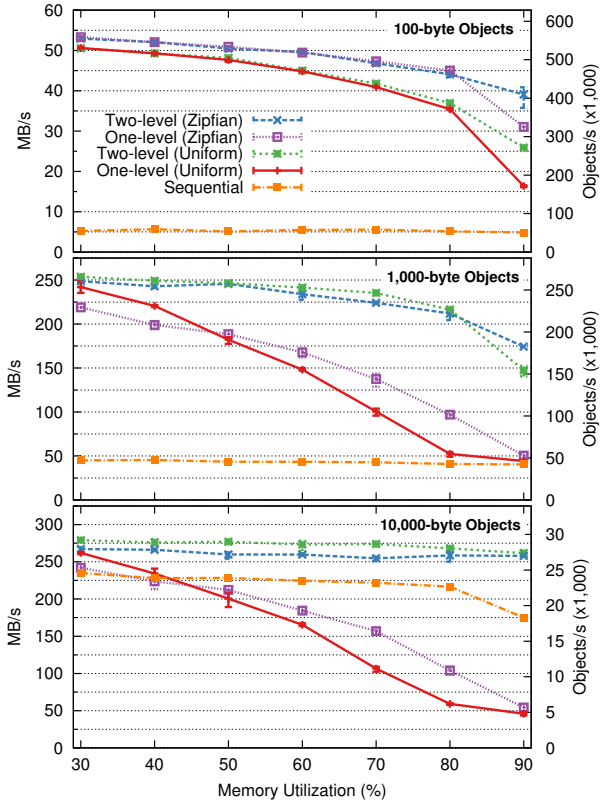


Figure 6: End-to-end client write performance as a function of memory utilization. For some experiments two-level cleaning was disabled, so only the combined cleaner was used. The "Sequential" curve used two-level cleaning and uniform access patterns with a single outstanding write request at a time. All other curves used the high-stress workload with concurrent multi-writes. Each point is averaged over 3 runs on different groups of servers.

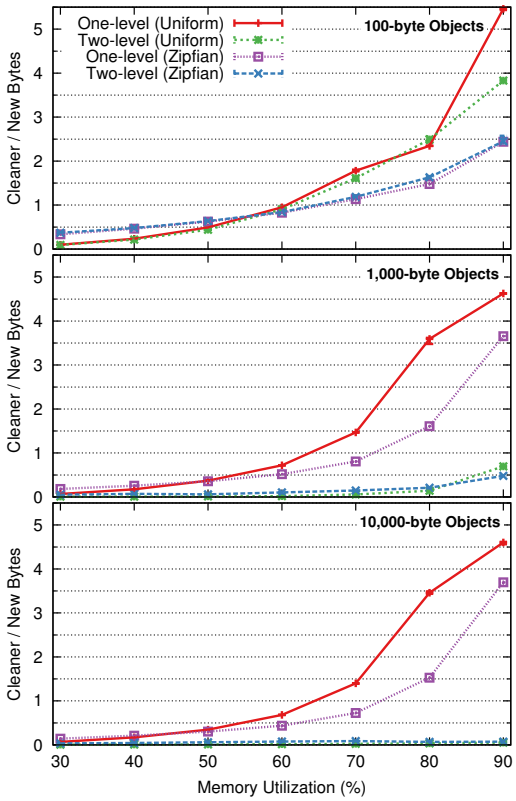


Figure 7: Cleaner bandwidth overhead (ratio of cleaner bandwidth to regular log write bandwidth) for the workloads in Figure 6. 1 means that for every byte of new data written to backups, the cleaner writes 1 byte of live data to backups while freeing segment space. The optimal ratio is 0.

wrote objects (maintaining a fixed amount of live data continuously) until the overhead for cleaning converged to a stable value.

We varied the workload in four ways to measure system behavior under different operating conditions:

1. Object Size: RAMCloud's performance depends on average object size (e.g., per-object overheads versus memory copying overheads), but not on the exact size distribution (see Section 8.5 for supporting evidence). Thus, unless otherwise noted, the objects for each test had the same fixed size. We ran different tests with sizes of 100, 1000, 10000, and 100,000 bytes (we omit the 100 KB measurements, since they were nearly identical to 10 KB).

2. Memory Utilization: The percentage of DRAM used for holding live data (not including tombstones) was fixed in each test. For example, at 50% and 90% utilization there were 8 GB and 14.4 GB of live data, respectively. In some experiments, total memory utilization was significantly higher than the listed number due to an accumulation of tombstones.

3. Locality: We ran experiments with both uniform random overwrites of objects and a Zipfian distribution in which 90% of writes were made to 15% of the objects. The uniform random case represents a workload with no locality; Zipfian represents locality similar to what has been observed in memcached deployments [7].

4. Stress Level: For most of the tests we created an artificially high workload in order to stress the master to its limit. To do this, the client issued write requests asynchronously, with 10 requests outstanding at any given time. Furthermore, each request was a multi-write containing 75 individual writes. We also ran tests where the client issued one synchronous request at a time, with a single write operation in each request; these tests are labeled "Sequential" in the graphs.

Figure 6 graphs the overall throughput of a RAMCloud master with different memory utilizations and workloads. With two-level cleaning enabled, client throughput drops only 10-20% as memory utilization increases from 30% to 80%, even with an artificially high workload. Throughput drops more significantly at 90% utilization: in the worst case (small objects with no locality), throughput at 90% utilization is about half that at 30%. At high utilization the cleaner is limited by disk bandwidth and cannot keep up with write traffic; new writes quickly exhaust all available segments and must wait for the cleaner.

These results exceed our original performance goals for RAMCloud. At the start of the project, we hoped that each RAMCloud server could support 100K small writes per second, out of a total of one million small operations per second. Even at 90% utilization, RAMCloud can support almost 410K small writes per second with some locality and nearly 270K with no locality.

If actual RAMCloud workloads are similar to our "Sequential" case, then it should be reasonable to run RAMCloud clusters at 90% memory utilization (for 100 and 1,000B objects there is almost no performance degradation). If workloads include many bulk writes, like most of the measurements in Figure 6, then it makes more sense to run at 80% utilization: the higher throughput will more than offset the 12.5% additional cost for memory.

Compared to the traditional storage allocators measured in Section 2, log-structured memory permits significantly higher memory utilization.

8.2 Two-Level Cleaning

Figure 6 also demonstrates the benefits of two-level cleaning. The figure contains additional measurements in which segment compaction was disabled ("One-level"); in these experiments, the system used RAMCloud's original one-level approach where only the combined cleaner ran. The two-level cleaning approach provides a considerable performance improvement: at 90% utilization, client throughput is up to 6x higher with two-level cleaning than single-level cleaning.

One of the motivations for two-level cleaning was to reduce the disk bandwidth used by cleaning, in order to make more bandwidth available for normal writes. Figure 7 shows that two-level cleaning reduces disk and network bandwidth overheads at high memory utilizations. The greatest benefits occur with larger object sizes, where two-level cleaning reduces overheads by 7-87x. Compaction is much more efficient in these cases because there are fewer objects to process.

8.3 CPU Overhead of Cleaning

Figure 8 shows the CPU time required for cleaning in two of the workloads from Figure 6. Each bar represents the average number of fully active cores used for combined cleaning and compaction in the master, as well as for backup RPC and disk I/O processing in the backups.

At low memory utilization a master under heavy load uses about 30-50% of one core for cleaning; backups account for the equivalent of at most 60% of one core across all six of them. Smaller objects require more CPU time for cleaning on the master due to per-object overheads, while larger objects stress backups more because the master can write up to 5 times as many megabytes per second (Figure 6). As free space becomes more scarce, the two cleaner threads are eventually active nearly all of the time. In the 100B case, RAMCloud's balancer prefers to run combined cleaning due to the accumulation of tombstones. With larger objects compaction tends to be more efficient, so combined cleaning accounts for only a small fraction of the CPU time.

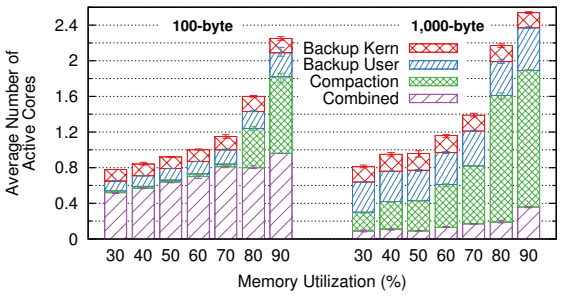


Figure 8: CPU overheads for two-level cleaning under the 100 and 1,000-byte Zipfian workloads in Figure 6, measured in average number of active cores. "Backup Kern" represents kernel time spent issuing I/Os to disks, and "Backup User" represents time spent servicing segment write RPCs on backup servers. Both of these bars are aggregated across all backups, and include traffic for normal writes as well as cleaning. "Compaction" and "Combined" represent time spent on the master in memory compaction and combined cleaning. Additional core usage unrelated to cleaning is not depicted. Each bar is averaged over 3 runs.

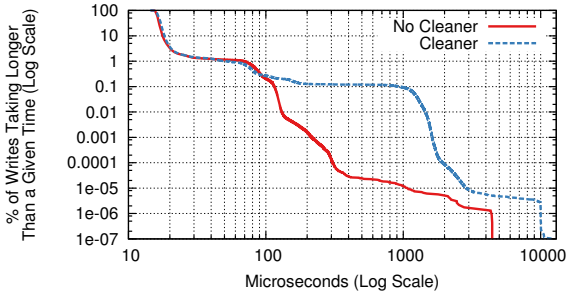


Figure 9: Reverse cumulative distribution of client write latencies when a single client issues back-to-back write requests for 100-byte objects using the uniform distribution. The "No cleaner" curve was measured with cleaning disabled. The "Cleaner" curve shows write latencies at 90% memory utilization with cleaning enabled. For example, about 10% of all write requests took longer than 18μs in both cases; with cleaning enabled, about 0.1% of all write requests took 1ms or more. The median latency was 16.70μs with cleaning enabled and 16.35μs with the cleaner disabled.

8.4 Can Cleaning Costs be Hidden?

One of the goals for RAMCloud's implementation of log-structured memory was to hide the cleaning costs so they don't affect client requests. Figure 9 graphs the latency of client write requests in normal operation with a cleaner running, and also in a special setup where the cleaner was disabled. The distributions are nearly identical up to about the 99.9th percentile, and cleaning only increased the median latency by 2% (from 16.35 to 16.70μs). About 0.1% of write requests suffer an additional 1ms or greater delay when cleaning. Preliminary

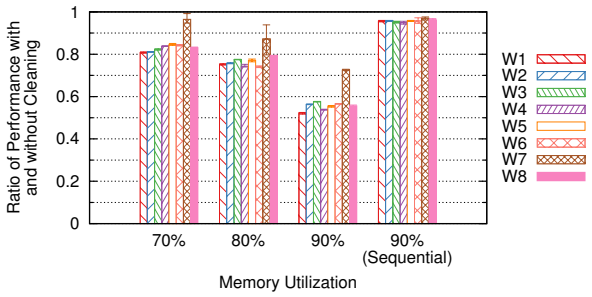


Figure 10: Client performance in RAMCloud under the same workloads as in Figure 1 from Section 2. Each bar measures the performance of a workload (with cleaning enabled) relative to the performance of the same workload with cleaning disabled. Higher is better and 1.0 is optimal; it means that the cleaner has no impact on the processing of normal requests. As in Figure 1, 100 GB of allocations were made and at most 10 GB of data was alive at once. The 70%, 80%, and 90% utilization bars were measured with the high-stress request pattern using concurrent multi-writes. The "Sequential" bars used a single outstanding write request at a time; the data size was scaled down by a factor of 10x for these experiments to make running times manageable. The master in these experiments ran on the same Xeon E5-2670 system as in Table 1.

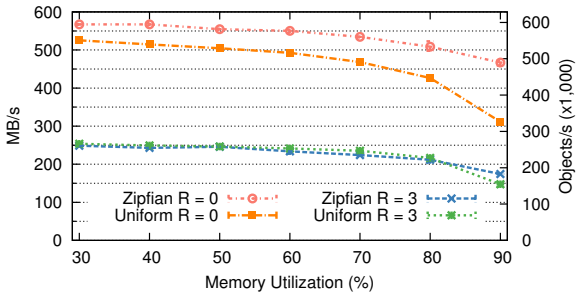


Figure 11: Two-level cleaning with (R = 3) and without replication (R = 0) for 1000-byte objects. The two lower curves are the same as in Figure 6.

Allocator	Fixed 25-byte	Zipfian 0 - 8 KB
Slab	8737	982
Log	11411	1125
Improvement	30.6%	14.6%

Table 3: Average number of objects stored per megabyte of cache in memcached, with its normal slab allocator and with a log-structured allocator. The "Fixed" column shows savings from reduced metadata (there is no fragmentation, since the 25-byte objects fit perfectly in one of the slab allocator's buckets). The "Zipfian" column shows savings from eliminating internal fragmentation in buckets. All experiments ran on a 16-core E5-2670 system with both client and server on the same machine to minimize network overhead. Memcached was given 2 GB of slab or log space for storing objects, and the slab rebalancer was enabled. YCSB [15] was used to generate the access patterns. Each run wrote 100 million objects with Zipfian-distributed key popularity and either fixed 25-byte or Zipfian-distributed sizes between 0 and 8 KB. Results were averaged over 5 runs.

experiments both with larger pools of backups and with replication disabled (not depicted) suggest that these delays are primarily due to contention for the NIC and RPC queueing delays in the single-threaded backup servers.

8.5 Performance Under Changing Workloads

Section 2 showed that changing workloads caused poor memory utilization in traditional storage allocators. For comparison, we ran those same workloads on RAMCloud, using the same general setup as for earlier experiments. The results are shown in Figure 10 (this figure is formatted differently than Figure 1 in order to show RAMCloud's performance as a function of memory utilization). We expected these workloads to exhibit performance similar to the workloads in Figure 6 (i.e. we expected the performance to be determined by the average object sizes and access patterns; workload changes per se should have no impact). Figure 10 confirms this hypothesis: with the high-stress request pattern, performance degradation due to cleaning was 10-20% at 70% utilization and 40-50% at 90% utilization. With the "Sequential" request pattern, performance degradation was 5% or less, even at 90% utilization.

8.6 Other Uses for Log-Structured Memory

Our implementation of log-structured memory is tied to RAMCloud's distributed replication mechanism, but we believe that log-structured memory also makes sense in other environments. To demonstrate this, we performed two additional experiments.

First, we re-ran some of the experiments from Figure 6 with replication disabled in order to simulate a DRAM-only storage system. We also disabled compaction (since there is no backup I/O to conserve) and had the server run the combined cleaner on in-memory segments only. Figure 11 shows that without replication, log-structured memory supports significantly higher throughput: RAMCloud's single writer thread scales to nearly 600K 1,000-byte operations per second. Under very high memory pressure throughput drops by 20-50% depending on access locality. At this object size, one writer thread and two cleaner threads suffice to handle between one quarter and one half of a 10 gigabit Ethernet link's worth of write requests.

Second, we modified the popular memcached [2] 1.4.15 object caching server to use RAMCloud's log and cleaner instead of its slab allocator. To make room for new cache entries, we modified the log cleaner to evict cold objects as it cleaned, rather than using memcached's slab-based LRU lists. Our policy was simple: segments

Allocator	Throughput (Writes/s x1000)	% CPU Cleaning
Slab	259.9 ± 0.6	0%
Log	268.0 ± 0.6	5.37 ± 0.3 %

Table 4: Average throughput and percentage of CPU used for cleaning under the same Zipfian write-only workload as in Table 3. Results were averaged over 5 runs.

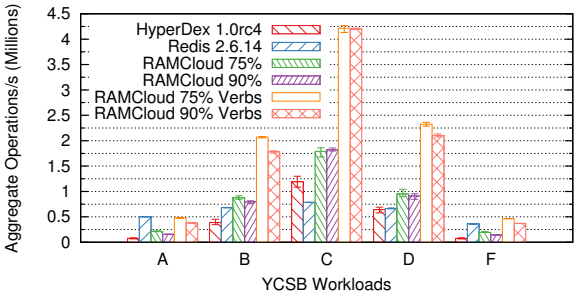


Figure 12: Performance of HyperDex, RAMCloud, and Redis under the default YCSB [15] workloads B, C, and D are read-heavy workloads, while A and F are write-heavy; workload E was omitted because RAMCloud does not support scans. Y-values represent aggregate average throughput of 24 YCSB clients running on 24 separate nodes (see Table 2). Each client performed 100 million operations on a data set of 100 million keys. Objects were 1 KB each (the workload default). An additional 12 nodes ran the storage servers. HyperDex and Redis used kernel-level sockets over Infiniband. The "RAMCloud 75%" and "RAMCloud 90%" bars were measured with kernel-level sockets over Infiniband at 75% and 90% memory utilisation, respectively (each server's share of the 10 million total records corresponded to 75% or 90% of log memory). The "RAMCloud 75% Verbs" and "RAMCloud 90% Verbs" bars were measured with RAMCloud's "kernel bypass" user-level Infiniband transport layer, which uses reliably-connected queue pairs via the Infiniband "Verbs" API. Each data point is averaged over 3 runs.

were selected for cleaning based on how many recent reads were made to objects in them (fewer requests indicate colder segments). After selecting segments, 75% of their most recently accessed objects were written to survivor segments (in order of access time); the rest were discarded. Porting the log to memcached was straightforward, requiring only minor changes to the RAMCloud sources and about 350 lines of changes to memcached.

Table 3 illustrates the main benefit of log-structured memory in memcached: increased memory efficiency. By using a log we were able to reduce per-object metadata overheads by 50% (primarily by eliminating LRU list pointers, like MemC3 [20]). This meant that small objects could be stored much more efficiently. Furthermore, using a log reduced internal fragmentation: the slab allocator must pick one of several fixed-size buckets for each object, whereas the log can pack objects of different sizes into a single segment. Table 4 shows that these benefits also came with no loss in throughput and only minimal cleaning overhead.

8.7 How does RAMCloud compare to other systems?

Figure 12 compares the performance of RAMCloud to HyperDex [18] and Redis [3] using the YCSB [15] benchmark suite. All systems were configured with triple replication. Since HyperDex is a disk-based store, we configured it to use a RAM-based file system to ensure that no operations were limited by disk I/O latencies, which the other systems specifically avoid. Both RAMCloud and Redis wrote to SSDs (Redis' append-only logging mechanism was used with a 1s fsync interval). It is worth noting that Redis is distributed with jemalloc [19], whose fragmentation issues we explored in Section 2.

RAMCloud outperforms HyperDex in every case, even when running at very high memory utilization and despite configuring HyperDex so that it does not write to disks. RAMCloud also outperforms Redis, except in write-dominated workloads A and F when kernel sockets are used. In these cases RAMCloud is limited by RPC latency, rather than allocation speed. In particular, RAMCloud must wait until data is replicated to all backups before replying to a client's write request. Redis, on the other hand, offers no durability guarantee; it responds immediately and batches updates to replicas. This unsafe mode of operation means that Redis is much less reliant on RPC latency for throughput.

Unlike the other two systems, RAMCloud was optimized for high-performance networking. For fairness, the "RAMCloud 75%" and "RAMCloud 90%" bars depict performance using the same kernel-level sockets as Redis and HyperDex. To show RAMCloud's full potential, however, we also included measurements using the Infiniband "Verbs" API, which permits low-latency access to the network card without going through the kernel. This is the normal transport used in RAMCloud; it more than doubles read throughput, and matches Redis' write throughput at 75% memory utilisation (RAMCloud is 25% slower than Redis for workload A at 90% utilization). Since Redis is less reliant on latency for performance, we do not expect it to benefit substantially if ported to use the Verbs API.

9 LFS Cost-Benefit Revisited

Like LFS [32], RAMCloud's combined cleaner uses a cost-benefit policy to choose which segments to clean. However, while evaluating cleaning techniques for RAMCloud we discovered a significant flaw in the original LFS policy for segment selection. A small change to the formula for segment selection fixes this flaw and improves cleaner performance by 50% or more at high utilization under a wide range of access localities (e.g., the Zipfian and uniform access patterns in Section 8.1). This improvement applies to any implementation of log-structured storage.

LFS selected segments to clean by evaluating the following formula for each segment and choosing the segments with the highest ratios of benefit to cost:

$$\frac{benefit}{cost} = \frac{(1-u) \times objectAge}{1+u}$$

In this formula, u is the segment's utilization (fraction of data still live), and $objectAge$ is the age of the youngest data in the segment. The cost of cleaning a segment is

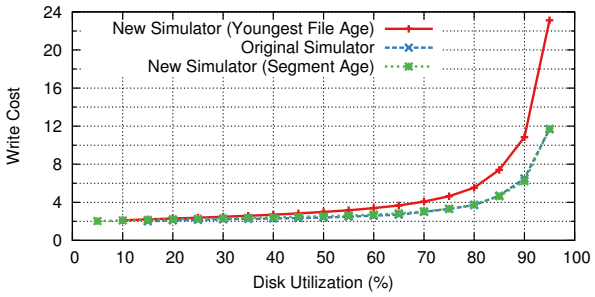


Figure 13: An original LFS simulation from [31]'s Figure 5-6 compared to results from our reimplemented simulator. The graph depicts how the I/O overhead of cleaning under a particular synthetic workload (see [31] for details) increases with disk utilization. Only by using segment age were we able to reproduce the original results (note that the bottom two lines coincide).

determined by the number of bytes that must be read or written from disk (the entire segment must be read, then the live bytes must be rewritten). The benefit of cleaning includes two factors: the amount of free space that will be reclaimed $(1 - u)$, and an additional factor intended to represent the stability of the data. If data in a segment is being overwritten rapidly then it is better to delay cleaning so that u will drop; if data in a segment is stable, it makes more sense to reclaim the free space now. $objectAge$ was used as an approximation for stability. LFS showed that cleaning can be made much more efficient by taking all these factors into account.

RAMCloud uses a slightly different formula for segment selection:

$$\frac{benefit}{cost} = \frac{(1-u) \times segmentAge}{u}$$

This differs from LFS in two ways. First, the cost has changed from $1 + u$ to u. This reflects the fact that RAMCloud keeps live segment contents in memory at all times, so the only cleaning cost is for rewriting live data.

The second change to RAMCloud's segment selection formula is in the way that data stability is estimated; this has a significant impact on cleaner performance. Using object age produces pathological cleaning behavior when there are very old objects. Eventually, some segments' objects become old enough to force the policy into cleaning the segments at extremely high utilization, which is very inefficient. Moreover, since live data is written to survivor segments in age-order (to segregate hot and cold data and make future cleaning more efficient), a vicious cycle ensues because the cleaner generates new segments with similarly high ages. These segments are then cleaned at high utilization, producing new survivors with high ages, and so on. In general, object age is not a reliable estimator of stability. For example, if objects are deleted uniform-randomly, then an objects's age provides no indication of how long it may persist.

To fix this problem, RAMCloud uses the age of the *segment*, not the age of its objects, in the formula for segment selection. This provides a better approximation to the stability of the segment's data: if a segment is very old, then its overall rate of decay must be low, otherwise its u-value would have dropped to the point of it being selected for cleaning. Furthermore, this age metric resets when a segment is cleaned, which prevents very old ages from accumulating. Figure 13 shows that this change improves overall write performance by 70% at 90% disk utilization. This improvement applies not just to RAMCloud, but to any log-structured system.

Intriguingly, although Sprite LFS used youngest object age in its cost-benefit formula, we believe that the LFS simulator, which was originally used to develop the cost-benefit policy, inadvertently used segment age instead. We reached this conclusion when we attempted to reproduce the original LFS simulation results and failed. Our initial simulation results were much worse than those reported for LFS (see Figure 13); when we switched from $objectAge$ to $segmentAge$, our simulations matched those for LFS exactly. Further evidence can be found in [26], which was based on a descendant of the original LFS simulator and describes the LFS cost-benefit policy as using the segment's age. Unfortunately, source code is no longer available for either of these simulators.

10 Future Work

There are additional opportunities to improve the performance of log-structured memory that we have not yet explored. One approach that has been used in many other storage systems is to compress the data being stored. This would allow memory to be used even more efficiently, but it would create additional CPU overheads both for reading and writing objects. Another possibility is to take advantage of periods of low load (in the middle of the night, for example) to clean aggressively in order to generate as much free space as possible; this could potentially reduce the cleaning overheads during periods of higher load.

Many of our experiments focused on worst-case synthetic scenarios (for example, heavy write loads at very high memory utilization, simple object size distributions and access patterns, etc.). In doing so we wanted to stress the system as much as possible to understand its limits. However, realistic workloads may be much less demanding. When RAMCloud begins to be deployed and used we hope to learn much more about its performance under real-world access patterns.

11 Related Work

DRAM has long been used to improve performance in main-memory database systems [17, 21], and large-scale Web applications have rekindled interest in DRAM-based storage in recent years. In addition to special-purpose systems like Web search engines [9], general-purpose storage systems like H-Store [25] and Bigtable [12] also keep part or all of their data in memory to maximize performance.

RAMCloud's storage management is superficially sim-

ilar to Bigtable [12] and its related LevelDB [4] library. For example, writes to Bigtable are first logged to GFS [22] and then stored in a DRAM buffer. Bigtable has several different mechanisms referred to as "compactions", which flush the DRAM buffer to a GFS file when it grows too large, reduce the number of files on disk, and reclaim space used by "delete entries" (analogous to tombstones in RAMCloud and called "deletion markers" in LevelDB). Unlike RAMCloud, the purpose of these compactions is not to reduce backup I/O, nor is it clear that these design choices improve memory efficiency. Bigtable does not incrementally remove delete entries from tables; instead it must rewrite them entirely. LevelDB's generational garbage collection mechanism [5], however, is more similar to RAMCloud's segmented log and cleaning.

Cleaning in log-structured memory serves a function similar to copying garbage collectors in many common programming languages such as Java and LISP [24, 37]. Section 2 has already discussed these systems.

Log-structured memory in RAMCloud was influenced by ideas introduced in log-structured file systems [32]. Much of the nomenclature and general techniques are shared (log segmentation, cleaning, and cost-benefit selection, for example). However, RAMCloud differs in its design and application. The key-value data model, for instance, allows RAMCloud to use simpler metadata structures than LFS. Furthermore, as a cluster system, RAMCloud has many disks at its disposal, which reduces contention between cleaning and regular log appends.

Efficiency has been a controversial topic in log-structured file systems [34, 35]. Additional techniques were introduced to reduce or hide the cost of cleaning [11, 26]. However, as an in-memory store, RAMCloud's use of a log is more efficient than LFS. First, RAMCloud need not read segments from disk during cleaning, which reduces cleaner I/O. Second, RAMCloud may run its disks at low utilization, making disk cleaning much cheaper with two-level cleaning. Third, since reads are always serviced from DRAM they are always fast, regardless of locality of access or placement in the log.

RAMCloud's data model and use of DRAM as the location of record for all data are similar to various "NoSQL" storage systems. Redis [3] is an in-memory store that supports a "persistence log" for durability, but does not do cleaning to reclaim free space, and offers weak durability guarantees. Memcached [2] stores all data in DRAM, but it is a volatile cache with no durability. Other NoSQL systems like Dynamo [16] and PNUTS [14] also have simplified data models, but do not service all reads from memory. HyperDex [18] offers similar durability and consistency to RAMCloud, but is a disk-based system and supports a richer data model, including range scans and efficient searches across multiple columns.

12 Conclusion

Logging has been used for decades to ensure durability and consistency in storage systems. When we began designing RAMCloud, it was a natural choice to use a logging approach on disk to back up the data stored in main memory. However, it was surprising to discover that logging also makes sense as a technique for managing the data in DRAM. Log-structured memory takes advantage of the restricted use of pointers in storage systems to eliminate the global memory scans that fundamentally limit existing garbage collectors. The result is an efficient and highly incremental form of copying garbage collector that allows memory to be used efficiently even at utilizations of 80-90%. A pleasant side effect of this discovery was that we were able to use a single technique for managing both disk and main memory, with small policy differences that optimize the usage of each medium.

Although we developed log-structured memory for RAMCloud, we believe that the ideas are generally applicable and that log-structured memory is a good candidate for managing memory in DRAM-based storage systems.

13 Acknowledgements

We would like to thank Asaf Cidon, Satoshi Matsushita, Diego Ongaro, Henry Qin, Mendel Rosenblum, Ryan Stutsman, Stephen Yang, the anonymous reviewers from FAST 2013, SOSP 2013, and FAST 2014, and our shepherd, Randy Katz, for their helpful comments. This work was supported in part by the Gigascale Systems Research Center and the Multiscale Systems Center, two of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA, and by the National Science Foundation under Grant No. 0963859. Additional support was provided by Stanford Experimental Data Center Laboratory affiliates Facebook, Mellanox, NEC, Cisco, Emulex, NetApp, SAP, Inventec, Google, VMware, and Samsung. Steve Rumble was supported by a Natural Sciences and Engineering Research Council of Canada Postgraduate Scholarship.

References

[1] Google performance tools, Mar. 2013. http://goog-perftools.sourceforge.net/.

[2] memcached: a distributed memory object caching system, Mar. 2013. http://www.memcached.org/.

[3] Redis, Mar. 2013. http://www.redis.io/.

[4] leveldb - a fast and lightweight key/value database library by google, Jan. 2014. http://code.google.com/p/leveldb/.

[5] Leveldb file layouts and compactions, Jan. 2014. http://leveldb.googlecode.com/svn/trunk/doc/impl.html.

[6] APPAVOO, J., HUI, K., SOULES, C. A. N., WISNIEWSKI, R. W., DA SILVA, D. M., KRIEGER, O., AUSLANDER, M. A., EDEL-

SOHN, D. J., GAMSA, B., GANGER, G. R., MCKENNEY, P., OSTROWSKI, M., ROSENBURG, B., STUMM, M., AND XENIDIS, J. Enabling autonomic behavior in systems software with hot swapping. *IBM Syst. J. 42*, 1 (Jan. 2003), 60–76.

[7] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 53–64.

[8] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2003), POPL '03, ACM, pp. 285–298.

[9] BARROSO, L. A., DEAN, J., AND HÖLZLE, U. Web search for a planet: The google cluster architecture. *IEEE Micro 23*, 2 (Mar. 2003), 22–28.

[10] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2000), ASPLOS IX, ACM, pp. 117–128.

[11] BLACKWELL, T., HARRIS, J., AND SELTZER, M. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX 1995 Technical Conference* (Berkeley, CA, USA, 1995), TCON'95, USENIX Association, pp. 277–288.

[12] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 205–218.

[13] CHENG, P., AND BLELLOCH, G. E. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), PLDI '01, ACM, pp. 125–136.

[14] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1* (August 2008), 1277–1288.

[15] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[16] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[17] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. A. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data* (New York, NY, USA, 1984), SIGMOD '84, ACM, pp. 1–8.

[18] ESCRIVA, R., WONG, B., AND SIRER, E. G. Hyperdex: a distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 25–36.

[19] EVANS, J. A scalable concurrent malloc (3) implementation for freebsd. In *Proceedings of the BSDCan Conference* (Apr. 2006).

[20] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Memc3: compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), NSDI'13, USENIX Association, pp. 371–384.

[21] GARCIA-MOLINA, H., AND SALEM, K. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng. 4* (December 1992), 509–516.

[22] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[23] HERTZ, M., AND BERGER, E. D. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2005), OOPSLA '05, ACM, pp. 313–326.

[24] JONES, R., HOSKING, A., AND MOSS, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management*, 1st ed. Chapman & Hall/CRC, 2011.

[25] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow. 1* (August 2008), 1496–1499.

[26] MATTHEWS, J. N., ROSELLI, D., COSTELLO, A. M., WANG, R. Y., AND ANDERSON, T. E. Improving the performance of log-structured file systems with adaptive methods. *SIGOPS Oper. Syst. Rev. 31*, 5 (Oct. 1997), 238–251.

[27] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, Oct. 1998), pp. 509–518.

[28] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), NSDI'13, USENIX Association, pp. 385–398.

[29] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 29–41.

[30] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramcloud. *Commun. ACM 54* (July 2011), 121–130.

[31] ROSENBLUM, M. *The design and implementation of a log-structured file system*. PhD thesis, Berkeley, CA, USA, 1992. UMI Order No. GAX93-30713.

[32] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10* (February 1992), 26–52.

[33] RUMBLE, S. M. *Memory and Object Management in RAMCloud*. PhD thesis, Stanford, CA, USA, 2014.

[34] SELTZER, M., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for unix. In *Proceedings of the 1993 Winter USENIX Technical Conference* (Berkeley, CA, USA, 1993), USENIX'93, USENIX Association, pp. 307–326.

[35] SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: a performance comparison. In *Proceedings of the USENIX 1995 Technical Conference* (Berkeley, CA, USA, 1995), TCON'95, USENIX Association, pp. 249–264.

[36] TENE, G., IYENGAR, B., AND WOLF, M. C4: the continuously concurrent compacting collector. In *Proceedings of the international symposium on Memory management* (New York, NY, USA, 2011), ISMM '11, ACM, pp. 79–88.

[37] WILSON, P. R. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (London, UK, UK, 1992), IWMM '92, Springer-Verlag, pp. 1–42.

[38] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association.

[39] ZORN, B. The measured cost of conservative garbage collection. *Softw. Pract. Exper. 23*, 7 (July 1993), 733–756.

Strata: Scalable High-Performance Storage on Virtualized Non-volatile Memory

Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan,
Daniel Stodden, Geoffrey Lefebvre, Daniel Ferstay, and Andrew Warfield

Coho Data
{firstname.lastname}@cohodata.com

Abstract

Strata is a commercial storage system designed around the high performance density of PCIe flash storage. We observe a parallel between the challenges introduced by this emerging flash hardware and the problems that were faced with underutilized server hardware about a decade ago. Borrowing ideas from hardware virtualization, we present a novel storage system design that partitions functionality into an address virtualization layer for high performance network-attached flash, and a hosted environment for implementing scalable protocol implementations. Our system targets the storage of virtual machine images for enterprise environments, and we demonstrate dynamic scale to over a million IO operations per second using NFSv3 in 13u of rack space, including switching.

1 Introduction

Flash-based storage devices are fast, expensive and demanding: a single device is capable of saturating a 10Gb/s network link (even for random IO), consuming significant CPU resources in the process. That same device may cost as much as (or more than) the server in which it is installed[1]. The cost and performance characteristics of fast, non-volatile media have changed the calculus of storage system design and present new challenges for building efficient and high-performance datacenter storage.

This paper describes the architecture of a commercial flash-based network-attached storage system, built using commodity hardware. In designing the system around PCIe flash, we begin with two observations about the effects of high-performance drives on large-scale storage systems. First, these devices are fast enough that in most environments, many concurrent workloads are needed to fully saturate them, and even a small degree of processing overhead will prevent full utilization. Thus, we must change our approach to the media from *aggregation* to *virtualization*. Second, aggregation is still necessary to achieve properties such as redundancy and scale. However, it must avoid the performance bottleneck that would result from the monolithic controller approach of a traditional storage array, which is designed around the obsolete assumption that media is the slowest component in the system. Further, to be practical in existing datacenter environments, we must remain compatible with existing client-side storage interfaces and support standard enterprise features like snapshots and deduplication.

In this paper we explore the implications of these two observations on the design of a scalable, high-performance NFSv3 implementation for the storage of virtual machine images. Our system is based on the building blocks of PCIe flash in commodity x86 servers connected by 10 gigabit switched Ethernet. We describe two broad technical contributions that form the basis of our design:

1. A delegated mapping and request dispatch interface from client data to physical resources through *global data address virtualization*, which allows clients to directly address data while still providing the coordination required for online data movement (e.g., in response to failures or for load balancing).
2. SDN-assisted *storage protocol virtualization* that allows clients to address a single virtual protocol gateway (e.g., NFS server) that is transparently scaled out across multiple real servers. We have built a scalable NFS server using this technique, but it applies to other protocols (such as iSCSI, SMB, and FCoE) as well.

At its core, Strata uses device-level object storage and dynamic, global address-space virtualization to achieve a clean and efficient separation between control and data paths in the storage system. Flash devices are split into

[1] Enterprise-class PCIe flash drives in the 1TB capacity range currently carry list prices in the range of $3-5K USD. Large-capacity, high-performance cards are available for list prices of up to $160K.

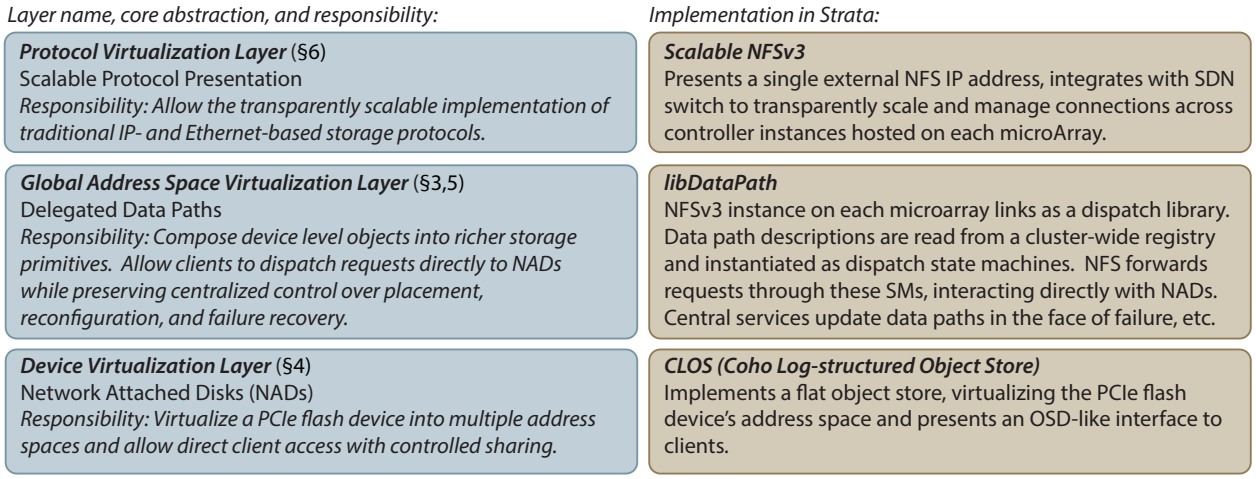


Figure 1: Strata network storage architecture.

virtual address spaces using an object storage-style interface, and clients are then allowed to directly communicate with these address spaces in a safe, low-overhead manner. In order to compose richer storage abstractions, a global address space virtualization layer allows clients to aggregate multiple per-device address spaces with mappings that achieve properties such as striping and replication. These delegated address space mappings are coordinated in a way that preserves direct client communications with storage devices, while still allowing dynamic and centralized control over data placement, migration, scale, and failure response.

Serving this storage over traditional protocols like NFS imposes a second scalability problem: clients of these protocols typically expect a single server IP address, which must be dynamically balanced over multiple servers to avoid being a performance bottleneck. In order to both scale request processing and to take advantage of full switch bandwidth between clients and storage resources, we developed a *scalable protocol presentation layer* that acts as a client to the lower layers of our architecture, and that interacts with a software-defined network switch to scale the implementation of the protocol component of a storage controller across arbitrarily many physical servers. By building protocol gateways as clients of the address virtualization layer, we preserve the ability to delegate scale-out access to device storage without requiring interface changes on the end hosts that consume the storage.

2 Architecture

The performance characteristics of emerging storage hardware demand that we completely reconsider storage architecture in order to build scalable, low-latency shared persistent memory. The reality of deployed applications is that interfaces must stay exactly the same in order for a storage system to have relevance. Strata's architecture aims to take a step toward the first of these goals, while keeping a pragmatic focus on the second.

Figure 1 characterizes the three layers of Strata's architecture. The goals and abstractions of each layer of the system are on the left-hand column, and the concrete embodiment of these goals in our implementation is on the right. At the base, we make devices accessible over an object storage interface, which is responsible for virtualizing the device's address space and allowing clients to interact with individual virtual devices. This approach reflects our view that system design for these storage devices today is similar to that of CPU virtualization ten years ago: devices provide greater performance than is required by most individual workloads and so require a lightweight interface for controlled sharing in order to allow multi-tenancy. We implement a per-device object store that allows a device to be virtualized into an address space of 2^{128} sparse objects, each of which may be up to 2^{64} bytes in size. Our implementation is similar in intention to the OSD specification, itself motivated by network attached secure disks [17]. While not broadly deployed to date, device-level object storage is receiving renewed attention today through pNFS's use of OSD as a backend, the NVMe *namespace* abstraction, and in emerging hardware such as Seagate's Kinetic drives [37]. Our object storage interface as a whole is not a significant technical contribution, but it does have some notable interface customizations described in Section 4. We refer to this layer as a *Network Attached Disk*, or *NAD*.

The middle layer of our architecture provides a global address space that supports the efficient composition of

*IO processor*s that translate client requests on a *virtual object* into operations on a set of NAD-level *physical objects*. We refer to the graph of IO processors for a particular virtual object as its *data path*, and we maintain the description of the data path for every object in a global *virtual address map*. Clients use a dispatch library to instantiate the processing graph described by each data path and perform direct IO on the physical objects at the leaves of the graph. The virtual address map is accessed through a coherence protocol that allows central services to update the data paths for virtual objects while they are in active use by clients. More concretely, data paths allow physical objects to be composed into richer storage primitives, providing properties such as striping and replication. The goal of this layer is to strike a balance between scalability and efficiency: it supports direct client access to device-level objects, without sacrificing central management of data placement, failure recovery, and more advanced storage features such as deduplication and snapshots.

Finally, the top layer performs *protocol virtualization* to allow clients to access storage over standard protocols (such as NFS) without losing the scalability of direct requests from clients to NADs. The presentation layer is tightly integrated with a 10Gb software-defined Ethernet switching fabric, allowing external clients the illusion of connecting to a single TCP endpoint, while transparently and dynamically balancing traffic to that single IP address across protocol instances on all of the NADs. Each protocol instance is a thin client of the layer below, which may communicate with other protocol instances to perform any additional synchronization required by the protocol (e.g., to maintain NFS namespace consistency).

The mapping of these layers onto the hardware that our system uses is shown in Figure 2. Requests travel from clients into Strata through an OpenFlow-enabled switch, which dispatches them according to load to the appropriate protocol handler running on a *MicroArray (µArray)* — a small host configured with flash devices and enough network and CPU to saturate them, containing the software stack representing a single NAD. For performance, each of the layers is implemented as a library, allowing a single process to handle the flow of requests from client to media. The NFSv3 implementation acts as a client of the underlying dispatch layer, which transforms requests on virtual objects into one or more requests on physical objects, issued through function calls to local physical objects and by RPC to remote objects. While the focus of the rest of this paper is on this concrete implementation of scale-out NFS, it is worth noting that the design is intended to allow applications the opportunity to link directly against the same data path library that the NFS implementation uses, resulting in a multi-tenant, multi-

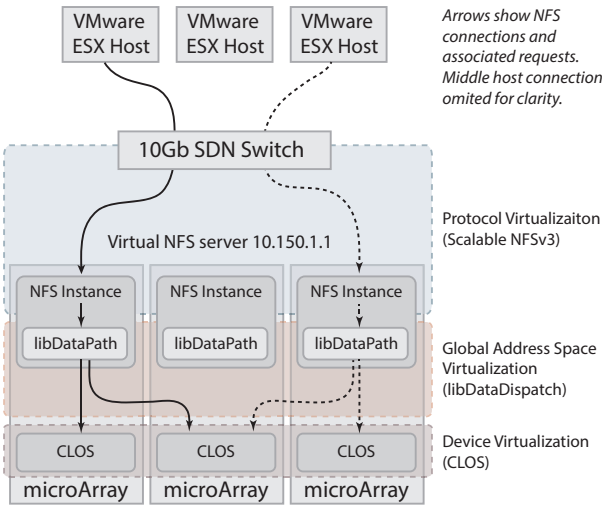


Figure 2: Hardware view of a Strata deployment

presentation storage system with a minimum of network and device-level overhead.

2.1 Scope of this Work

There are three aspects of our design that are not considered in detail within this presentation. First, we only discuss NFS as a concrete implementation of protocol virtualization. Strata has been designed to host and support multiple protocols and tenants, but our initial product release is specifically NFSv3 for VMware clients, so we focus on this type of deployment in describing the implementation. Second, Strata was initially designed to be a software layer that is co-located on the same physical servers that host virtual machines. We have moved to a separate physical hosting model where we directly build on dedicated hardware, but there is nothing that prevents the system from being deployed in a more co-located (or "converged") manner. Finally, our full implementation incorporates a tier of spinning disks on each of the storage nodes to allow cold data to be stored more economically behind the flash layer. However, in this paper we configure and describe a single-tier, all-flash system to simplify the exposition.

In the next sections we discuss three relevant aspects of Strata—address space virtualization, dynamic reconfiguration, and scalable protocol support—in more detail. We then describe some specifics of how these three components interact in our NFSv3 implementation for VM image storage before providing a performance evaluation of the system as a whole.

3 Data Paths

Strata provides a common library interface to data that underlies the higher-level, client-specific protocols described in Section 6. This library presents a notion of virtual objects, which are available cluster-wide and may comprise multiple physical objects bundled together for parallel data access, fault tolerance, or other reasons (e.g., data deduplication). The library provides a superset of the object storage interface provided by the NADs (Section 4), with additional interfaces to manage the placement of objects (and ranges within objects) across NADs, to maintain data invariants (e.g., replication levels and consistent updates) when object ranges are replicated or striped, and to coordinate both concurrent access to data and concurrent manipulation of the *virtual address maps* describing their layout.

To avoid IO bottlenecks, users of the data path interface (which may be native clients or protocol gateways such as our NFS server) access data directly. To do so, they map requests from virtual objects to physical objects using the virtual address map. This is not simply a pointer from a virtual object *(id, range)* pair to a set of physical object *(id, range)* pairs. Rather, each virtual range is associated with a particular *processor* for that range, along with processor-specific context. Strata uses a dispatch-oriented programming model in which a pipeline of operations is performed on requests as they are passed from an originating client, through a set of transformations, and eventually to the appropriate storage device(s). Our model borrows ideas from packet processing systems such as X-Kernel [19], Scout [25], and Click [21], but adapts them to a storage context, in which modules along the pipeline perform translations through a set of layered address spaces, and may fork and/or collect requests and responses as they are passed.

The dispatch library provides a collection of request processors, which can stand alone or be combined with other processors. Each processor takes a storage request (e.g., a read or write request) as input and produces one or more requests to its children. NADs expose isolated sparse objects; processors perform translations that allow multiple objects to be combined for some functional purpose, and present them as a single object, which may in turn be used by other processors. The idea of request-based address translation to build storage features has been used in other systems [24, 35, 36], often as the basis for volume management; Strata disentangles it from the underlying storage system and treats it as a first-class dispatch abstraction.

The composition of dispatch modules bears similarity to Click [21], but the application in a storage domain carries a number of differences. First, requests are generally acknowledged at the point that they reach a storage device, and so as a result they differ from packet forwarding logic in that they travel both down and then back up through a dispatch stack; processors contain logic to handle both requests *and* responses. Second, it is common for requests to be split or merged as they traverse a processor — for example, a replication processor may duplicate a request and issue it to multiple nodes, and then collect all responses before passing a single response back up to its parent. Finally, while processors describe fast, library-based request dispatching logic, they typically depend on additional facilities from the system. Strata allows processor implementations access to APIs for shared, cluster-wide state which may be used on a control path to, for instance, store replica configuration. It additionally provides facilities for background functionality such as NAD failure detection and response. The intention of the processor organization is to allow dispatch decisions to be pushed out to client implementations and be made with minimal performance impact, while still benefiting from common system-wide infrastructure for maintaining the system and responding to failures. The responsibilities of the dispatch library are described in more detail in the following subsections.

3.1 The Virtual Address Map

```
/objects/112:
    type=regular dispatch={object=111
                           type=dispatch}

/objects/111:
    type=dispatch
    stripe={stripecount=8 chunksize=524288
        0={object=103 type=dispatch}
        1={object=104 type=dispatch}}

/objects/103:
    type=dispatch
    rpl={policy=mirror storecount=2
            {storeid=a98f2... state=in-sync}
            {storeid=fc89f... state=in-sync}}
```

Figure 3: Virtual object to physical object range mapping

Figure 3 shows the relevant information stored in the virtual address map for a typical object. Each object has an identifier, a type, some type-specific context, and may contain other metadata such as cached size or modification time information (which is not canonical, for reasons discussed below).

The entry point into the virtual address map is a *regular* object. This contains no location information on its own, but delegates to a top-level *dispatch* object. In Figure 3, object 112 is a regular object that delegates to a dispatch processor whose context is identified by object 111 (the IDs are in reverse order here because the dispatch graph

is created from the bottom up, but traversed from the top down). Thus when a client opens file 112, it instantiates a dispatcher using the data in object 111 as context. This context informs the dispatcher that it will be delegating IO through a *striped* processor, using 2 stripes for the object and a stripe width of 512K. The dispatcher in turn instantiates 8 processors (one for each stripe), each configured with the information stored in the object associated with each stripe (e.g., stripe 0 uses object 103). Finally, when the stripe dispatcher performs IO on stripe 0, it will use the context in the object descriptor for object 103 to instantiate a *replicated* processor, which mirrors writes to the NADs listed in its replica set, and issues reads to the nearest in sync replica (where distance is currently simply local or remote).

In addition to the striping and mirroring processors described here, the map can support other more advanced processors, such as erasure coding, or byte-range mappings to arbitrary objects (which supports among other things data deduplication).

3.2 Dispatch

IO requests are handled by a chain of dispatchers, each of which has some common functionality. Dispatchers may have to fragment requests into pieces if they span the ranges covered by different subprocessors, or clone requests into multiple subrequests (e.g., for replication), and they must collect the results of subrequests and deal with partial failures.

The replication and striping modules included in the standard library are representative of the ways processors transform requests as they traverse a dispatch stack. The replication processor allows a request to be split and issued concurrently to a set of replica objects. The request address remains unchanged within each object, and responses are not returned until all replicas have acknowledged a request as complete. The processor prioritizes reading from local replicas, but forwards requests to remote replicas in the event of a failure (either an error response or a timeout). It imposes a global ordering on write requests and streams them to all replicas in parallel. It also periodically commits a light-weight checkpoint to each replica's log to maintain a persistent record of synchronization points; these checkpoints are used for crash recovery (Section 5.1.3).

The striping processor distributes data across a collection of sparse objects. It is parameterized to take a stripe size (in bytes) and a list of objects to act as the ordered stripe set. In the event that a request crosses a stripe boundary, the processor splits that request into a set of per-stripe requests and issues those asynchronously, collecting the responses before returning. Static, address-based striping is a relatively simple load balancing and data distribution mechanism as compared to placement schemes such as consistent hashing [20]. Our experience has been that the approach is effective, because data placement tends to be reasonably uniform within an object address space, and because using a reasonably large stripe size (we default to 512KB) preserves locality well enough to keep request fragmentation overhead low in normal operation.

3.3 Coherence

Strata clients also participate in a simple coordination protocol in order to allow the virtual address map for a virtual object to be updated even while that object is in use. Online reconfiguration provides a means for recovering from failures, responding to capacity changes, and even moving objects in response to observed or predicted load (on a device basis — this is distinct from client load balancing, which we also support through a switch-based protocol described in Section 6.2).

The virtual address maps are stored in a distributed, synchronized *configuration database* implemented over Apache Zookeeper, which is also available for any low-bandwidth synchronization required by services elsewhere in the software stack. The coherence protocol is built on top of the configuration database. It is currently optimized for a single writer per object, and works as follows: when a client wishes to write to a virtual object, it first claims a lock for it in the configuration database. If the object is already locked, the client requests that the holder release it so that the client can claim it. If the holder does not voluntarily release it within a reasonable time, the holder is considered unresponsive and fenced from the system using the mechanism described in Section 6.2. This is enough to allow movement of objects, by first creating new, out of sync physical objects at the desired location, then requesting a release of the object's lock holder if there is one. The user of the object will reacquire the lock on the next write, and in the process discover the new out of sync replica and initiate resynchronization. When the new replica is in sync, the same process may be repeated to delete replicas that are at undesirable locations.

4 Network Attached Disks

The unit of storage in Strata is a Network Attached Disk (NAD), consisting of a balanced combination of CPU, network and storage components. In our current hardware, each NAD has two 10 gigabit Ethernet ports, two PCIe flash cards capable of 10 gigabits of throughput each, and a pair of Xeon processors that can keep up with request load and host additional services alongside the data path. Each NAD provides two distinct services.

First, it efficiently multiplexes the raw storage hardware across multiple concurrent users, using an object storage protocol. Second, it hosts applications that provide higher level services over the cluster. Object rebalancing (Section 5.2.1) and the NFS protocol interface (Section 6.1) are examples of these services.

At the device level, we multiplex the underlying storage into objects, named by 128-bit identifiers and consisting of sparse 2^{64} byte data address spaces. These address spaces are currently backed by a garbage-collected log-structured object store, but the implementation of the object store is opaque to the layers above and could be replaced if newer storage technologies made different access patterns more efficient. We also provide increased capacity by allowing each object to flush low priority or infrequently used data to disk, but this is again hidden behind the object interface. The details of disk tiering, garbage collection, and the layout of the file system are beyond the scope of this paper.

The physical object interface is for the most part a traditional object-based storage device [37,38] with a CRUD interface for sparse objects, as well as a few extensions to assist with our clustering protocol (Section 5.1.2). It is significantly simpler than existing block device interfaces, such as the SCSI command set, but is also intended to be more direct and general purpose than even narrower interfaces such as those of a key-value store. Providing a low-level hardware abstraction layer allows the implementation to be customized to accommodate best practices of individual flash implementations, and also allows more dramatic design changes at the media interface level as new technologies become available.

4.1 Network Integration

As with any distributed system, we must deal with misbehaving nodes. We address this problem by tightly coupling with managed Ethernet switches, which we discuss at more length in Section 6.2. This approach borrows ideas from systems such as Sane [8] and Ethane [7], in which a managed network is used to enforce isolation between independent endpoints. The system integrates with both OpenFlow-based switches and software switching at the VMM to ensure that Strata objects are only addressable by their authorized clients.

Our initial implementation used Ethernet VLANs, because this form of hardware-supported isolation is in common use in enterprise environments. In the current implementation, we have moved to OpenFlow, which provides a more flexible tunneling abstraction for traffic isolation.

We also expose an isolated private virtual network for out-of-band control and management operations internal to the cluster. This allows NADs themselves to access remote objects for peer-wise resynchronization and reorganization under the control of a cluster monitor.

5 Online Reconfiguration

There are two broad categories of events to which Strata must respond in order to maintain its performance and reliability properties. The first category includes faults that occur directly on the data path. The dispatch library recovers from such faults immediately and automatically by reconfiguring the affected virtual objects on behalf of the client. The second category includes events such as device failures and load imbalance. These are handled by a dedicated *cluster monitor* which performs large-scale reconfiguration tasks to maintain the health of the system as a whole. In all cases, reconfiguration is performed online and has minimal impact on client availability.

5.1 Object Reconfiguration

A number of error recovery mechanisms are built directly into the dispatch library. These mechanisms allow clients to quickly recover from failures by reconfiguring individual virtual objects on the data path.

5.1.1 IO Errors

The replication IO processor responds to read errors in the obvious way: by immediately resubmitting failed requests to different replicas. In addition, clients maintain per-device error counts; if the aggregated error count for a device exceeds a configurable threshold, a background task takes the device offline and coordinates a system-wide reconfiguration (Section 5.2.2).

IO processors respond to write errors by synchronously reconfiguring virtual objects at the time of the failure. This involves three steps. First, the affected replica is marked *out of sync* in the configuration database. This serves as a global, persistent indication that the replica may not be used to serve reads because it contains potentially stale data. Second, a best-effort attempt is made to inform the NAD of the error so that it can initiate a background task to resynchronize the affected replica. This allows the system to recover from transient failures almost immediately. Finally, the IO processor allocates a special *patch* object on a separate device and adds this to the replica set. Once a replica has been marked out of sync, no further writes are issued to it until it has been resynchronized; patches prevent device failures from impeding progress by providing a temporary buffer to absorb writes under these degraded conditions. With the patch object allocated, the IO processor can continue to

meet the replication requirements for new writes while out of sync replicas are repaired in the background. A replica set remains available as long as an in sync replica or an out of sync replica *and* all of its patches are available.

5.1.2 Resynchronization

In addition to providing clients direct access to devices via virtual address maps, Strata provides a number of background services to maintain the health of individual virtual objects and the system as a whole. The most fundamental of these is the *resync* service, which provides a background task that can resynchronize objects replicated across multiple devices.

Resync is built on top of a special NAD `resync` API that exposes the underlying log structure of the object stores. NADs maintain a Log Serial Number (LSN) with every physical object in their stores; when a record is appended to an object's log, its LSN is monotonically incremented. The IO processor uses these LSNs to impose a global ordering on the changes made to physical objects that are replicated across stores and to verify that all replicas have received all updates.

If a write failure causes a replica to go out of sync, the client can request the system to resynchronize the replica. It does this by invoking the `resync` RPC on the NAD which hosts the out of sync replica. The server then starts a background task which streams the missing log records from an in sync replica and applies them to the local out of sync copy, using the LSN to identify which records the local copy is missing.

During resync, the background task has exclusive write access to the out of sync replica because all clients have been reconfigured to use patches. Thus the resync task can chase the tail of the in sync object's log while clients continue to write. When the bulk of the data has been copied, the resync task enters a final *stop-and-copy* phase in which it acquires exclusive write access to all replicas in the replica set, finalizes the resync, applies any client writes received in the interim, marks the replica as in sync in the configuration database, and removes the patch.

It is important to ensure that resync makes timely progress to limit vulnerability to data loss. Very heavy client write loads may interfere with resync tasks and, in the worst case, result in unbounded transfer times. For this reason, when an object is under resync, client writes are throttled and resync requests are prioritized.

5.1.3 Crash Recovery

Special care must be taken in the event of an unclean shutdown. On a clean shutdown, all objects are released by removing their locks from the configuration database. Crashes are detected when replica sets are discovered with stale locks (i.e., locks identifying unresponsive IO processors). When this happens, it is not safe to assume that replicas marked *in sync* in the configuration database are truly in sync, because a crash might have occured midway through a the configuration database update; instead, all the replicas in the set must be queried directly to determine their states.

In the common case, the IO processor retrieves the LSN for every replica in the set and determines which replicas, if any, are out of sync. If all replicas have the same LSN, then no resynchronization is required. If different LSNs are discovered, then the replica with the highest LSN is designated as the authoritative copy, and all other replicas are marked out of sync and resync tasks are initiated.

If a replica cannot be queried during the recovery procedure, it is marked as *diverged* in the configuration database and the replica with the highest LSN from the remaining available replicas is chosen as the authoritative copy. In this case, writes may have been committed to the diverged replica that were not committed to any others. If the diverged replica becomes available again some time in the future, these extra writes must be discarded. This is achieved by rolling the replica back to its last checkpoint and starting a resync from that point in its log. Consistency in the face of such rollbacks is guaranteed by ensuring that objects are successfully marked out of sync in the configuration database *before* writes are acknowledged to clients. Thus write failures are guaranteed to either mark replicas out of sync in the configuration database (and create corresponding patches) or propagate back to the client.

5.2 System Reconfiguration

Strata also provides a highly-available monitoring service that watches over the health of the system and coordinates system-wide recovery procedures as necessary. Monitors collect information from clients, SMART diagnostic tools, and NAD RPCs to gauge the status of the system. Monitors build on the per-object reconfiguration mechanisms described above to respond to events that individual clients don't address, such as load imbalance across the system, stores nearing capacity, and device failures.

5.2.1 Rebalance

Strata provides a rebalance facility which is capable of performing system-wide reconfiguration to repair broken replicas, prevent NADs from filling to capacity, and improve load distribution across NADs. This facility is in turn used to recover from device failures and expand onto new hardware.

Rebalance proceeds in two stages. In the first stage, the monitor retrieves the current system configuration, including the status of all NADs and virtual address map of every virtual object. It then constructs a new layout for the replicas according to a customizable placement policy. This process is scriptable and can be easily tailored to suit specific performance and durability requirements for individual deployments (see Section 7.3 for some analysis of the effects of different placement policies). The default policy uses a greedy algorithm that considers a number of criteria designed to ensure that replicated physical objects do not share fault domains, capacity imbalances are avoided as much as possible, and migration overheads are kept reasonably low. The new layout is formulated as a rebalance plan describing what changes need to be applied to individual replica sets to achieve the desired configuration.

In the second stage, the monitor coordinates the execution of the rebalance plan by initiating resync tasks on individual NADs to effect the necessary data migration. When replicas need to be moved, the migration is performed in three steps:

1. A new replica is added to the destination NAD
2. A resync task is performed to transfer the data
3. The old replica is removed from the source NAD

This requires two reconfiguration events for the replica set, the first to extend it to include the new replica, and the second to prune the original after the resync has completed. The monitor coordinates this procedure across all NADs and clients for all modified virtual objects.

5.2.2 Device Failure

Strata determines that a NAD has failed either when it receives a hardware failure notification from a responsive NAD (such as a failed flash device or excessive error count) or when it observes that a NAD has stopped responding to requests for more than a configurable timeout. In either case, the monitor responds by taking the NAD offline and initiating a system-wide reconfiguration to repair redundancy.

The first thing the monitor does when taking a NAD offline is to disconnect it from the data path VLAN. This is a strong benefit of integrating directly against an Ethernet switch in our environment: prior to taking corrective action, the NAD is synchronously disconnected from the network for all request traffic, avoiding the distributed systems complexities that stem from things such as overloaded components appearing to fail and then returning long after a timeout in an inconsistent state. Rather than attempting to use completely end-host mechanisms such as watchdogs to trigger reboots, or agreement protocols to inform all clients of a NAD's failure, Strata disables the VLAN and requires that the failed NAD reconnect on the (separate) control VLAN in the event that it returns to life in the future.

From this point, the recovery logic is straight forward. The NAD is marked as failed in the configuration database and a rebalance job is initiated to repair any replica sets containing replicas on the failed NAD.

5.2.3 Elastic Scale Out

Strata responds to the introduction of new hardware much in the same way that it responds to failures. When the monitor observes that new hardware has been installed, it uses the rebalance facility to generate a layout that incorporates the new devices. Because replication is generally configured underneath striping, we can migrate virtual objects at the granularity of individual stripes, allowing a single striped file to exploit the aggregated performance of many devices. Objects, whether whole files or individual stripes, can be moved to another NAD even while the file is online, using the existing resync mechanism. New NADs are populated in a controlled manner to limit the impact of background IO on active client workloads.

6 Storage Protocols

Strata supports legacy protocols by providing an execution runtime for hosting protocol servers. Protocols are built as thin presentation layers on top of the dispatch interfaces; multiple protocol instances can operate side by side. Implementations can also leverage SDN-based protocol scaling to transparently spread multiple clients across the distributed runtime environment.

6.1 Scalable NFS

Strata is designed so that application developers can focus primarily on implementing protocol specifications without worrying much about how to organize data on disk. We expect that many storage protocols can be implemented as thin wrappers around the provided dispatch library. Our NFS implementation, for example, maps very cleanly onto the high-level dispatch APIs, providing

only protocol-specific extensions like RPC marshalling and NFS-style access control. It takes advantage of the configuration database to store mappings between the NFS namespace and the backend objects, and it relies exclusively on the striping and replication processors to implement the data path. Moreover, Strata allows NFS servers to be instantiated across multiple backend nodes, automatically distributing the additional processing overhead across backend compute resources.

6.2 SDN Protocol Scaling

Scaling legacy storage protocols can be challenging, especially when the protocols were not originally designed for a distributed back end. Protocol scalability limitations may not pose significant problems for traditional arrays, which already sit behind relatively narrow network interfaces, but they can become a performance bottleneck in Strata's distributed architecture.

A core property that limits scale of access bandwidth of conventional IP storage protocols is the presentation of storage servers behind a single IP address. Fortunately, emerging "software defined" network (SDN) switches provide interfaces that allow applications to take more precise control over packet forwarding through Ethernet switches than has traditionally been possible.

Using the OpenFlow protocol, a software controller is able to interact with the switch by pushing flow-specific rules onto the switch's forwarding path. OpenFlow rules are effectively wild-carded packet filters and associated actions that tell a switch what to do when a matching packet is identified. SDN switches (our implementation currently uses an Arista Networks 7050T-52) interpret these flow rules and push them down onto the switch's TCAM or L2/L3 forwarding tables.

By manipulating traffic through the switch at the granularity of individual flows, Strata protocol implementations are able to present a single logical IP address to multiple clients. Rules are installed on the switch to trigger a fault event whenever a new NFS session is opened, and the resulting exception path determines which protocol instance to forward that session to initially. A service monitors network activity and migrates client connections as necessary to maintain an even workload distribution.

The protocol scaling API wraps and extends the conventional socket API, allowing a protocol implementation to bind to and listen on a shared IP address across all of its instances. The client load balancer then monitors the traffic demands across all of these connections and initiates flow migration in response to overload on any individual physical connection.

In its simplest form, client migration is handled entirely at the transport layer. When the protocol load balancer observes that a specific NAD is overloaded, it updates the routing tables to redirect the busiest client workload to a different NAD. Once the client's traffic is diverted, it receives a TCP `RST` from the new NAD and establishes a new connection, thereby transparently migrating traffic to the new NAD.

Strata also provides hooks for situations where application layer coordination is required to make migration safe. For example, our NFS implementation registers a pre-migration routine with the load balancer, which allows the source NFS server to flush any pending, non-idempotent requests (such as `create` or `remove`) before the connection is redirected to the destination server.

7 Evaluation

In this section we evaluate our system both in terms of effective use of flash resources, and as a scalable, reliable provider of storage for NFS clients. First, we establish baseline performance over a traditional NFS server on the same hardware. Then we evaluate how performance scales as nodes are added and removed from the system, using VM-based workloads over the legacy NFS interface, which is oblivious to cluster changes. In addition, we compare the effects of load balancing and object placement policy on performance. We then test reliability in the face of node failure, which is a crucial feature of any distributed storage system. We also examine the relation between CPU power and performance in our system as a demonstration of the need to balance node power between flash, network and CPU.

7.1 Test environment

Evaluation was performed on a cluster of the maximum size allowed by our 48-port switch: 12 NADs, each of which has two 10 gigabit Ethernet ports, two 800 GB Intel 910 PCIe flash cards, 6 3 TB SATA drives, 64 GB of RAM, and 2 Xen E5-2620 processors at 2 GHz with 6 cores/12 threads each, and 12 clients, in the form of Dell PowerEdge R420 servers running ESXi 5.0, with two 10 gigabit ports each, 64 GB of RAM, and 2 Xeon E5-2470 processors at 2.3 GHz with 8 cores/16 threads each. We configured the deployment to maintain two replicas of every stored object, without striping (since it unnecessarily complicates placement comparisons and has little benefit for symmetric workloads). Garbage collection is active, and the deployment is in its standard configuration with a disk tier enabled, but the workloads have been configured to fit entirely within flash, as the effects of

Server	Read IOPS	Write IOPS
Strata	40287	9960
KNFS	23377	5796

Table 1: Random IO performance on Strata versus KNFS.

cache misses to magnetic media are not relevant to this paper.

7.2 Baseline performance

To provide some performance context for our architecture versus a typical NFS implementation, we compare two minimal deployments of NFS over flash. We set Strata to serve a single flash card, with no replication or striping, and mounted it loopback. We ran a fio [34] workload with a 4K IO size 80/20 read-write mix at a queue depth of 128 against a fully allocated file. We then formatted the flash card with ext4, exported it with the linux kernel NFS server, and ran the same test. The results are in Table 1. As the table shows, we offer good NFS performance at the level of individual devices. In the following section we proceed to evaluate scalability.

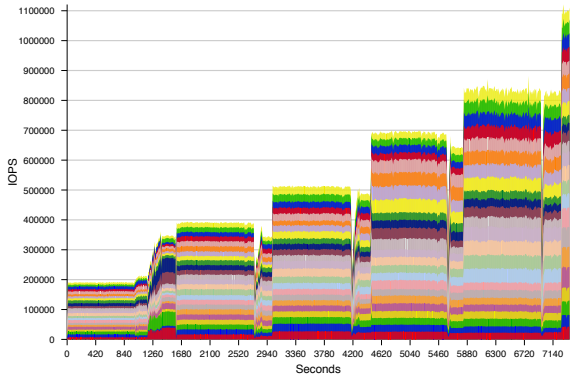


Figure 4: IOPS over time, read-only workload.

7.3 Scalability

In this section we evaluate how well performance scales as we add NADs to the cluster. We begin each test by deploying 96 VMs (8 per client) into a cluster of 2 NADs. We choose this number of VMs because ESXi limits the queue depth for a VM to 32 outstanding requests, but we do not see maximum performance until a queue depth of 128 per flash card. The VMs are each configured to run the same fio workload for a given test. In Figure 4, fio generates 4K random reads to focus on IOPS scalability. In Figure 5, fio generates an 80/20 mix of reads and writes at 128K block size in a Pareto distribution such that 80% of requests go to 20% of the data. This is meant to be more representative of real VM workloads, but with enough offered load to completely saturate the cluster.

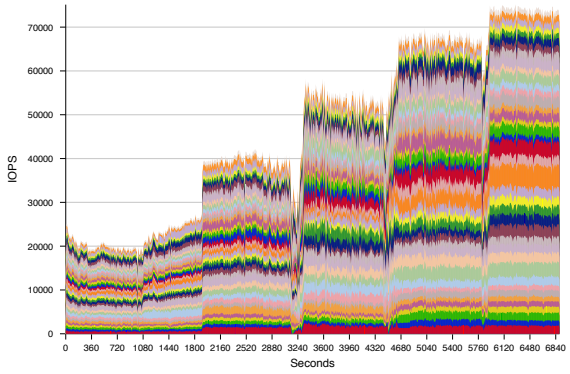


Figure 5: IOPS over time, 80/20 R/W workload.

As the tests run, we periodically add NADs, two at a time, up to a maximum of twelve[2]. When each pair of NADs comes online, a rebalancing process automatically begins to move data across the cluster so that the amount of data on each NAD is balanced. When it completes, we run in a steady state for two minutes and then add the next pair. In both figures, the periods where rebalancing is in progress are reflected by a temporary drop in performance (as the rebalance process competes with client workloads for resources), followed by a rapid increase in overall performance when the new nodes are marked available, triggering the switch to load-balance clients to them. A cluster of 12 NADs achieves over 1 million IOPS in the IOPS test, and 10 NADs achieve 70,000 IOPS (representing more than 9 gigabytes/second of throughput) in the 80/20 test.

We also test the effect of placement and load balancing on overall performance. If the location of a workload source is unpredictable (as in a VM data center with virtual machine migration enabled), we need to be able to migrate clients quickly in response to load. However, if the configuration is more static or can be predicted in advance, we may benefit from attempting to place clients and data together to reduce the network overhead incurred by remote IO requests. As discussed in Section 5.2.1, the load-balancing and data migration features of Strata make both approaches possible. Figure 4 is the result of an aggressive local placement policy, in which data is placed on the same NAD as its clients, and both are moved as the number of devices changes. This achieves the best possible performance at the cost of considerable data movement. In contrast, Figure 6 shows the

[2]ten for the read/write test due to an unfortunate test harness problem

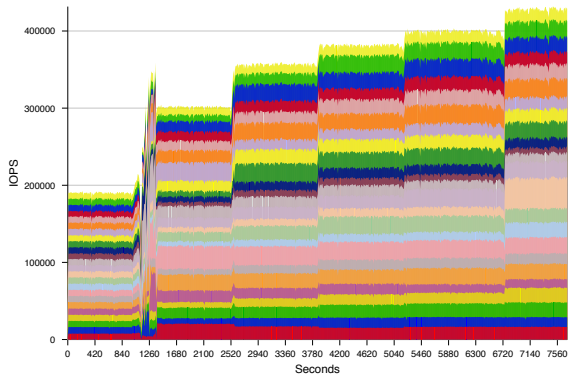


Figure 6: IOPS over time, read-only workload with random placement

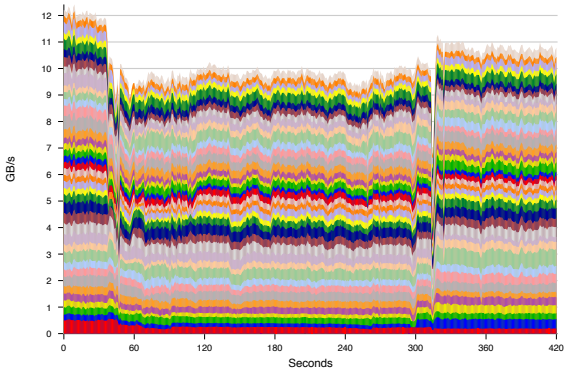


Figure 7: Aggregate bandwidth for 80/20 clients during failover and recovery

performance of an otherwise identical test configuration when data is placed randomly (while still satisfying fault tolerance and even distribution constraints), rather than being moved according to client requests. The pareto workload (Figure 5) is also configured with the default random placement policy, which is the main reason that it does not scale linearly: as the number of nodes increases, so does the probability that a request will need to be forwarded to a remote NAD.

7.4 Node Failure

As a counterpoint to the scalability tests run in the previous section, we also tested the behaviour of the cluster when a node is lost. We configured a 10 NAD cluster with 10 clients hosting 4 VMs each, running the 80/20 Pareto workload described earlier. Figure 7 shows the behaviour of the system during this experiment. After the VMs had been running for a short time, we powered off one of the NADs by IPMI, waited 60 seconds, then powered it back on. During the node outage, the system continued to run uninterrupted but with lower throughput. When the node came back up, it spent some time resynchronizing its objects to restore full replication to the system, and then rejoined the cluster. The client load balancer shifted clients onto it and throughput was restored (within the variance resulting from the client load balancer's placement decisions).

7.5 Protocol overhead

The benchmarks up to this point have all been run inside VMs whose storage is provided by a virtual disk that Strata exports by NFS to ESXi. This configuration requires no changes on the part of the clients to scale across a cluster, but does impose overheads. To quantify these overheads we wrote a custom fio engine that is capable of performing IO directly against our native dispatch interface (that is, the API by which our NFS protocol gateway interacts with the NADs). We then compared the performance of a single VM running a random 4k read fio workload (for maximum possible IOPS) against a VMDK exported by NFS to the same workload run against our native dispatch engine. In this experiment, the VMDK-based experiment produced an average of 50240 IOPS, whereas direct access achieved 54060 IOPS, for an improvement of roughly 8%.

CPU	IOPS	Freq (Cores)	Price
E5-2620	127K	2 GHz (6)	$406
E5-2640	153K (+*20%*)	2.5 GHz (6)	$885
E5-2650v2	188K (+*48%*)	2.6 GHz (8)	$1166
E5-2660v2	183K (+*44%*)	2.2 GHz (10)	$1389

Table 2: Achieved IOPS on an 80/20 random 4K workload across 2 MicroArrays

7.6 Effect of CPU on Performance

A workload running at full throttle with small requests completely saturates the CPU. This remains true despite significant development effort in performance debugging, and a great many improvements to minimize data movement and contention. In this section we report the performance improvements resulting from faster CPUs. These results are from random 4K NFS requests in an 80/20 readwrite mix at 128 queue depth over four 10Gb links to a cluster of two NADs, each equipped with 2 physical CPUs.

Table 2 shows the results of these tests. In short, it is possible to "buy" additional storage performance under full load by upgrading the CPUs into a more "balanced" configuration. The wins are significant and carry a non-trivial increase in the system cost. As a result of this

experimentation, we elected to use a higher performance CPU in the shipping version of the product.

8 Related Work

Strata applies principles from prior work in server virtualization, both in the form of hypervisor [5, 32] and libOS [14] architectures, to solve the problem of sharing and scaling access to fast non-volatile memories among a heterogeneous set of clients. Our contributions build upon the efforts of existing research in several areas.

Recently, researchers have begin to investigate a broad range of system performance problems posed by storage class memory in single servers [3], including current PCIe flash devices [30], next generation PCM [1], and byte addressability [13]. Moneta [9] proposed solutions to an extensive set of performance bottlenecks over the PCIe bus interface to storage, and others have investigated improving the performance of storage class memory through polling [33], and avoiding system call overheads altogether [10]. We draw from this body of work to optimize the performance of our dispatch library, and use this baseline to deliver a high performance scale-out network storage service. In many cases, we would benefit further from these efforts—for example, our implementation could be optimized to offload per-object access control checks, as in Moneta-D [10]. There is also a body of work on efficiently using flash as a caching layer for slower, cheaper storage in the context of large file hosting. For example, S-CAVE [23] optimizes cache utilization on flash for multiple virtual machines on a single VMware host by running as a hypervisor module. This work is largely complementary to ours; we support using flash as a caching layer and would benefit from more effective cache management strategies.

Prior research into scale-out storage systems, such as FAWN [2], and Corfu [4] has considered the impact of a range of NV memory devices on cluster storage performance. However, to date these systems have been designed towards lightweight processors paired with simple flash devices. It is not clear that this balance is the correct one, as evidenced by the tendency to evaluate these same designs on significantly more powerful hardware platforms than they are intended to operate [4]. Strata is explicitly designed for dense virtualized server clusters backed by performance-dense PCIe-based non-volatile memory. In addition, like older commodity disk-oriented systems including Petal [22, 29] and FAB [28], prior storage systems have tended to focus on building aggregation features at the lowest level of their designs, and then adding a single presentation layer on top. Strata in contrasts isolates shares each powerful PCIe-based storage class memory as its underlying primitive. This has allowed us to present a scalable runtime environment in which multiple protocols can coexist as peers without sacrificing the raw performance that today's high performance memory can provide. Many scale-out storage systems, including NV-Heaps [12], Ceph/RADOS [31], and even PNFS [18] are unable to support the legacy formats in enterprise environments. Our agnosticism to any particular protocol is similar to approach used by Ursa Minor [16], which also boasted a versatile client library protocol to share access to a cluster of magnetic disks.

Strata does not attempt to provide storage for datacenter-scale environments, unlike systems including Azure [6], FDS [26], or Bigtable [11]. Storage systems in this space differ significantly in their intended workload, as they emphasize high throughput linear operations. Strata's managed network would also need to be extended to support datacenter-sized scale out. We also differ from in-RAM approaches such a RAMCloud [27] and memcached [15], which offer a different class of durability guarantee and cost.

9 Conclusion

Storage system design faces a sea change resulting from the dramatic increase in the performance density of its component media. Distributed storage systems composed of even a small number of network-attached flash devices are now capable of matching the offered load of traditional systems that would have required multiple racks of spinning disks.

Strata is an enterprise storage architecture that responds to the performance characteristics of PCIe storage devices. Using building blocks of well-balanced flash, compute, and network resources and then pairing the design with the integration of SDN-based Ethernet switches, Strata provides an incrementally deployable, dynamically scalable storage system.

Strata's initial design is specifically targeted at enterprise deployments of VMware ESX, which is one of the dominant drivers of new storage deployments in enterprise environments today. The system achieves high performance and scalability for this specific NFS environment while allowing applications to interact directly with virtualized, network-attached flash hardware over new protocols. This is achieved by cleanly partitioning our storage implementation into an underlying, low-overhead virtualization layer and a scalable framework for implementing storage protocols. Over the next year, we intend to extend the system to provide general-purpose NFS support by layering a scalable and distributed metadata service and small object support above the base layer of coarse-grained storage primitives.

References

[1] AKEL, A., CAULFIELD, A. M., MOLLOV, T. I., GUPTA, R. K., AND SWANSON, S. Onyx: a protoype phase change memory storage array. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems* (Berkeley, CA, USA, 2011), HotStorage'11, USENIX Association, pp. 2–2.

[2] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), SOSP '09, pp. 1–14.

[3] BAILEY, K., CEZE, L., GRIBBLE, S. D., AND LEVY, H. M. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems* (Berkeley, CA, USA, 2011), HotOS'13, USENIX Association, pp. 2–2.

[4] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBER, T., WEI, M., AND DAVIS, J. D. Corfu: a shared log design for flash clusters. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), NSDI'12.

[5] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (2003), SOSP '03, pp. 164–177.

[6] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., HARIDAS, J., UDDARAJU, C., KHATRI, H., EDWARDS, A., BEDEKAR, V., MAINALI, S., ABBASI, R., AGARWAL, A., HAQ, M. F. U., HAQ, M. I. U., BHARDWAJ, D., DAYANAND, S., ADUSUMILLI, A., MCNETT, M., SANKARAN, S., MANIVANNAN, K., AND RIGAS, L. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), SOSP '11, pp. 143–157.

[7] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking control of the enterprise. In *In SIGCOMM Computer Comm. Rev* (2007).

[8] CASADO, M., GARFINKEL, T., AKELLA, A., FREEDMAN, M. J., BONEH, D., MCKEOWN, N., AND SHENKER, S. Sane: a protection architecture for enterprise networks. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15* (Berkeley, CA, USA, 2006), USENIX-SS'06, USENIX Association.

[9] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (2010), MICRO '43, pp. 385–395.

[10] CAULFIELD, A. M., MOLLOV, T. I., EISNER, L. A., DE, A., COBURN, J., AND SWANSON, S. Providing safe, user space access to fast, solid state disks. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII, pp. 387–400.

[11] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[12] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. 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* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 105–118.

[13] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 133–146.

[14] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles* (1995), SOSP '95, pp. 251–266.

[15] FITZPATRICK, B. Distributed caching with memcached. *Linux J. 2004*, 124 (Aug. 2004), 5–.

[16] GANGER, G. R., ABD-EL-MALEK, M., CRANOR, C., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M., PRASAD, M., SALMON, B., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., AND WYLIE, J. J. Ursa minor: versatile cluster-based storage, 2005.

[17] GIBSON, G. A., AMIRI, K., AND NAGLE, D. F. A case for network-attached secure disks. Tech. Rep. CMU-CS-96-142, Carnegie-Mellon University.Computer science. Pittsburgh (PA US), Pittsburgh, 1996.

[18] HILDEBRAND, D., AND HONEYMAN, P. Exporting storage systems in a scalable manner with pnfs. In *IN PROCEEDINGS OF 22ND IEEE/13TH NASA GODDARD CONFERENCE ON MASS STORAGE SYSTEMS AND TECHNOLOGIES (MSST* (2005).

[19] HUTCHINSON, N. C., AND PETERSON, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng. 17*, 1 (Jan. 1991), 64–76.

[20] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (1997), STOC '97, pp. 654–663.

[21] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The click modular router. *ACM Trans. Comput. Syst. 18*, 3 (Aug. 2000), 263–297.

[22] LEE, E. K., AND THEKKATH, C. A. Petal: distributed virtual disks. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems* (1996), ASPLOS VII, pp. 84–92.

[23] LUO, T., MA, S., LEE, R., ZHANG, X., LIU, D., AND ZHOU, L. S-cave: Effective ssd caching to improve virtual machine storage performance. In *Parallel Architectures and Compilation Techniques* (2013), PACT '13, pp. 103–112.

[24] MEYER, D. T., CULLY, B., WIRES, J., HUTCHINSON, N. C., AND WARFIELD, A. Block mason. In *Proceedings of the First conference on I/O virtualization* (2008), WIOV'08.

[25] MOSBERGER, D., AND PETERSON, L. L. Making paths explicit in the scout operating system. In *Proceedings of the second USENIX symposium on Operating systems design and implementation* (1996), OSDI '96, pp. 153–167.

[26] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O., HOWELL, J., AND SUZUE, Y. Flat datacenter storage. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 1–15.

[27] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramcloud. *Commun. ACM 54*, 7 (July 2011), 121–130.

[28] SAITO, Y., FRØLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. Fab: building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2004), ASPLOS XI, ACM, pp. 48–58.

[29] THEKKATH, C. A., MANN, T., AND LEE, E. K. Frangipani: a scalable distributed file system. In *Proceedings of the sixteenth ACM symposium on Operating systems principles* (1997), SOSP '97, pp. 224–237.

[30] VASUDEVAN, V., KAMINSKY, M., AND ANDERSEN, D. G. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 8:1–8:13.

[31] WEIL, S. A., WANG, F., XIN, Q., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable object-based storage system. Tech. rep., 2006.

[32] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Denali: A scalable isolation kernel. In *Proceedings of the Tenth ACM SIGOPS European Workshop* (2002).

[33] YANG, J., MINTURN, D. B., AND HADY, F. When poll is better than interrupt. In *Proceedings of the 10th USENIX conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 3–3.

[34] Flexible io tester. http://git.kernel.dk/?p=fio.git;a=summary.

[35] Linux device mapper resource page. http://sourceware.org/dm/.

[36] Linux logical volume manager (lvm2) resource page. http://sourceware.org/lvm2/.

[37] Seagate kinetic open storage documentation. https://developers.seagate.com/display/KV/Kinetic+Open+Storage+Documentation+Wiki.

[38] Scsi object-based storage device commands - 2, 2011. http://www.incits.org/scopes/1729.htm.

Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches

Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, Lawrence Chiu
IBM Almaden Research

Abstract

Storage systems based on Phase Change Memory (PCM) devices are beginning to generate considerable attention in both industry and academic communities. But whether the technology in its current state will be a commercially and technically viable alternative to entrenched technologies such as flash-based SSDs remains undecided. To address this it is important to consider PCM SSD devices not just from a device standpoint, but also from a holistic perspective.

This paper presents the results of our performance study of a recent all-PCM SSD prototype. The average latency for a 4 KiB random read is 6.7 µs, which is about 16× faster than a comparable eMLC flash SSD. The distribution of I/O response times is also much narrower than flash SSD for both reads and writes. Based on the performance measurements and real-world workload traces, we explore two typical storage use-cases: tiering and caching. For tiering, we model a hypothetical storage system that consists of flash, HDD, and PCM to identify the combinations of device types that offer the best performance within cost constraints. For caching, we study whether PCM can improve performance compared to flash in terms of aggregate I/O time and read latency. We report that the IOPS/$ of a tiered storage system can be improved by 12–66% and the aggregate elapsed time of a server-side caching solution can be improved by up to 35% by adding PCM.

Our results show that – even at current price points – PCM storage devices show promising performance as a new component in enterprise storage systems.

1 Introduction

In the last decade, solid-state storage technology has dramatically changed the architecture of enterprise storage systems. Flash memory based solid state drives (SSDs) outperform hard disk drives (HDDs) along a number of dimensions. When compared to HDDs, SSDs have higher storage density, lower power consumption, a smaller thermal footprint and orders of magnitude lower latency. Flash storage has been deployed at various levels in enterprise storage architecture ranging from a storage tier in a multi-tiered environment (e.g., IBM Easy Tier [15], EMC FAST [9]) to a caching layer within the storage server (e.g., IBM XIV SSD cache [17]), to an application server-side cache (e.g., IBM Easy Tier Server [16], EMC XtreamSW Cache [10], NetApp Flash Accel [24], FusionIO ioTurbine [11]). More recently, several all-flash storage systems that completely eliminate HDDs (e.g., IBM FlashSystem 820 [14], Pure Storage [25]) have also been developed. However, flash memory based SSDs come with their own set of concerns such as durability and high-latency erase operations.

Several non-volatile memory technologies are being considered as successors to flash. Magneto-resistive Random Access Memory (MRAM [2]) promises even lower latency than DRAM, but it requires improvements to solve its density issues; the current MRAM designs do not come close to flash in terms of cell size. Ferroelectric Random Access Memory (FeRAM [13]) also promises better performance characteristics than flash, but lower storage density, capacity limitations, and higher cost issues remain to be addressed. On the other hand, Phase Change Memory (PCM [29]) is a more imminent technology that has reached a level of maturity that permits deployment at commercial scale. Micron announced mass production of a 128 Mbit PCM device in 2008 while Samsung announced the mass production of 512 Mbit PCM device follow-on in 2009. In 2012, Micron also announced in volume production of a 1 Gbit PCM device.

PCM technology stores data bits by alternating the phase of material between *crystalline* and *amorphous*. The crystalline state represents a logical 1 while the amorphous state represents a logical 0. The phase is alternated by applying varying length current pulses de-

pending upon the phase to be achieved, representing the write operation. Read operations involve applying a small current and measuring the resistance of the material.

Flash and DRAM technologies represent data by storing electric charge. Hence these technologies have difficulty scaling down to thinner manufacturing processes, which may result in bit errors. On the other hand, PCM technology is based on the phase of material rather than electric charge and has therefore been regarded as more scalable and durable than flash memory [28].

In order to evaluate the feasibility and benefits of PCM technologies from a systems perspective, access to accurate *system-level* device performance characteristics is essential. Extrapolating *material-level* characteristics to a *system-level* without careful consideration may result in inaccuracies. For instance, a previously published paper states that PCM write performance is only 12× slower than DRAM based on the 150 ns *set operation time* reported in [4]. However, the reported write throughput from the referred publication [4] is only 2.5 MiB/s, and thus the statement that PCM write performance is only 12× slower is misleading. The missing link is that only *two bits* can be written during 200 µs on the PCM chip because of circuit delay and power consumption issues [4]. While we may conclude that *PCM write operations* are 12× slower than *DRAM write operations*, it is incorrect to conclude that a *PCM device* is only 12× slower than a *DRAM device* for writes. This reinforces the need to consider PCM performance characteristics from a system perspective based on independent measurement in the right setting as opposed to simply re-using device level performance characteristics.

Our first contribution is the result of our *system-level* performance study based on a real prototype all-PCM SSD from Micron. In order to conduct this study, we have developed a framework that can measure I/O latencies at nanosecond granularity for read and write operations. Measured over five million random 4 KiB read requests, the PCM SSD device achieves an average latency of 6.7 µs. Over one million random 4 KiB write requests, the average latency of a PCM SSD device is about 128.3 µs. We compared the performance of the PCM SSD with an *Enterprise Multi-Level Cell (eMLC)* flash based SSD. The results show that in comparison to eMLC SSD, read latency is about 16× shorter, but write latency is 3.5× longer on the PCM SSD device.

Our second contribution is an evaluation of the feasibility and benefits of including a PCM SSD device as a tier within a multi-tier enterprise storage system. Based on the conclusions of our performance study, reads are faster but writes are slower on PCM SSDs when compared to flash SSDs, and at present PCM SSDs are priced higher than flash SSD ($ / GB). Does a system built with a PCM SSD offer any advantage over one without PCM SSDs? We approach this issue by modeling a hypothetical storage system that consists of three device types: PCM SSDs, flash SSDs, and HDDs. We evaluate this storage system using several real-world traces to identify optimal configurations for each workload. Our results show that PCM SSDs can remarkably improve the performance of a tiered storage system. For instance, for a one week retail workload trace, 30% PCM + 67% flash + 3% HDD combination has about 81% increased IOPS/$ from the best configuration without PCM, 94% flash + 6% HDD even when we assume that PCM SSD devices are four times more expensive than flash SSDs.

Our third contribution is an evaluation of the feasibility and benefits of using a PCM SSD device as an application server-side cache *instead of* or *in combination with* flash. Today flash SSD based server-side caching solutions are appearing in the industry [10, 11, 16, 24] and also gaining attention in academia [12, 20]. What is the impact of using the 16× faster (for reads) PCM SSD instead of flash SSD as a server-side caching device? We run cache simulations with real-world workload traces from enterprise storage systems to evaluate this. According to our observations, a combination of flash and PCM SSDs can provide better aggregate I/O time and read latency than a flash only configuration.

The rest of the paper is structured as follows: Section 2 provides a brief background and discusses related work. We present our measurement study on a real all-PCM prototype SSD in Section 3. Section 4 describes our model and analysis for a hypothetical tiered storage system with PCM, flash, and HDD devices. Section 5 covers the use-case for server-side caching with PCM. We present a discussion of the observations in Section 6 and conclude in Section 7.

2 Background and related work

There are two possible approaches to using PCM devices in systems: as storage or as memory. The storage approach is a natural option considering the non-volatile characteristics of PCM, and there are several very interesting studies based on real PCM devices.

In 2008, Kim, et al. proposed a hybrid Flash Translation Layer (FTL) architecture, and conducted experiments with a real 64 MiB PCM device (KPS1215EZM) [19]. We believe that the PCM chip was based on 90 nm technology, published in early 2007 [22]. The paper reported 80 ns and 10 µs as word (16 bits) access time for read and write, respectively. Better write performance numbers are found in Samsung's 2007 90 nm PCM paper [22]: 0.58 MB/s in ×2 division-write mode, 4.64 MB/s in ×16 accelerated write mode.

Table 1: A PCM SSD prototype: *Micron built an all-PCM SSD prototype with their newest 45 nm PCM chips.*

Usable Capacity	64 GiB
System Interface	PCIe gen2 x8
Minimum Access Size	4 KiB
Seq. Read BW. (128 KiB)	2.6 GiB/s
Seq. Write BW. (128 KiB)	100-300 MiB/s

In 2011, a prototype all-PCM 10 GB SSD was built by researchers from the University of California, San Diego [1]. This SSD, named *Onyx*, was based on Micron's first-generation P8P 16 MiB PCM chips (NP8P128A13B1760E). On the chip, a read operation for 16 bytes takes 314 ns (48.6 MB/s), and a write operation for 64 bytes requires 120 µs (0.5 MB/s). Onyx drives many PCM chips concurrently, and provides 38 µs and 179 µs for 4 KiB read and write latencies, respectively. The Onyx design corroborates the potential of PCM as a storage device which allows massive parallelization to improve the limited write throughput of today's PCM chips. In 2012, another paper was published based on a different prototype PCM SSD built by Micron [3], using the same Micron 90 nm PCM chip used in Onyx. This prototype PCM SSD provides 12 GB capacity, and takes 20 µs and 250 µs for 4 KiB read and write, respectively, excluding software overhead. This device shows better read performance and worse write performance than the one presented in Oynx. The authors compare the PCM SSD with Fusion IO's Single-Level Cell (SLC) flash SSD, and point out that PCM SSD is about 2× faster for read, and 1.6× slower for write than the compared flash SSD.

Alternatively, PCM devices can be used as memory [18, 21, 23, 26, 27]. The main challenge in using PCM devices as a memory device is that writes are too slow. In PCM technology, high heat (over 600°C) is applied to a storage cell to change the phase to store data. The combination of quick heating and cooling results in the amorphous phase, and this operation is referred to as a *reset operation*. The *set operation* requires a longer cooling time to switch to the crystalline phase, and write performance is determined by the time required for a set operation. In several papers, PCM's set operation time is used as an approximation for the write performance for a simulated PCM device. However, care needs to be taken to differentiate among material, chip-level and device level performance. Set and reset operation times describe material level performance, which is often very different from chip level performance. For example, in Bedeschi et al. [4], the set operation time is 150 ns, but reported write throughput is only 2.5 MB/s because only two bits can be written concurrently, and there is an additional circuit delay of 50 ns. Similarly, the chip level performance differs from the device level (SSD) performance. In the rest of the paper, our performance measurements address *device level* performance based on a recent PCM SSD prototype device based on newer 45 nm chips from Micron.

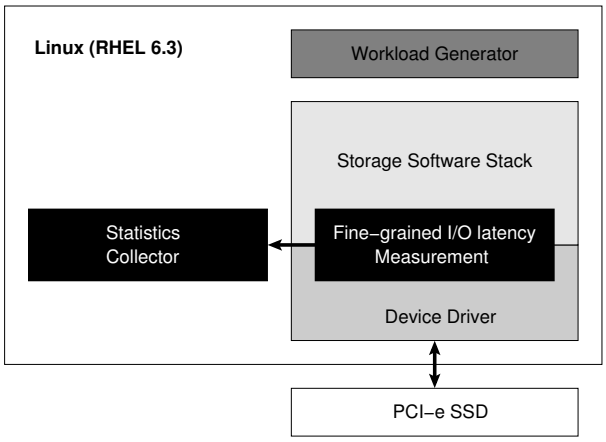


Figure 1: Measurement framework: *we modified both the Linux kernel and the device driver to collect I/O latencies in nanosecond units. We also use an in-house workload generator and a statistics collector.*

3 PCM SSD performance

In this section we describe our methodology and results for the characterization of system-level performance of a PCM SSD device. Table 1 summarizes the main features of the prototype PCM SSD device used for this study.

In order to collect fine-grained I/O latency measurements, we have patched the kernel of Red Hat Enterprise Linux 6.3. Our kernel patch enables measurement of I/O response times at nanosecond granularity. We have also modified the drivers of the SSD devices to measure the elapsed time from the arrival of an I/O request at the SSD to its completion (at the SSD). Therefore, the I/O latency measured by our method includes minimal software overhead.

Figure 1 shows our measurement framework. The system consists of a workload generator, a modified storage stack within the Linux kernel that can measure I/O latencies at nanosecond granularity, a statistics collector, and a modified device driver that measures the elapsed time for an I/O request. For each I/O request generated by the workload generator, the device driver measures the time required to service the request and passes that information back to the Linux kernel. The modified Linux kernel keeps the data in two different forms: a histogram (for long term statistics) and a fixed length log (for precise

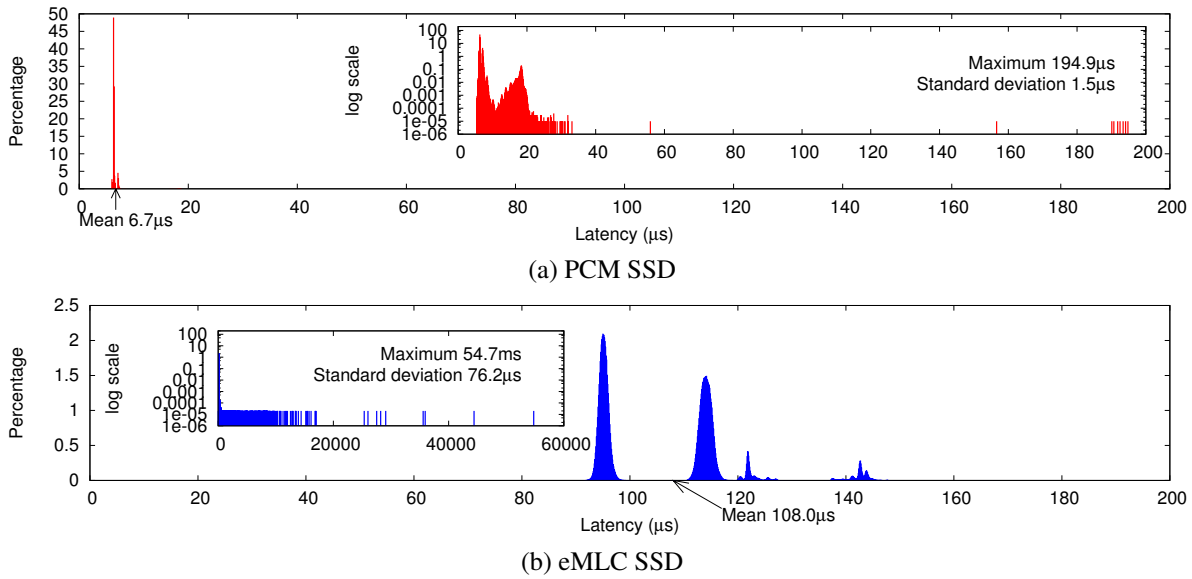


Figure 2: **4 KiB random read latencies for five million samples**: *PCM SSD shows about 16× faster average, much smaller maximum, and also much narrower distribution than eMLC SSD.*

data collection). Periodically, the collected information is passed to an external *statistics collector*, which stores the data in a file.

For the purpose of comparison, we use an eMLC flash-based PCI-e SSD providing 1.8 TiB user capacity. To capture the performance characteristics at extreme conditions, we precondition both the PCM and the eMLC flash SSDs using the following steps: 1) Perform raw formatting using tools provided by SSD vendors. 2) Fill the whole device (usable capacity) with random data, sequentially. 3) Run full random, 20% write, 80% read I/O requests with 256 concurrent streams for one hour.

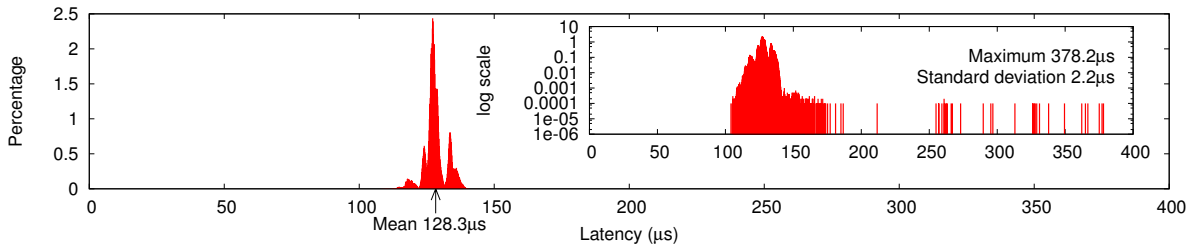
3.1 I/O Latency

Immediately after the preconditioning is complete we set the workload generator to issue one million 4 KiB sized random write requests with a single thread. We collect write latency for each request and the collected data is periodically retrieved and written to a performance log file. After one million writes complete, we set the workload generator to issue five million 4 KiB sized random read requests by using a single thread. Read latencies are collected using the same method.

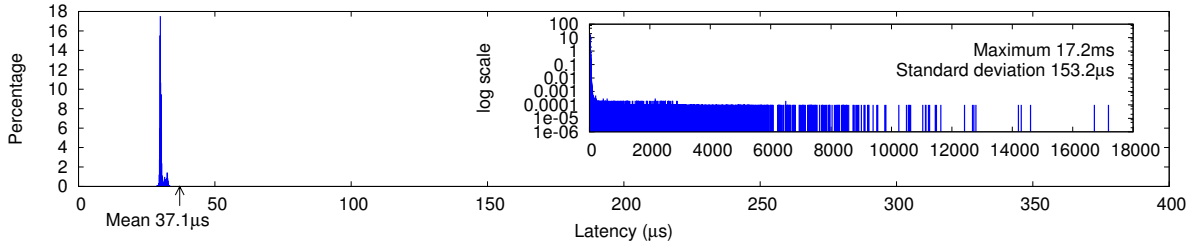
Figure 2 shows the distributions of collected read latencies for the PCM SSD (Figure 2(a)) and the eMLC SSD (Figure 2(b)). The X-axis represents the measured read latency, and the Y-axis represents the percentage of data samples. Each graph has a smaller graph embedded, which presents the whole data range with a log scaled Y-axis.

Several important results can be observed from the graphs. First, the average latency of the PCM SSD device is only 6.7 µs, which is about 16× faster than the eMLC flash SSD's average read latency of 108.0 µs. This number is much improved from the prior PCM SSD prototypes (Onyx: 38 µs [1], 90 nm Micron: 20 µs [3]). Second, the PCM SSD latency measurements show much smaller standard deviation (1.5 µs, 22% of mean) than the eMLC flash SSD's measurements (76.2 µs, 71% of average). Finally, the maximum latency is also much smaller on the PCM SSD (194.9 µs) than on the eMLC flash SSD (54.7 ms).

Figure 3 shows the latency distribution graphs for 4 KiB random writes. Interestingly, eMLC flash SSD (Figure 3(b)) shows a very short average write response time of only 37.1 µs. We believe that this is due to the RAM buffer within the eMLC flash SSD. Note that over 240 µs latency was measured for 4 KiB random writes even on Fusion IO's SLC flash SSD [3]. According to our investigation, the PCM SSD prototype does not implement RAM based write buffering, and the measured write latency is 128.3 µs (Figure 3(a)). Even though this latency number is about 3.5× longer than the eMLC SSD's average, it is still much better than the performance measurements from previous PCM prototypes. Previous measurements reported for 4 KiB write latencies are 179 µs and 250 µs in Onyx [1] and 90 nm PCM SSDs [3], respectively. As in the case of reads, for standard deviation and maximum value measurements the PCM SSD outperforms the eMLC SSD; the PCM SSD's standard deviation is only 2% of the average and the

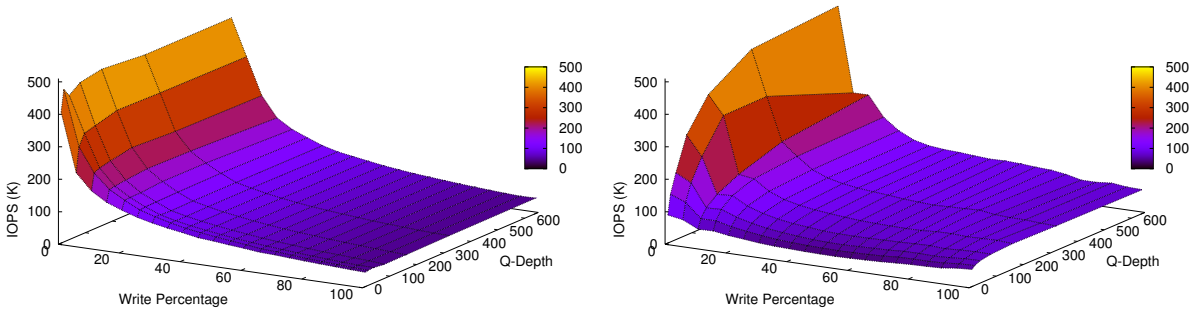


(a) PCM SSD



(b) eMLC SSD

Figure 3: **4 KiB random write latencies for one million samples**: *PCM SSD shows about 3.5× slower mean, but its maximum and distribution are smaller and narrower than eMLC SSD.*



(a) PCM SSD

(b) eMLC SSD

Figure 4: Asynchronous IOPS: *I/O request handling capability for different read and write ratios and for different degree of parallelism.*

maximum latency is 378.2 µs while the eMLC flash SSD shows 153.2 µs standard deviation (413% of the average) and 17.2 ms maximum latency value. These results lead us to conclude that the PCM SSD performance is more consistent and hence predictable than that of the eMLC flash SSD.

Micron provided this feedback on our measurements: this prototype SSD uses a PCM chip architecture that was designed for code storage applications, and thus has limited write bandwidth. Micron expects future devices targeted at this application to have lower write latency. Furthermore, the write performance measured in the drive is not the full capability of PCM technology. Additional work is ongoing to improve the write characteristics of PCM.

3.2 Asynchronous I/O

In this test, we observe the number of I/Os per second (IOPS) while varying the read and write ratio and the degree of parallelism. In Figure 4, two 3-dimensional graphs show the measured results. The X-axis represents the percentage of writes, the Y-axis represents the queue depth (i.e. number of concurrent IO requests issued), and the Z-axis represents the IOPS measured. The most obvious difference between the two graphs occurs when the queue depth is low and all requests are reads (lower left corner of the graphs). At this point, the PCM SSD shows much higher IOPS than the eMLC flash SSD. For the PCM SSD, performance does not vary much with variation in queue depth. However, on the eMLC SSD, IOPS increases with increase in queue depth. In general, the

Table 2: The parameters for tiering simulation

	PCM	eMLC	15K HDD
4 KiB R. Lat.	6.7 µs	108.0 µs	5 ms
4 KiB W. Lat.	128.3 µs	37.1 µs	5 ms
Norm. Cost	24	6	1

PCM SSD shows smoother surfaces when varying the read / write ratio. It again supports our finding that the PCM SSD is more predictable than the eMLC flash SSD.

4 Workload simulation for storage tiering

The results of our measurements on PCM SSD device performance show that the PCM SSD improves read performance by 16×, but shows about 3.5× slower write performance than eMLC flash SSD. Will such a storage device be useful for building enterprise storage systems? Current flash SSD and HDD tiered storage systems maximize *performance per dollar* (price-performance ratio) by placing hot data on faster flash SSD storage and cold data on cheaper HDD devices. Based on PCM SSD device performance, an obvious approach is to place hot, read intensive data on PCM devices; hot, write intensive data on flash SSD devices; and cold data on HDD to maximize performance per dollar. But do real-world workloads demonstrate such workload distribution characteristics? In order to address this question, we first model a hypothetical tiered storage system consisting of PCM SSD, flash SSD and HDD devices. Next we apply to our model several real-world workload traces collected from enterprise tiered storage systems consisting of flash SSD and HDD devices. Our goal is to understand whether there is any advantage to using PCM SSD devices based on the characteristics exhibited by real workload traces.

Table 2 shows the parameters used for our modeling. For PCM and flash SSDs, we use the data collected from our measurements. For the HDD device we use 5 ms for both 4 KiB random read and write latencies [7]. We compare the various alternative configurations using performance per dollar as a metric. In order to use this metric, we need price estimates for the storage devices. We assume that a PCM device is 4× more expensive than eMLC flash, and eMLC flash is 6× more expensive than 15 K RPM HDD. The flash-HDD price assumption is based on today's (June 2013) market prices from Dell's web page [6, 8]. We prefer the Dell's prices to Newegg's or Amazon's because we want to use prices for enterprise class devices. The PCM-flash price assumption is based on an opinion from an expert who prefers to remain anonymous; it is our best effort considering that the 45 nm PCM device is not available in the market yet.

We present two methodologies for evaluating PCM capabilities for a tiering approach: static optimal tiering and dynamic tiering. Static optimal tiering assumes static and optimal data placement based on complete knowledge about a given workload. While this methodology provides a simple back-of-the-envelope calculation to evaluate the effectiveness of PCM, we acknowledge that this assumption may be unrealistic and that data placements need to adapt dynamically to runtime changes in workload characteristics.

Accordingly, our second evaluation methodology is a simulation-based technique to evaluate PCM deployments in a dynamic tiered setting. Dynamic tiering assumes that data migrations are reactive and dynamic in nature and in response to changes in workload characteristics and system conditions. The simulated system begins with no prior knowledge about the workload. The simulation algorithm then periodically gathers I/O statistics, learns workload behavior and migrates data to appropriate locations in response to workload characteristics.

4.1 Evaluation metric

For a given workload observation window and a hypothetical storage composed of *X*% of PCM, *Y*% of flash, and *Z*% of HDD, we calculate the IOPS/$ metric using the following steps:

Step 1. From a given workload during the observation window, aggregate the total amount of read and write I/O traffic at an extent (1 GiB) granularity. An extent is the unit of data migration in tiered storage environment. In our analysis, the extent size is set to 1 GiB accordingly to the configuration of the real-world tiered storage systems from which our workload traces were collected.

Step 2. Let $ReadLat._{HDD}$, $ReadLat._{Flash}$ and $ReadLat._{PCM}$ represent the read latencies of HDD, flash and PCM devices respectively. Similarly, let $WriteLat._{HDD}$, $WriteLat._{Flash}$ and $WriteLat._{PCM}$ represent the write latencies. Let $ReadAmount_{Extent}$ and $WriteAmount_{Extent}$ represent the amount of read and write traffic given to the extent under consideration. For each extent, calculate $Score_{Extent}$ using the following equations:

$$Score_{PCM} = (ReadLat._{HDD} - ReadLat._{PCM}) \times ReadAmount_{Extent} + (WriteLat._{HDD} - WriteLat._{PCM}) \times WriteAmount_{Extent}$$

$$Score_{Flash} = (ReadLat._{HDD} - ReadLat._{Flash}) \times ReadAmount_{Extent} + (WriteLat._{HDD} - WriteLat._{Flash}) \times WriteAmount_{Extent}$$

$$Score_{Extent} = MAX(Score_{PCM}, Score_{Flash})$$

Step 3. Sort extents by $Score_{Extent}$ in descending order.

Step 4. Assign a tier for each extent based on Algorithm 1. This algorithm can fail if either (1) HDD is the best choice, or (2) we run out of HDD space, but that will never happen with our configuration parameters.

Algorithm 1 Data placement algorithm

```
for e in SortedExtentsByScore do
    tgtTier ← (e.scorePCM > e.scoreFlash)?PCM : FLASH
    if (tgtTier.freeExt > 0) then
        e.tier ← tgtTier
        tgtTier.freeExt ← tgtTier.freeExt − 1
    else
        tgtTier ← (tgtTier == PCM)?FLASH : PCM
        if (tgtTier.freeExt > 0) then
            e.tier ← tgtTier
            tgtTier.freeExt ← tgtTier.freeExt − 1
        else
            e.tier ← HDD
        end if
    end if
end for
```

Step 5. Aggregate the amount of read and write I/O traffic for PCM, flash, and HDD tiers based on the data placement.

Step 6. Calculate *expected average latency* based on the amount of read and write traffic received by each storage media type and the parameters in Table 2.

Step 7. Calculate *expected average IOPS* as 1 / *expected average latency*.

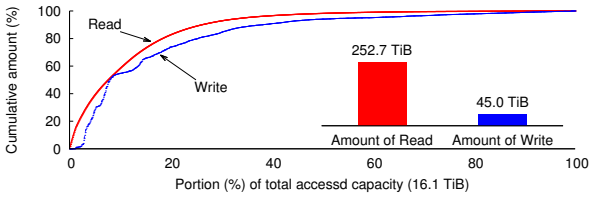
Step 8. Calculate *normalized cost* based on the percentage of storage: for example, the *normalized cost* for an all-HDD configuration is 1, and the *normalized cost* for a 50% PCM + 50% flash configuration is $(24 \times 0.5) + (6 \times 0.5) = 15$.

Step 9. Calculate *performance-price ratio = IOPS/$* as *expected average IOPS* (from Step 7) / *normalized cost* (from Step 8).

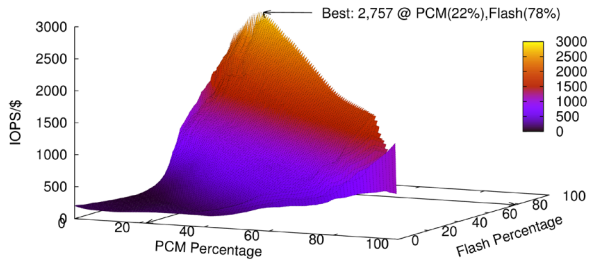
The value obtained from Step 9 represents the IOPS per normalized cost – a higher value implies better performance per dollar. We repeat this calculation for every possible combination of PCM, flash, and HDD to find the most desirable combination for a given workload.

4.2 Simulation methodology

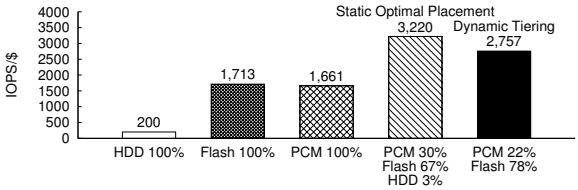
In the case of the static optimal placement methodology, the entire workload duration is treated as a single observation window and we assume unlimited migration bandwidth. The dynamic tiering methodology uses a two-hour workload observation window before making migration decisions and assumes a migration bandwidth of 41 MiB/s according to the configurations of real-world tiered storage systems from which we collected workload traces. Our experimental evaluation shows that utilizing PCM can result in a significant performance improvement. We compare the results from the static optimal methodology and the dynamic tiering methodology using the evaluation metric described in Section 4.1.



(a) CDF and I/O amount



(b) 3D IOPS/$ by dynamic tiering



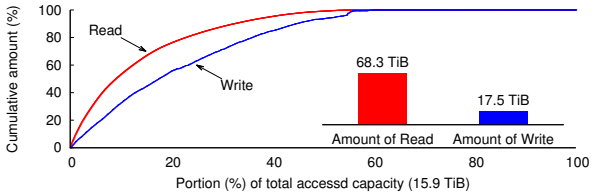
(c) IOPS/$ for key configuration points

Figure 5: Simulation result for the retail store trace: *this workload is very friendly for PCM; read dominant and highly skewed spatially – PCM (22%) + flash (78%) configuration can make the best IOPS/$ value (2,757) in dynamic tiering simulation.*

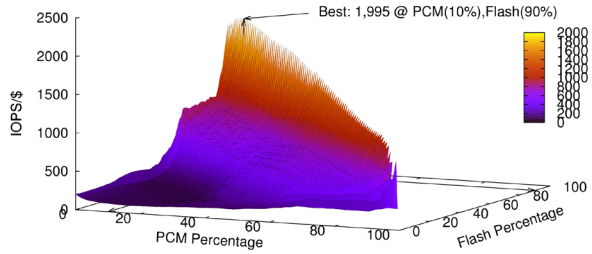
4.3 Result 1: Retail store

The first trace is a one week trace collected from an enterprise storage system used for online transactions at a retail store. Figure 5(a) shows the cumulative distribution as well as the total amount of read and write I/O traffic: the total storage capacity accessed during this duration is 16.1 TiB, the total amount of read traffic is 252.7 TiB, and the total amount of write traffic is 45.0 TiB. As can be seen from the distribution, the workload is heavily skewed, with 20% of the storage capacity receiving 83% of the read traffic and 74% of the write traffic. The distribution also exhibits a heavy skew toward reads, with nearly six times more reads than writes.

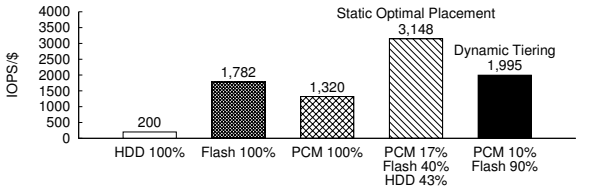
Figures 5 (b) and (c) show the modeling results. Graph (b) represents performance price ratios obtained by dynamic tiering simulation on a 3-dimensional surface, and graph (c) shows the same performance–price values (IOPS/$) for several important data points: all-HDD, all-flash, all-PCM, the best configuration for static optimal data placement, and the best configuration for



(a) CDF and I/O amount

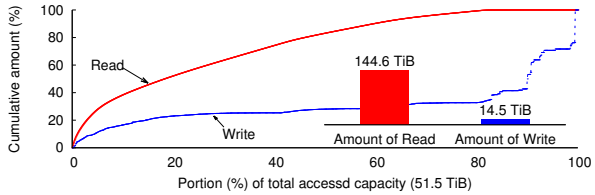


(b) 3D IOPS/$ by dynamic tiering

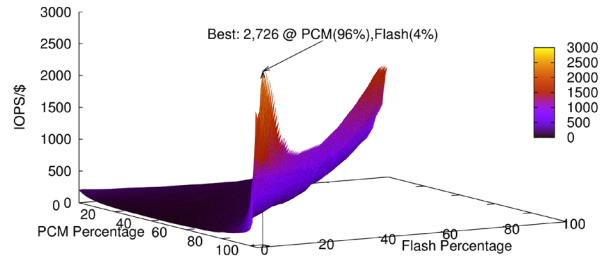


(c) IOPS/$ for key configuration points

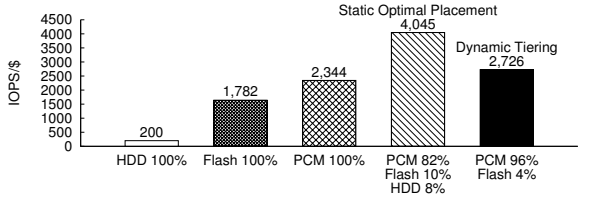
Figure 6: Simulation result for the bank trace: *this workload is less friendly for PCM than the retail workload – PCM (10%) + flash (90%) configuration can make the best IOPS/$ value (1,995) in dynamic tiering simulation.*



(a) CDF and I/O amount



(b) 3D IOPS/$ by dynamic tiering



(c) IOPS/$ for key configuration points

Figure 7: Simulation result for the telecommunication company trace: *this workload is less spatially skewed, but the amount of read is about 10× of the amount of write – PCM (96%) + flash (4%) configuration can make the best IOPS/$ value (2,726) in dynamic tiering simulation.*

dynamic tiering. Note that for the first three homogeneous storage configurations, there is no difference between static and dynamic simulation results. The best combination using static data placement consists of PCM (30%) + flash (67%) + HDD (3%), and the calculated IOPS/$ value is 3,220, which is about 81% higher than the best combination without PCM: 94% flash + 6% HDD yielding 1,777 IOPS/$; the best combination from dynamic tiering simulation consists of PCM (22%) + flash (78%), and the obtained IOPS/$ value is 2,757. This value is about 61% higher than the best combination without PCM: 100% flash yielding 1,713 IOPS/$.

4.4 Result 2: Bank

The second trace is a one week trace from a bank. The total storage capacity accessed is 15.9 TiB, the total amount of read traffic is 68.3 TiB, and the total amount of write traffic is 17.5 TiB as shown in Figure 6(a). Read to write ratio is 3.9 : 1, and the degree of skew toward reads is less than the previous retail store trace (Figure 5(a)). Approximately 20% of the storage capacity receives about 76% of the read traffic and 56% of the write traffic.

Figures 6(b) and (c) show the modeling results. The best combination using static data placement consists of PCM (17%) + flash (40%) + HDD (43%), and the calculated IOPS/$ value is 3,148, which is about 14% higher than the best combination without PCM: 57% flash + 43% HDD yielding 2,772; the best combination from dynamic tiering simulation consists of PCM (10%) + flash (90%), and the obtained IOPS/$ value is 1,995. This value is about 12% higher than the best combination without PCM: 100% flash yielding 1,782 IOPS/$.

4.5 Result 3: Telecommunication company

The last trace is a one week trace from a telecommunication provider. The total accessed storage capacity is 51.5 TiB, the total amount of read traffic is 144.6 TiB, and the total amount of write traffic is about 14.5 TiB. As shown in Figure 7(a), this workload is less spatially

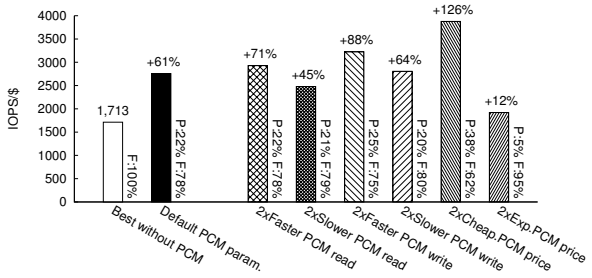


Figure 8: The best IOPS/$ for Retail store workload with varied PCM parameters

skewed than the retail and bank workloads; approximately 20% of the storage capacity receives about 52% of the read traffic and 23% of the write traffic. But read to write ratio is about 10 : 1, which is the most read dominant among the three workloads.

According to Figures 7(b) and (c), the best combination from static data placement consists of PCM (82%) + flash (10%) + HDD (8%), and calculated IOPS/$ value is 4,045, which is about 2.2× better than the best combination without PCM: 84% flash + 16% HDD yielding 1,853; the best combination from dynamic tiering simulation consists of PCM (96%) + flash (4%), and the obtained IOPS/$ value is 2,726. This value is about 66% higher than the best combination without PCM: 100% flash yielding 1,641 IOPS/$.

4.6 Sensitivity analysis for tiering

The simulation parameters are based on our best effort estimation of market price and the current state of PCM technologies, or based on discussions with experts. However, PCM technology and its markets are still evolving, and there are uncertainties about its characteristics and pricing. To understand the sensitivity of our simulation results to PCM parameters, we tried six variations of PCM parameters in three aspects: read performance, write performance, and price. For each aspect, we tried half-size and double-size values. For instance, we tested 4.35 µs and 13.4 µs instead of the original 6.7 µs for PCM 4 KiB read latency.

Figure 8 shows the highest IOPS/$ value for varying PCM parameters. We observe that our IOPS/$ measure is most sensitive to PCM price. If PCM is only twice as expensive as flash while maintaining its read and write performance, the PCM (38%) + flash (62%) configuration can yield about 126% higher IOPS/$ (3,878); if PCM is 8× more expensive than flash, PCM (5%) + flash (95%) configuration yields 1,921, which is 12% higher than the IOPS/$ value from the best configuration without PCM.

Interestingly, the configuration with twice slower PCM write latency yields an IOPS/$ of 2,806, which is slightly higher than the baseline value (2,757). That may happen because the dynamic tiering algorithm is not perfect. With the static optimal placement method, 2× longer PCM write latency results in 3,216, which is lower than the original value of 3,220.

4.7 Summary of tiering simulation

Based on the results above, we observe that PCM can increase IOPS/$ value by 12% (bank) to 66% (telecommunication company) even assuming that PCM is 4× more expensive than flash. These results suggest that PCM has high potential as a new component for enterprise storage systems in a multi-tiered environment.

5 Workload simulation for server caching

Server-side caching is gaining popularity in enterprise storage systems today [5, 10, 11, 12, 16, 20, 24]. By placing frequently accessed data close to the application on a locally attached (flash) cache, network latencies are eliminated and speedup is achieved. The remote storage node benefits from decreased contention and the overall system throughput increases.

At first glance PCM SSD seems to be promising for server-side caching, considering the 16× faster read time compared to eMLC flash SSD. But given that PCM is more expensive and slower for write than flash, will PCM be a cost effective alternative? To address this question we use a second set of real-world traces to simulate caching performance. The prior set of traces used for tiered storage simulation could not be used to evaluate cache performance since the traces were summarized spatially and temporally at a coarse granularity. Three new IO-by-IO traces are used: 1) a 24 hour trace from a manufacturing company, 2) a 36 hours trace from a media company, and 3) a 24 hour trace from a medical service company. We chose three cache friendly workloads – highly skewed and read intensive – since our goal was to compare PCM and flash for server-side caching scenarios.

5.1 Cache simulation

We built a cache simulator using an LRU cache replacement scheme, 4 KiB page size, and write-through policy, which are the typical choices for enterprise server-side caching solutions. The simulator supports both single tier and hybrid (i.e. multi-tier) cache devices to test a configuration using PCM as a first level cache and flash as a second level cache. Our measurements (Table 2) are used for PCM and flash SSDs, and for networked storage

Table 3: Networked storage related parameters from [12]

Network base latency	8.2 µs / packet
Network data latency	1 ns / bit
File server fast read	92 µs / 4 KiB
File server slow read	7,952 µs / 4 KiB
File server write	92 µs / 4 KiB
File server fast read rate	90%

Table 4: Cache simulation parameters

	PCM	eMLC	Net. Storage
4 KiB R. Lat.	6.7 µs	108.0 µs	919.0 µs
4 KiB W. Lat.	128.3 µs	37.1 µs	133.0 µs
Norm. Cost	4	1	–

we use 919 µs and 133 µs for 4 KiB read and write, respectively. These numbers are based on the timing model parameters (Table 3) from previous work [12]; network overhead for 4 KiB is calculated as 41.0 µs (8.2 µs base latency + (4,096 × 8) bits × 1 ns), write time is 133 µs (write time 92 µs + network overhead 41 µs), and read time is 919 µs (90% × fast read time 92 µs + 10% × slow read time 7,952 µs + network overhead 41 µs).

The simulator captures the total number of read and write I/Os to the caching device and the networked storage separately, and then calculates average read latency as our evaluation metric; with write-through policy, write latency cannot be improved.

We vary the cache size from 64 GiB to a size that is large enough to hold the entire dataset. We then calculate the average read latency for all-flash and all-PCM configurations.

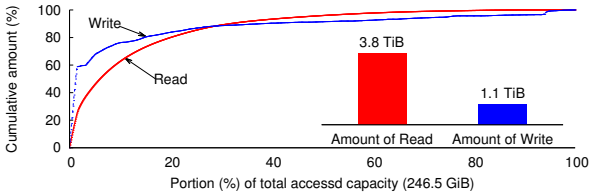
Next, we compare the cache performance for all-PCM, all-flash, and PCM and flash hybrid combinations having the same cost.

5.2 Result 1: Manufacturing company

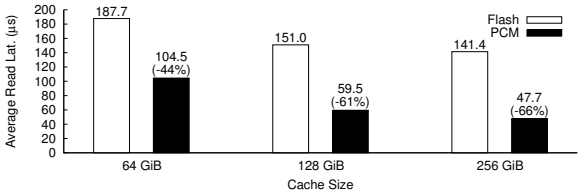
The first trace is from the storage server of a manufacturing company, running an On-Line Transaction Processing (OLTP) database on a ZFS file system.

Figure 9(a) shows the cumulative distribution as well as the total amount of read and write I/O traffic for this workload. The total accessed capacity (during 24 hours) is 246.5 GiB, the total amount of read traffic is 3.8 TiB, and the total amount of write traffic is 1.1 TiB. The workload exhibits strong skew: 20% of the storage capacity receives 80% of the read traffic and 84% of the write traffic.

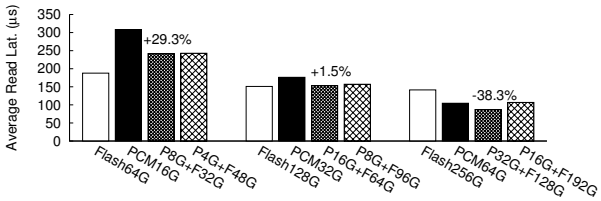
Figure 9(b) shows the average read latency (Y-axis) for flash and PCM with different cache sizes. From the



(a) CDF and I/O amount



(b) Average read latency



(c) Average read latency for even cost configurations

Figure 9: Cache simulation result for manufacturing company trace

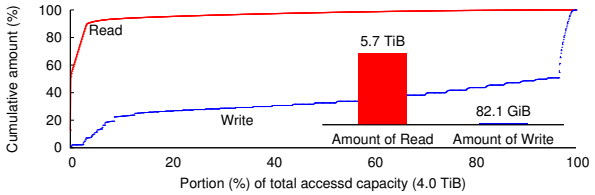
results, we see that PCM can provide an improvement of 44–66% over flash. Note that this figure assumes equal amount of PCM and flash and hence the PCM caching solution results in 4 times higher cost than an all-flash setup (Table 4).

Next, Figures 9(c) shows average read latency for cost-aware configurations. The results are divided into three groups. Within each group, we vary the ratio of PCM and flash while keeping the cost constant. For the first two groups, all-flash configurations (64 GiB, 128 GiB flash) show superior results to any configuration with PCM. For the third group (256 GiB flash), the 32 $GiB PCM + 128\ GiB flash$ combination shows about 38% shorter average read latency than an all-flash configuration.

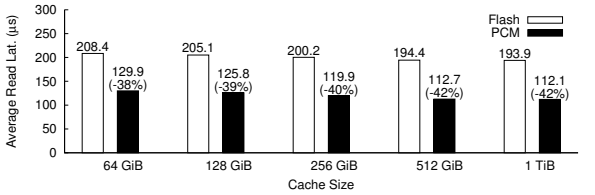
5.3 Result 2: Media company

The second trace is from the storage server of a media company, also running an OLTP database.

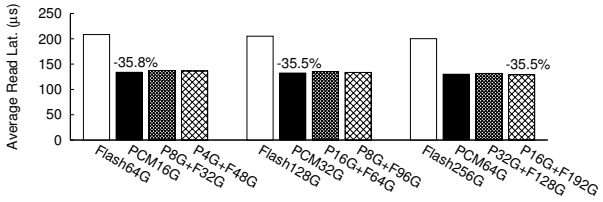
The cumulative distribution and the total amount of read and write I/O traffic are shown in Figure 10(a). The total accessed storage capacity is 4.0 TiB, the total amount of read traffic is 5.7 TiB, and the total amount of write traffic is 82.1 GiB. This workload is highly skewed and read intensive. Compared to other workloads, this workload has a larger working set size and a longer tail,



(a) CDF and I/O amount

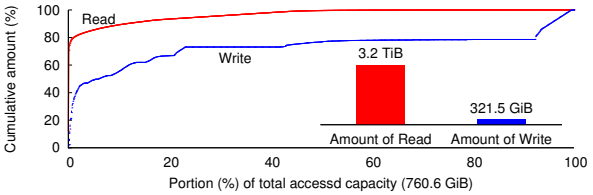


(b) Average read latency

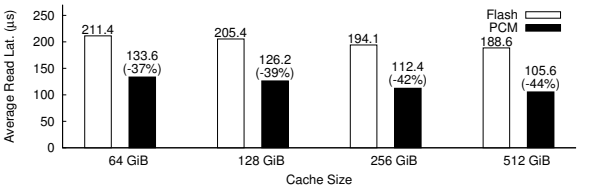


(c) Average read latency for even cost configurations

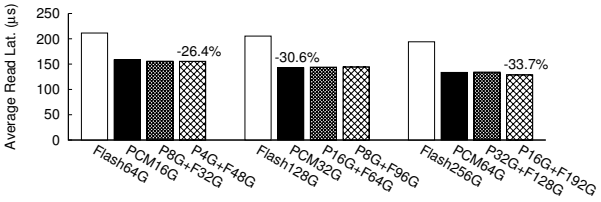
Figure 10: Cache simulation result for media company trace



(a) CDF and I/O amount



(b) Average read latency



(c) Average read latency for even cost configurations

Figure 11: Cache simulation result for medical database trace

which results in a higher proportion of cold misses.

Figure 10(b) shows average read latency (Y-axis) for different cache configurations ranging from 64 GiB to 1 TiB. Because of the large number of cold misses, the improvements are less then those observed for the first workload: 38–42% shorter read latency than flash.

Figures 10(c) shows the simulation results for cost-aware configurations. Again, the results are divided into three groups. Within each group, we vary the ratio of PCM and flash while keeping the cost constant. Unlike the previous workload (manufacturing company), PCM reduces read latency in all three groups by about 35% compared to flash.

5.4 Result 3: Medical database

The last trace was captured from a front-line patient management system. Traces were captured over a period of 24 hours, and in total 760.6 GiB of storage space was touched. The amount of read traffic (3.2 TiB) is about 10× more than the amount of write traffic (321.5 GiB), and read requests are highly skewed as shown in Figure 11(a).

Figure 11(b) shows the aggregate I/O time (Y-axis) with 64 GiB to 512 GiB cache sizes. We observe that PCM can provide 37–44% shorter read latency than flash.

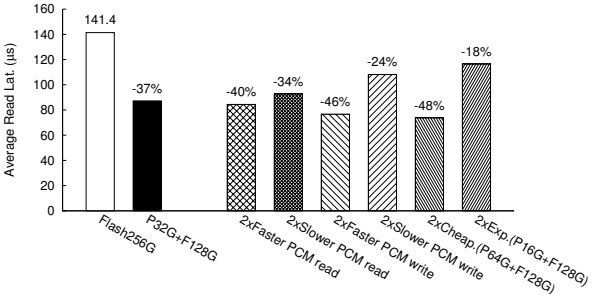


Figure 12: The average read latency for manufacturing company trace with varied PCM parameters

For the cost-aware configurations, PCM can improve read latency by 26.4–33.7% (Figure 11(d)) compared to configurations without PCM.

5.5 Sensitivity analysis for caching

Similar to the study of tiering in Section 4.6, we run sensitivity analysis for server caching as well. We test six variations of PCM parameters: (1) 2× shorter PCM read latency (4.35 µs), (2) 2× longer PCM read latency (13.4 µs), (3) 2× shorter PCM write latency (64.15 µs), (4) 2× longer PCM write latency (256.6 µs), (5) 2× cheaper normalized PCM cost (12), and finally (6) 2× more expensive normalized PCM cost (48). We pick the manufacturing company trace and its best configuration

(PCM 32 GiB + flash 128 GiB).

Figure 12 shows the simulated average read latencies for varied configurations. The same trend is shown as observed from the result for tiering (Figure 8); price creates the biggest impacts; even when performing half as well as our measured device, PCM still achieves 18–34% shorter average read latencies than all flash configuration.

5.6 Summary of caching simulation

Our cache simulation study with real-world storage access traces has demonstrated that PCM can improve aggregate I/O time by up to 66% (manufacturing company trace) compared to a configuration that uses the same size of flash. With cost-aware configurations, we show that PCM can improve average read latency up to 38% (again, manufacturing company trace) compared to the flash only configuration.

From our results, we observe that the result from the first workload (manufacturing) is different from the results of the second (media) and third (medical). While configurations with PCM offer significant performance improvement over any combination without PCM in the second and third workloads, we observe that that is true only for larger cache sizes in the first workload (i.e. Figures 9(c). This can be attributed to the varying degrees of skewing in the workloads. The first workload exhibits less skew (for read I/Os) than the second and third workloads and hence has a larger working-set size. As a result, by increasing the cache size to capture the entire working set for the first workload (data point PCM 32 GiB + flash 128 GiB), we are eventually able to achieve a configuration that captures the active working-set.

These results point to the fact that PCM-based caching options are a viable, cost-effective option to flash-based server-side caches, given a fitting workload profile. Consequently, analysis of workload characteristics is required to identify critical parameters such as proportion of writes, skew and working set size.

6 Limitations and discussion

Our study into the applicability of PCM devices in realistic enterprise storage settings has provided several insights. But we acknowledge that our analysis does have several limitations: First, since our evaluation is based on a simulation, it may not accurately represent system conditions. Second, from our asynchronous I/O test (see section 3.2), we observe that the prototype PCM device does not exploit I/O parallelism much, unlike the eMLC flash SSD. This means that it may not be fair to say that the PCM SSD is 16× faster than the eMLC SSD for read, because the eMLC SSD can handle multiple read I/O requests concurrently. It is a fair concern if we ignore the capacity of the SSDs. The eMLC flash SSD has 1.8 TiB capacity while the PCM SSD has only 64 GiB capacity. We assume that as the capacity of PCM SSD increases, its parallel I/O handling capability will increase as well. Finally, in order to understand long-term architectural implications, longer evaluation runs may be required for performance characterization.

In this study, we approach PCM as storage rather than memory, and our evaluation is focused on average performance improvements. However, we believe that the PCM technology may be capable of much more. As shown in our I/O latency measurement study, PCM can provide well-bounded I/O response times. These performance characteristics will prove to be very useful to provide Quality of Service (QoS) and multi-tenancy features. We leave exploration of these directions to future work.

7 Conclusion

Emerging workloads seem to have an ever-increasing appetite for storage performance. Today, enterprise storage systems are actively adopting flash technology. However, we must continue to explore the possibilities of next generation non-volatile memory technologies to address increasing application demands as well as to enable new applications. As PCM technology matures and production at scale begins, it is important to understand its capabilities, limitations and applicability.

In this study, we explore the opportunities for PCM technology within enterprise storage systems. We compare the latest PCM SSD prototype to an eMLC flash SSD to understand the performance characteristics of the PCM SSD as another storage tier, given the right workload mixture. We conduct a modeling study to analyze the feasibility of PCM devices in a tiered storage environment.

8 Acknowledgments

We first thank our shepherd Steven Hand and anonymous reviewers. We appreciate Micron for providing their PCM prototype hardware for our evaluation study and answering our questions. We also thank Hillery Hunter, Michael Tsao, and Luis Lastras for helping our experiments, and Paul Muench, Ohad Rodeh, Aayush Gupta, Maohua Lu, Richard Freitas, Yang Liu for their valuable comments and help.

References

[1] AKEL, A., CAULFIELD, A. M., MOLLOV, T. I., GUPTA, R. K., AND SWANSON, S. Onyx: a protoype phase change memory storage array. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems* (Berkeley, CA, USA, 2011), HotStorage'11, USENIX Association, pp. 2–2.

[2] AKERMAN, J. Toward a universal memory. *Science 308*, 5721 (2005), 508–510.

[3] ATHANASSOULIS, M., BHATTACHARJEE, B., CANIM, M., AND ROSS, K. A. Path Processing using Solid State Storage. In *Proceedings of the 3rd International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2012)* (2012).

[4] BEDESCHI, F., RESTA, C., ET AL. An 8mb demonstrator for high-density 1.8v phase-change memories. In *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on* (2004), pp. 442–445.

[5] BYAN, S., LENTINI, J., MADAN, A., PABON, L., CONDICT, M., KIMMEL, J., KLEIMAN, S., SMALL, C., AND STORER, M. Mercury: Host-side flash caching for the data center. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on* (2012), pp. 1–12.

[6] DELL. 300 gb 15,000 rpm serial attached scsi hotplug hard drive for select dell poweredge servers / powervault storage.

[7] DELL. Dell Enterprise Hard Drive and Solid-State Drive Specifications. `http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/enterprise-hdd-sdd-specification.pdf`.

[8] DELL. LSI Logic Nytro WrapDrive BLP4-1600 - Solid State Drive -1.6 TB - Internal. `http://accessories.us.dell.com/sna/productdetail.aspx?sku=A6423584`.

[9] EMC. FAST: Fully Automated Storage Tiering. `http://www.emc.com/storage/symmetrix-vmax/fast.htm`.

[10] EMC. XtreamSW Cache: Intelligent caching software that leverages server-based flash technology and write-through caching for accelerated application performance with data protection. `http://www.emc.com/storage/xtrem/xtremsw-cache.htm`.

[11] FUSION-IO. ioTurbine: Turbo Boost Virtualization. `http://www.fusionio.com/products/ioturbine`.

[12] HOLLAND, D. A., ANGELINO, E., WALD, G., AND SELTZER, M. I. Flash caching on the storage client. In *Proceedings of the 11th USENIX conference on USENIX annual technical conference* (2013), USENIXATC'13, USENIX Association.

[13] HOYA, K., TAKASHIMA, D., ET AL. A 64mb chain feram with quad-bl architecture and 200mb/s burst mode. In *Solid-State Circuits Conference, 2006. ISSCC 2006. Digest of Technical Papers. IEEE International* (2006), pp. 459–466.

[14] IBM. IBM FlashSystem 820 and IBM FlashSystem 720. `http://www.ibm.com/systems/storage/flash/720-820`.

[15] IBM. IBM System Storage DS8000 Easy Tier. `http://www.redbooks.ibm.com/abstracts/redp4667.html`.

[16] IBM. IBM System Storage DS8000 Easy Tier Server. `http://www.redbooks.ibm.com/Redbooks.nsf/RedbookAbstracts/redp5013.html`.

[17] IBM. IBM XIV Storage System. `http://www.ibm.com/systems/storage/disk/xiv`.

[18] KIM, D., LEE, S., CHUNG, J., KIM, D. H., WOO, D. H., YOO, S., AND LEE, S. Hybrid dram/pram-based main memory for single-chip cpu/gpu. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE* (2012), pp. 888–896.

[19] KIM, J. K., LEE, H. G., CHOI, S., AND BAHNG, K. I. A pram and nand flash hybrid architecture for high-performance embedded storage subsystems. In *Proceedings of the 8th ACM international conference on Embedded software* (New York, NY, USA, 2008), EMSOFT '08, ACM, pp. 31–40.

[20] KOLLER, R., MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write policies for host-side flash caches. In *Proceedings of the 11th USENIX conference on File and Storage Technologies* (2013), FAST'13, USENIX Association.

[21] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual international symposium on Computer architecture* (New York, NY, USA, 2009), ISCA '09, ACM, pp. 2–13.

[22] LEE, K.-J., ET AL. A 90nm 1.8v 512mb diode-switch pram with 266mb/s read throughput. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International* (2007), pp. 472–616.

[23] MOGUL, J. C., ARGOLLO, E., SHAH, M., AND FARABOSCHI, P. Operating system support for nvm+dram hybrid main memory. In *Proceedings of the 12th conference on Hot topics in operating systems* (Berkeley, CA, USA, 2009), HotOS'09, USENIX Association, pp. 14–14.

[24] NETAPP. Flash Accel software improves application performance by extending NetApp Virtual Storage Tier to enterprise servers. `http://www.netapp.com/us/products/storage-systems/flash-accel`.

[25] PURESTORAGE. FlashArray, Meet the new 3rd-generation FlashArray. `http://www.purestorage.com/flash-array/`.

[26] QURESHI, M. K., FRANCESCHINI, M. M., JAGMOHAN, A., AND LASTRAS, L. A. Preset: improving performance of phase change memories by exploiting asymmetry in write times. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2012), ISCA '12, IEEE Computer Society, pp. 380–391.

[27] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture* (New York, NY, USA, 2009), ISCA '09, ACM, pp. 24–33.

[28] RAOUX, S., BURR, G., BREITWISCH, M., RETTNER, C., CHEN, Y., SHELBY, R., SALINGA, M., KREBS, D., CHEN, S.-H., LUNG, H. L., AND LAM, C. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development 52*, 4.5 (2008), 465–479.

[29] SIE, C. *Memory Cell Using Bistable Resistivity in Amorphous As-Te-Ge- Film*. Iowa State University, 1969.

Wear Unleveling:
Improving NAND Flash Lifetime by Balancing Page Endurance

Xavier Jimenez, David Novo and Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland

Abstract

Flash memory cells typically undergo a few thousand *Program/Erase* (P/E) cycles before they wear out. However, the programming strategy of flash devices and process variations cause some flash cells to wear out significantly faster than others. This paper studies this variability on two commercial devices, acknowledges its unavoidability, figures out how to identify the weakest cells, and introduces a wear unbalancing technique that let the strongest cells relieve the weak ones in order to lengthen the overall lifetime of the device. Our technique periodically skips or relieves the weakest pages whenever a flash block is programmed. Relieving the weakest pages can lead to a lifetime extension of up to 60% for a negligible memory and storage overhead, while minimally affecting (sometimes improving) the write performance. Future technology nodes will bring larger variance to page endurance, increasing the need for techniques similar to the one proposed in this work.

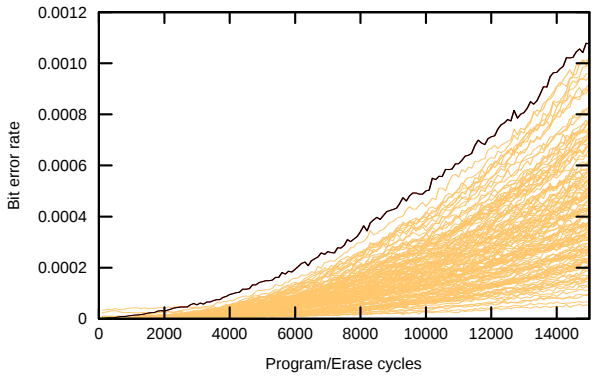


Figure 1: **Page degradation speed variation.** These data were generated by continuously writing random values into the 128 pages of a single block of flash. The BER grows at widely different speeds among pages of the same block. We suggest to reduce the stress on the weakest pages in order to enhance the block endurance.

1 Introduction

NAND flash is extensively used for general storage and transfer of data in memory cards, USB flash drives, solid-state drives, and mobile devices, such as MP3 players, smartphones, tablets or netbooks. It features low power consumption, high responsiveness and high storage density. However, flash technology also has several disadvantages. For instance, devices are physically organized in a very specific manner, in blocks of pages of bits, which results in a coarse granularity of data accesses. The memory blocks must be erased before they are able to program (i.e., write) their pages again, which results in cumbersome out-of-place updates. More importantly, flash memory cells can only experience a limited number of *Program/Erase* (P/E) cycles before they wear out. The severity of these limitations is somehow mitigated by a software abstraction layer, called a *Flash Translation Layer* (FTL), which interfaces between common file systems and the flash device.

This paper proposes a technique to extend flash devices' lifetime that can be adopted by any FTL mapping the data at the page level. It is also suitable for hybrid mappings [13, 6, 12, 5], which combine page level mapping with other coarser granularities.

The starting point of our idea is the observation that the various pages that constitute a block deteriorate at significantly different speeds (see Figure 1). Consequently, we detect the weakest pages (i.e., the pages degrading faster) to relieve them and improve the yield of the block. In essence, to relieve a page means not programming it during a P/E cycle. The idea has a similar goal as wear leveling, which balances the wear of every block. However, rather than balancing the wear, our technique carefully unbalances it in order to transfer the stress from weaker pages to stronger ones. This means

that every block of the device will be able to provide its full capacity for a longer time.

The result is a device lifetime extension of up to 60% for the experimented flash chips, at the expense of negligible storage and memory overheads, and with a stable performance. Importantly, the increase of process variations of future technology nodes and the trend of including a growing number of pages in a single block let us envision an even more significant lifetime extension in future flash memories.

2 Related Work

Flash lifetime is one of the main concerns of these devices and is becoming even more worrisome today due to the increasing variability and retention capability inherent to smaller technology nodes. Most of the techniques trying to improve the device lifetime focus on improving the ECC robustness [15, 26], on reducing garbage collection overheads [14, 25], or on improving traditional wear-leveling techniques [20]. All of these contributions are complementary to our technique.

Lue et al. suggest to add a built-in local heater on the flash circuitry [16], which would heat cells at 800 ° C for milliseconds to accelerate the healing of the accumulated damage on the oxide layer that isolates the floating gates. Based on prototyping and simulations, the authors envision a flash cell endurance increase of several orders magnitude. While the endurance improvement is impressive, it would require significant efforts and modifications in current flash architectures before being available on the market. Furthermore, further analysis (e.g., power, temperature dissipation, cost) might reveal constraints that are only affordable for a niche market, whereas our technique can be used today with off-the-shelf NAND flash chips.

Wang and Wong [24] combine the healthy pages of multiple bad blocks to form a smaller set of virtually healthy blocks. In the same spirit, we revive *Multi-Level Cell* (MLC) bad blocks in *Single-Level Cell* (SLC) mode in a previous work [11]: writing a single bit per cell is more robust and can sustain more stress before a cell becomes completely unusable. Both techniques wait for blocks to turn bad before acting, which somehow limits their potentials (17% lifetime extension at best); on the other hand, by relieving early the weakest pages, we benefit more from the strongest cells and thus show a better lifetime improvement.

Pan et al. acknowledge the block endurance variance and suggest to adapt classical wear-leveling algorithm to compare blocks on their *Bit Error Rate* (BER) rather than their P/E cycles count [20]. However, in order to monitor a block BER, the authors assume homogeneous page endurance and a negligible faulty bit count variance between P/E cycles. For the two chips we studied, both assumptions were not applicable and would require a more complex approach to compare the BER of multiple blocks. Furthermore, we observed a significantly larger endurance variance on the page level than the block level. Hence, by acting on the page endurance, our approach has more room to expand the device lifetime.

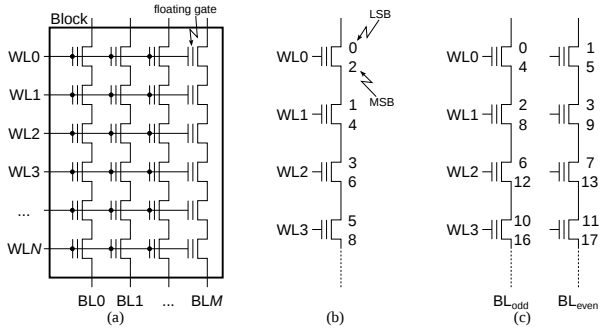


Figure 2: **Flash cells organization.** Figure 2(a) shows the organization of cells inside a block. A block is made of cell strings for each bitline (BL). Each bit of an MLC is mapped to a different page. Figures 2(b) and 2(c) show two examples of cell-to-page mappings in 2-bit MLC flash memories. For instance, in Figure 2(b), the LSB and MSB of WL_1 are mapped to pages 1 and 4, respectively. The page numbering also gives the programming order.

In this work, for more efficiency, we restrict the relief mechanism to data that is frequently updated, which is a strategy shared with techniques proposing to allocating those data in SLC-mode (i.e., programming only one bit per cell) to reduce the write latency [9, 10]. In a previous work, we characterized the effect of the SLC-mode and observed that it could write more data for the same amount of wear compared to regular writes and provided a lifetime improvement of up to 10% [10]. In this work, we propose to go further in the lifetime extension.

3 NAND Flash

NAND flash memory cells are grouped into pages (typically 8–32 kB) and blocks of hundreds of pages. Figure 2(a) illustrates the cell organization of a NAND flash block. In current flash architectures, more than one page can share the same *WordLine* (WL). This is particularly true for *Multi-Level Cells* (MLC), where the *Least Significant Bits* and *Most Significant Bits* (LSB and MSB) of a cell are mapped to different pages. Figures 2(b) and 2(c) show two cell-to-page mappings used in MLC flash devices, *All-BitLine* (ABL) and *interleaved*, respectively.

Flash memories store information by using electron tunneling to place and remove charges into floating gates.

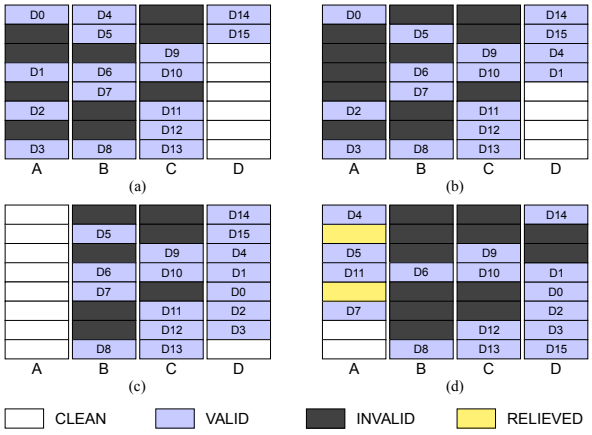


Figure 3: **Pages state transitions.** Figure (a) shows the various page states found in typical flash storage: *clean* when it has been freshly erased, *valid* when it holds valid data, and *invalid* when its data has been updated elsewhere. In Figure (b), data D1 and D4 are invalidated from blocks A and B, and updated in block D. In Figure (c), block A is reclaimed by the garbage collector; its remaining valid data are first copied to block D, before block A gets erased. Figure (d) illustrates the mechanism proposed in this work: we opportunistically relieve weak pages to limit their cumulative stress.

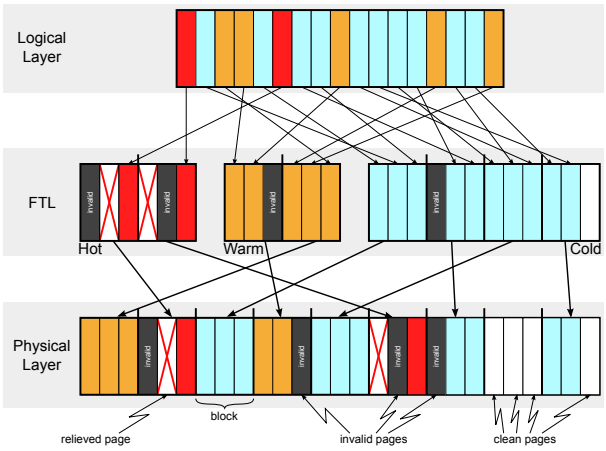


Figure 4: **Flash Translation Layer example.** An example of page-level mapping distinguishing update frequencies in three categories: *hot*, *warm* and *cold*. In this work, we propose to idle the weakest pages when their corresponding block is allocated to the hot partition. It limits the capacity loss to a small portion of the storage but still benefits from high update frequency to increase page-relief opportunities.

The action of adding a charge to a cell is called *programming*, whereas its removal is called *erasing*. Reading and programming cells is performed on the page level, whereas erasing must be performed on an entire block. Furthermore, pages in a block must be programmed sequentially. The sequence is designed to minimize the programming disturbance on neighboring pages, which receive undesired voltage shifts despite not being selected. In the sequences defined by both cell-to-page mappings, the LSBs of WL_{i+1} are programmed before the MSBs of WL_i. In this manner, any interference occurring between the WL_i LSB and MSB program will be inhibited after the WL_i MSB is programmed [17].

Importantly, the flash cells have limited endurance: they deteriorate with P/E cycles and become unreliable after a certain number of such cycles. Interestingly, the different pages of a block deteriorate at different rates, as shown in Figure 1. This observation serves as motivation for this work, which proposes a technique to reduce the endurance difference by regularly relieving the weakest pages.

3.1 Logical to Physical Translation

Flash Translation Layers (FTLs) hide the flash physical aspects to the host system and map logical addresses to physical flash locations to provide a simple interface similar to classical magnetic disks. To do this, the FTL needs to maintain the state of every page—typical states are *clean*, *valid*, or *invalid*, as illustrated in Figure 3(a). Only clean pages (i.e., erased) can be programmed. Invalid and valid pages cannot be reprogrammed without being erased before, which means the FTL must always have clean pages available and will direct incoming writes to them. Whenever data is written, the selected clean page becomes valid and the old copy becomes invalid. This is illustrated in Figure 3(b), where D1 and D4 have been reallocated.

To enable our technique, we introduced a fourth page state, *relieved*, to indicate pages to be relieved (i.e., not programmed) during a P/E cycle. Relieving pages during a P/E cycle is perfectly practical, because it does not break the programming sequentiality constraint and does not compromise the neighbors information. In fact, it is electrically equivalent to programming a page to the erase state (i.e., all 1's). Hence, to the best of our knowledge, any standard NAND flash architecture should support this technique.

3.2 Garbage Collection

The number of invalid pages grows as the device is written. At some point, the FTL must trigger the reuse of invalid pages into clean pages. This reuse process is known as garbage collection, which is illustrated in Figure 3(c), where block A is selected as the victim.

Copying the remaining valid data of a victim block represents a significant overhead, both in terms of performance and lifetime. Therefore, it is crucial to select the data that will be allocated onto the same block carefully in order provide an efficient storage system. Wu and Zwaenepoel addressed this problem by regrouping data with similar update frequencies [25]. *Hot* data have a higher probability of being updated and invalidated soon, resulting in *hot* blocks with a large number of invalid pages that reduce the garbage collection overhead. Figure 4 shows an example FTL that identifies three different *temperatures* (i.e., update frequencies), labeled as *hot*, *warm*, and *cold*. Literature is rich with heuristics to identify hot data [12, 4, 9, 22, 21].

In the present study, we propose to relieve the weakest pages in order to balance their endurance with their stronger neighbors. We have restricted the relieved pages to the hottest partition in order to limit the resulting capacity loss to a small and contained part of the storage, while benefiting from a large update frequency to better exploit the presented effect. Following sections will further analyze the costs and benefits of our approach, as well as its challenges.

3.3 Block Endurance

While accumulating P/E cycles, a block becomes progressively less efficient in the retention of charges and its BER increases exponentially. Typically, flash blocks are considered unreliable after a specified number of P/E cycles known as the endurance. Yet, it is well understood that the endurance specified by manufacturers serves as a certification but is hardly sufficient to evaluate the actual endurance of a block [8, 18]. A block endurance depends on the following factors: First, the cell design and technology will define its resistance to stress; this is generally a trade-off with performance and density. Second, the endurance is associated with a retention time, that is, how long data is guaranteed to remain readable after being written; a longer retention time requirement will require relatively healthy cells and limit the endurance to lower values. Finally, ECCs are typically used to correct a limited number of errors within a page; the ECC strength (i.e., number of correctable bits) influences the block endurance. The ECC strength required to maintain the endurance specified by manufacturers increases drastically at every new technology nodes. A stronger ECC grows in size and requires a more complex and longer error decoding process, which compromises read latency. Additionally, the strength of an ECC is chosen according to the weakest page of a block and, as suggested by Figure 1, the chosen strength will only be justified for a minority of pages. Our proposed balancing of page endurance within a block will reduce the BER of the weakest pages; therefore, our idea can either be used to reduce the ECC strength requirement or to extend the device lifetime. However, in this work, we only explore the impact of our technique in device lifetime extension.

FTLs implement several techniques that maximize the use of this limited endurance to guarantee a sufficient device lifetime and reliability. Typical wear-leveling algorithms implemented in FTLs target the even distribution of P/E counts over the blocks. Additionally, to avoid latent errors, *scrubbing* [1, 23] may be used, which consists in detecting data that accumulates too many errors and rewriting it before it exceeds the ECC capability.

3.4 Bad Blocks

A block is considered *bad* whenever an erase or program operation fails, or when the BER grows close to the ECC capabilities. In the former case, an operation failure is notified by a status register to the FTL, which reacts by marking the failing block as bad. In the latter case, despite a programming operation having been completed successfully, a certain number of page cells might have become too sensitive to neighboring programming disturbances or have started to leak charges faster than the specified retention time and will compromise the stored data [17]. Henceforth, the FTL will stop using the block and the flash device will die at the point in time when no spare blocks remain to replace all failing blocks.

To study the degradation speed of the different pages within a block, we conducted an experiment on a real NAND flash chip in which we continuously programmed pages with random data and monitored each page BER by averaging their error counts over 100 P/E cycles. We have already anticipated the results in Figure 1, which shows how the number of error bits increases with the number of P/E operations for all the pages in a particular block. At some point in time, the weakest page (darker line on the graph) will show a BER that is too high and the entire block will be considered unreliable. Interestingly, a large majority of the remaining pages could withstand a significant amount of extra writes before becoming truly unreliable. Clearly, flash blocks suffer a premature death if no countermeasures are taken and our approach attempts to postpone the moment at which a page block becomes bad by proactively relieving its weakest pages. The following sections further study the degradation process of individual pages and detail the technique that uses strong pages to relieve weak ones.

4 Relieving Pages

In this section we introduce the relief strategy and characterize its effects from experiments on two real 30-nm class NAND flash chips.

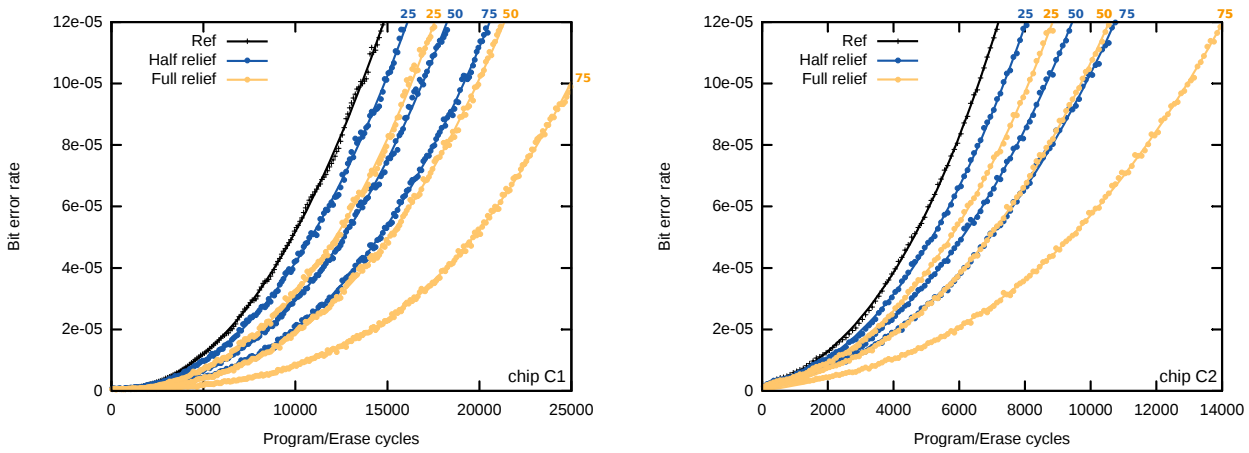


Figure 5: **Measured effect of relieving pages.** The degradation speed for various relief rates and types are measured on both chips. The *Ref* curve reports the BER of the entire reference blocks, whereas for the relieved blocks, the BER is only evaluated on the relieved page. The labels '25', '50', and '75' indicate the corresponding relief rate in percent. The BER is evaluated over a 100-cycle period.

4.1 Definition

We define a *relief* cycle on a page the fact of not programming it between two erase cycles. Although relieved pages are not programmed, they are still erased, which, in addition to the disturbances coming from neighbors undergoing normal P/E cycles, generates some stress that we characterize in Section 4.2. In the case of MLC, the cells are mapped to an LSB and MSB page pair and can either be *fully* relieved, when both pages are skipped, or *half* relieved, when only the MSB page is skipped. The level of damage done to a cell during a P/E cycle is correlated to the amount of charge injected for programming; of course, more charges means more damage to the cell. Therefore, a page will experience minimal damage during a *full* relief cycle while a *half* relief cycle will apply a stress level somewhere between the *full* relief and a normal P/E cycle.

4.2 Understanding the Relieving Effect

In order to characterize the effects of relieving pages, we selected two typical 32 Gb MLC chips from two different manufacturers. We will refer them as C1 and C2; their characteristics are summarized in Table 1. The read latency, the block size, and the cell-to-page mapping architecture are the most relevant differences between the two chips. The C1 chip has slower reads and smaller blocks than C2, and it implements the *All-Bit Line* (ABL) architecture illustrated in Figure 2(b). The C2 chip implements the *interleaved* architecture illustrated in Figure 2(c). We design an experiment to measure on our flash chips how the relief rate impacts the page degradation speed. Accordingly, we selected a set of 28 blocks

Table 1: MLC NAND Flash Chips Characteristics

Features	C1	C2
Total size	32 Gb	32 Gb
Pages per block	128	256
Page size	8 kB	8 kB
Spare bytes	448	448
Read latency	150 μs	40-60 μs
LSB write lat.	450 μs	450 μs
MSB write lat.	1,800 μs	1,500 μs
Erase latency	4 ms	3 ms
Architecture	ABL	interleaved

and divided them into seven sets of four blocks each. One set is configured as a reference, where blocks are always programmed normally—i.e., no page is ever relieved. We allocate then three sets for each of the two relief types (i.e., *full* and *half*), and each of these three sets is relieved at a different frequency (25%, 50% and 75%). For each relieved block, only one LSB/MSB page pair out of four is actually relieved, while the others are always programmed normally. Therefore, the relieved page pairs are isolated from each other by three normally-programmed page pairs. Hence, we take into account the impact of normal neighboring pages activity on the relieved pages. Furthermore, within each four-block relieved sets, we alternate the set of page pairs that are actually relieved in order to evaluate evenly the relief effects for every page pair physical position and discard any measurement bias. Finally, every ten P/E cycles we enforce a regular program cycle for every relieved blocks (including relieved pages) in order to average out the absence of disturbance coming from relieved neighbors and collect unbiased error counts for every page. Indeed,

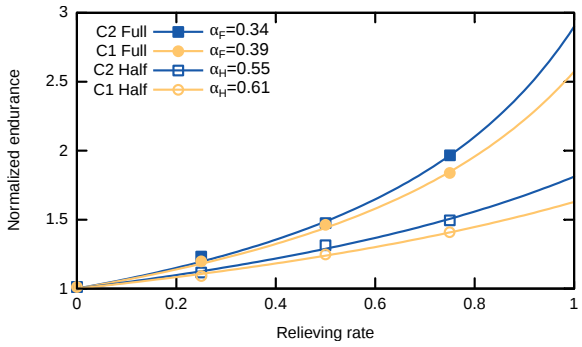


Figure 6: **Normalized page endurance vs. relief rate.** The graph shows how relieving pages extends their endurance. The endurance is normalized to the normal page endurance, corresponding to a maximum BER of 10^{-4}. For each chip, the relative stress of the full and half relief type is extracted by fitting the measured points.

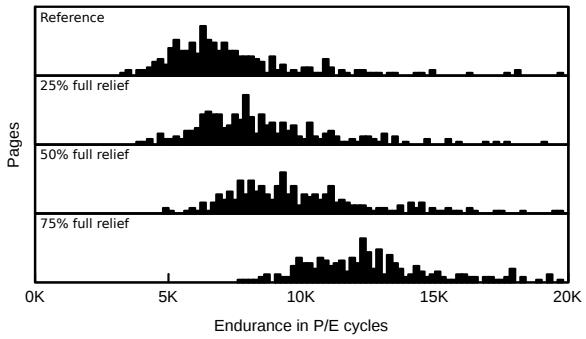


Figure 7: **Measured page endurance distribution.** The clusters on the left and right correspond to MSB and LSB pages, respectively. Both clusters endurance are extended homogeneously when relieved.

pages close to relieved pages experience less disturbance and show a significantly lower BER.

Figure 5 shows the evolution of the average BER with the number of P/E cycles for every set of blocks as measured on the chips. For the relieved sets, only the relieved pages are considered for the average BER evaluation. Clearly, the relief of pages slows down the degradation compared to regular cycles and extends the number of possible P/E cycles before reaching a given BER.

In order to model the stress endured by pages undergoing a full or half relief cycle, we first define the relationship between page endurance and the stress experienced during a P/E cycle. The endurance E of a page is inversely proportional to the stress ω that the page receives during a P/E cycle:

$$E = \frac{1}{\omega}. \tag{1}$$

Considering a page being relieved with a relative stress α at a given rate ρ, the resulting extended endurance E_X is expressed as the inverse of the average stress:

$$E_X(\rho,\alpha) = \frac{1}{(1-\rho)\omega + \rho\alpha\omega} = \frac{E}{(1-\rho)+\rho\alpha}. \tag{2}$$

Assuming a maximum BER of 10^{-4} to define a page endurance, we show in Figure 6 the endurance of relieved pages for the three relief rates measured, with the endurance normalized to the reference set. For each chip, we also fit the data points to the model of Equation (2) and report the extracted α parameters on the figure. Consistently across the two chips, a full relief incurs less damage to the cell than a half relief, which in turn incurs less damage than regular P/E cycles. Interestingly, half reliefs are more efficient than full reliefs in term of stress per written data: for example, for chip C1, the fraction of stress associated to half and full relief cycles is $\alpha_H = 0.61$ and $\alpha_F = 0.39$, respectively. Over two P/E cycles, if an LSB/MSB page pair gets twice half relieved or once fully relieved, two pages would have been written in both cases but the cumulated stress would be larger with a full relief:

$$2 \cdot \alpha_H = 1.22 < 1.39 = 1 + \alpha_F. \tag{3}$$

Furthermore, a half relief cycle consists in programming solely the LSB of a LSB/MSB pair, and, intrinsically, programming the LSB has a significantly smaller latency than the MSB (see Table 1). Thus, a half relief is not only more efficient for the same amount of written data, but it also displays better performance.

Figure 7 provides further insight on the relief effect on a page population. The figure shows the number of P/E cycles tolerated by the different pages before reaching an BER of 10^{-4} evaluated over 100 P/E cycles.

In the next sections we will discuss how relief cycles can opportunistically be implemented into common FTLs to balance the page endurance and improve the device lifetime.

5 Implementation in FTLs

In this section, we describe the implementation details required to upgrade existing FTL with our technique.

5.1 Mitigating the Capacity Loss

Relieving pages during a P/E cycle temporarily reduces the effective capacity of a block. Therefore, relieving pages in a block-level mapped storage would be impractical. Conversely, performing it on blocks that are mapped to the page level (or finer level) is straightforward. Consequently, in order to limit the total capacity loss while still being able to frequently relieve pages,

we propose to exclusively enable relief cycles in blocks that are allocated to the hottest partition, where the FTL writes data identified as very likely to be updated soon.

Actually, the hot partition is an ideal candidate for our technique because of two reasons: (1) hot data generally represent a small portion of the total device capacity (e.g., less than 10%), which bounds the capacity loss to a small fraction; also, (2) hot partitions usually receive a significant fraction of the total writes (our evaluated workloads show often more than 50% of writes identified as hot), which provides plenty of opportunities to relieve pages. Note that flash blocks are dynamically mapped to the logical partitions, and thus, all of the physical blocks in the device will eventually be allocated to the hottest partition. Furthermore, classical wear-leveling mechanisms will regularly swap cold blocks with hot blocks in order to balance their P/E counts. Accordingly, our technique has a global effect on the flash device despite acting only on a small logical partition.

We will now describe two different approaches to balance the page endurance with our relief strategies. The first one can be qualified as *reactive*, in that it will regularly monitor the faulty bit count to identify weak pages. The second one, which we call *proactive*, estimates beforehand what the endurance of every page will be and sets up a relief plan that can be followed from the first P/E cycle. Currently, manufacturers do not provide all the information that would be required to directly specify the parameters needed for our techniques. Until then, both techniques would require some characterization of the chips to be used in order to extract parameters α_F and α_H, and the page endurance distribution.

5.2 Identifying Weak Pages on the Fly

The *reactive* relief technique relies on the evolution of the page BER to detect weakest pages as early as possible. The FTL must therefore periodically monitor the amount of faulty bits per page which is very similar to the scrubbing process [1]. This monitoring happens every time that a cold (i.e., non-*hot*) block is selected by the garbage collector. Concretely, we must read every page and collect the error counts reported by the ECC unit before erasing a block.

A simple approach to identify the weakest pages is to detect which ones reach a particular error threshold first. Assuming that an ECC can handle up to n faulty bits per page, we can set an intermediate threshold k, with $k < n$, that can be used to flag pages getting close to their endurance limit. The parameter n is given by the strength of the ECC in place, while the parameter k must be chosen to maximize the efficiency of the technique and will depend on the page endurance variance. As soon as a page reaches the threshold k, our heuristic will systematically relieve the corresponding LSB/MSB page pair when it is allocated to the hot partition. In order to control the capacity loss, we also set a maximum amount of pages to relieve per block; only the r first pages reaching the threshold within a block will get relieved. For our evaluation, we bound the relieved page count, r, to 25% of the block capacity. A larger r would increase the range of pages that can be relieved but decrease the efficiency of the buffer. Besides, the latest pages to be identified as weak do not require a relief as aggressive than the weakest ones. Hence, we propose to fully relieve the r_h first weak pages and to half relieve the remaining $r - r_h$ pages. In our case, we found the best compromise with r_h equal to 5% and 10% of the block capacity for C1 and C2, respectively. Choosing efficiently r_h for a new chip requires the information on its page endurance distribution. The larger is its variance, the larger r_h should be.

The *reactive* approach requires extra storage for its metadata. This overhead includes two bits per LSB/MSB page pair, which will indicate whether any of the pages has reached the k threshold and whether it should be fully or half relieved, and a (redundant) counter indicating the number of detected weak LSB/MSB page pairs so far. Accordingly, 133 extra bits (128 bits for the flags and 5 bits for the counter) per block will need to be stored in a device containing 128-page blocks. In the concrete case of C1, for instance, this extra storage corresponds to an insignificant amount of the total 458,752 spare bits that are available for extra storage in every block. Additionally, the FTL main memory will need to temporally store the practically insignificant metadata of a single block to be able to restore the metadata after erasing the block. Overall, the extra storage needed by this technique appears to be negligible in typical flash devices.

The monitoring required by this technique needs the FTL to read a whole block before erasing it, which adds an overhead to the erasing time. The monitoring represents an overhead of 10% of the total time spent writing cold data, since flash read latency is typically ten times smaller than write latency. However, the monitoring process can often be performed in the background, making this estimation—which we will use in all of our experiments—quite conservative. If hiding the monitoring in the background is not feasible or not sufficiently effective, the FTL can also monitor the errors only every several erase cycles. Accordingly, we evaluated how the lifetime improvement is affected by a limited monitoring frequency and observed that a monitoring frequency of 20% (i.e., blocks are monitored once every five P/E cycles) provides sufficient information to sustain the same lifetime extension than full monitoring. In substance, while the process of identifying the weakest pages could at worst require one page read per page written, simple

	Plan 0 (ρ_0=60%)		Plan 1 (ρ_1=75%)		Plan 2 (ρ_2=90%)	
	4000 cycles		2000 cycles		2000 cycles	
Page #	Half rel.	Full rel.	Half rel.	Full rel.	Half rel.	Full rel.
0	-	-	-	-	-	-
1	-	-	40%	-	60%	40%
2	-	-	-	-	-	-
3	30%	-	-	100%	60%	40%
4	-	-	-	-	-	-
5	-	100%	-	100%	60%	40%
6	-	-	-	-	-	-
7	-	-	-	-	-	-
8	-	-	-	-	-	-
9	-	-	30%	-	60%	40%
10	-	-	-	-	-	-
11	-	-	-	-	100%	-
12	-	-	-	-	-	-
13	90%	10%	-	100%	60%	40%
14	-	-	-	-	-	-
15	-	-	-	-	-	-

Figure 8: **Example of a relief plan.** The relief plan is actually made of several plans, each valid for a given amount of relief cycles. According to this plan, blocks will follow Plan 0 during the first 4000 relief cycles then move on to Plan 1 for the next 2000 relief cycles and so on. A plan provides for each page its probability to be relieved. In the example, page 5 is the weakest page and is relieved to the maximum in Plan 0 and Plan 1.

techniques can reduce this overhead to negligible levels without a loss in the effectiveness of the idea.

5.3 Relief Planning Ahead of Time

The *reactive* approach requires to identify the weakest pages during operation and while significant deterioration has already occurred, which somehow limits the potential for relief. More efficient would be to relieve the weakest pages from the very first writes to the device. Interestingly, previous work observed noticeable BER correlation with the page number [7, 3]. Similarly, we observe on our chips a significant correlation between a page position in a block and its endurance. This correlation is important enough to allow us to rank every page per endurance. Thereby, we developed a *proactive* technique to exploit the relief potential more efficiently.

The *proactive* technique requires first a small analysis of the flash chip that we consider. We must characterize the endurance of LSB/MSB page pairs in every position in a block, for a given BER. For each page pair, only the shorter page endurance is considered. This information can be extracted from a relatively small set of blocks (e.g., 10 blocks). Thanks to this information, we will be able to rank the page pairs by their endurance and know which page should be relieved the most. Yet, building an efficient relief plan would also require the knowledge of how many times a block will be allocated to the hot partition during its lifetime, which corresponds to the amount of opportunities to relieve its weakest pages. With this information, one could evaluate to what extent the weakest page of a block can be relieved and how many times the other pages should be relieved to meet the same extended endurance. However, in practice, one cannot have this information ahead of time. Instead, we prepare a sequence of plans targeting increasing hot allocation counts; Figure 8 gives an example of such a sequence. In this example, Plan 0 contains the relief information for the first 4000 relief cycles. Once a block has been allocated to the hot partition 4000 times, one moves to Plan 1 for the next 2000 relief cycles. The entries in the plans are probabilities for a page to be either fully relieved, half relieved, or normally programmed. Hence, when a block is allocated to the hot partition, before programming a page, one should first consult the plan and decide whether or not the current page should be skipped.

To create such plans, sequentially starting from Plan 0, we first refer to the page pairs endurance analysis to identify the weakest pair position w. Each Plan p is built assuming an intermediate hot allocation ratio ρ_p (e.g., 60% for Plan 0) that grows from one plan to the next. The higher it is, the more flexible the plan will be and applications with large hot ratios will largely benefit from half relief cycles, while applications with low hot ratios will not be relieved as aggressively as they should. After choosing a ratio, we evaluate the maximum possible endurance extension with full relief for the weakest page pair w, $E_{T,p} = E_{X,w}(\rho_p, \alpha_F)$. The expected number of relief cycles for this Plan p is thus $L_p = \rho_p \cdot E_{X,w}$ minus the total length of the previous plans. Hence in the example, the hot allocation ratio ρ_1 of Plan 1 would provide 2000 more relief cycle than Plan 0. Thereby, when a block exceeds 4000 relief cycles before turning bad, it means that the actual ρ is larger than ρ_0 and the block should move on to the next plan, which targets a higher ρ.

Once the target endurance is set, for every page pair i having an endurance E_i lower than $E_{T,p}$, we compute the number of relief cycles R_i that would be required for them to align their endurance to $E_{T,p}$. Setting

$$E_{X,i}(\rho_i, \alpha) = \frac{E_i}{(1-\rho_i)+\rho_i\alpha} = E_T \tag{4}$$

and considering that $\rho_i = R_i/E_T$, we simply obtain

$$R_i = \frac{E_T - E_i}{1-\alpha}. \tag{5}$$

Here, α is the fraction of stress corresponding to half or full relief cycles, or to a combination of the two, and we still need to decide which type of relief to use.

As discussed in Section 4.2, half relief is most efficient in terms of avoided stress per written data and in terms of performance, and, hence, we will maximize its usage. For every page i to be relieved, we evaluate with Equation (5) and $\alpha = \alpha_H$ the number of half relief cycles that

would be necessary to reach the endurance $E_{T,p}$. If the required number of half relief cycles is larger than the number of relief cycles in this plan L_p, we are forced to consider some full relief as well. Trivially, from Equation (5) and with $L_p = R_i$, we determine the fraction λ of full relief cycles such that the average fraction of stress is

$$\alpha = \lambda\,\alpha_F + (1-\lambda)\,\alpha_H = 1 - \frac{E_T - E_i}{L_p}. \qquad (6)$$

To construct Plan $p+1$, every page that was relieved, even partially, according to Plan p will be set to the maximum relief rate (i.e., 100% full relief), and the above process is repeated.

Similarly to the *reactive* approach, we restrict to r the maximum number of relieved pages in order to limit the potential performance drop. For the *proactive* technique, we can solely evaluate what would be the average number of pages relieved per plan by summing every page probability to get relieved. For example, in Figure 8, for Plan 0 the average number of relieved pages is $2 \cdot (1+0.1) + 0.3 + 0.9 = 3.4$ pages out of 32 (remember that a full relief skips two pages). Limiting the average number of pages relieved will at some point bound the target endurance. This is illustrated in Figure 8 with Plan 2. Assuming that a maximum of eight pages on average is allowed, the original $E_{T,2}$ would have required the number of relieved pages to be larger than this. Hence the $E_{T,2}$ is reduced to meet the requirements, which reduces the relief rate of every page to meet the average of eight relieved pages per cycle. The plan that requires to reduce its original target endurance becomes the latest plan. Once a block completed this last plan, it will simply stop having to relieve any page until the end of its lifetime.

This technique requires to store the plans in the FTL memory. Each plan has two entries for each LSB/MSB pair and each entry can be encoded on 8 or 16 bits, depending on the desired precision, resulting in 256–512 Bytes per plan, which is negligible for most environments. Besides, the tables are largely sparse and could be further reduced by means of classical compression strategies (e.g., hash tables) to fit in memory sensitive environments.

6 Experiments and Results

We evaluate here the expected lifetime extension achievable with the two relief strategies presented. In the next sections, we explain how we begin by combining error traces acquired from real NAND flash chips with simulation to obtain a first assessment of the improvements of block endurance and, consequently, of device lifetime. We then refine our experimental methodology by implementing a trace-driven simulator and a couple of state-of-the-art FTLs, and by evaluating more accurately the impact of our technique. We use a number of benchmarks to show not only the lifetime improvement but also the minimal effect (often favorable) of our technique on execution time.

6.1 Collecting Traces and Simulating Wear

To assess the impact of our technique, we first collected real error traces from 100 blocks from each of our chips that went through thousands of regular P/E cycles; we collected the error count of every page at every P/E cycle. We then used the collected traces to simulate what would happen of the blocks when going through P/E cycles during normal use of the device. At each simulated P/E cycle, each block is either allocated to the hot partition (i.e., where pages can be relieved) or to the cold one, depending on a hot-write probability; this parameter simulates the behaviour of an FTL and defines the probability for a block to be allocated to the hot partition. When a block is allocated to the cold partition, a normal P/E cycle occurs: every page is considered programmed. When a block is allocated to the hot partition, the weak pages are relieved instead. The *reactive* approach uses the error counts to determine pages as weak if they have reached the predefined threshold k. The *proactive* approach, on the other hand, relies solely on the relief plans prepared in advance to determine the weak pages to be relieved. While we simulate successive writes to the device, we count how many times each page has been written and to what extent it has been relieved. Whenever our real traces tell us that one page of a block has reached a given BER, considered as the maximum correctable BER, we render the block as bad and stop using it. At the end, the simulator reports the total amount of data that could be written in each block—that is, the lifetime of the block under a realistic usage of the device.

6.2 Block Lifetime Extension

We use our wear simulation method to first evaluate the lifetime enhancement provided by our techniques at the block level. In this context, we consider a block to be bad as soon as one of its pages reaches the given BER. Considering a 60% hot write ratio, Figure 9 shows the lifetime of every block for both our flash chips assuming a maximum BER of 10^{-4}; it compares our *proactive* and *reactive* techniques to the baseline. The blocks are ordered on the x-axis with the one with the lowest lifetime on the left up to the one with the largest on the right. The bottom curve is the lifetime of each block when stressed normally, while the two curves on the top corresponds to the lifetime when applying our techniques. The relief effectiveness varies depending on the actual block,

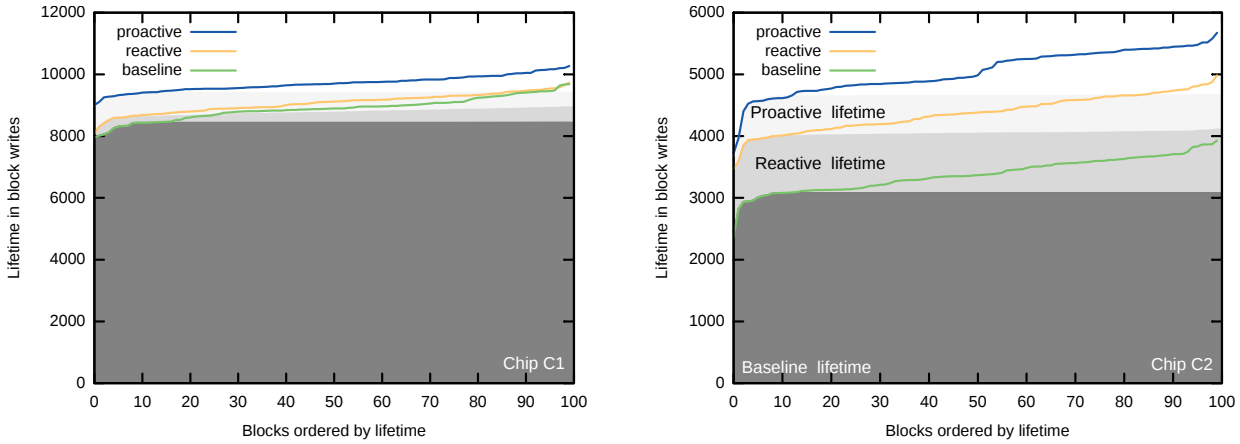


Figure 9: **Block lifetime improvement.** The curves show the individual block lifetime, and the surface areas the device lifetime, assuming it can cumulate up to 10% bad blocks. As expected, the *proactive* technique is more efficient than the *reactive* one. Chip C1 has a relatively small page endurance variance, which limits the efficiency of the *proactive* approach to 10% lifetime extension. Comparatively, C2 offers more room to exploit the relief mechanism and allows the *proactive* approach to extend by 50% the lifetime. For these graphs, we assume a limit BER of 10^{-4} as well as a 60% write frequency to the hot partition.

thereby the block ordering for the two curves is not necessarily the same. The *proactive* approach is more efficient, as it starts relieving pages much sooner than the *reactive* approach. Yet, we believe that there is room to improve our simple weak-page detection heuristic in order to act sooner and be more efficient. Chip C1 shows a relatively small page endurance variance, which limits our techniques potential with a lifetime improvement of 10% maximum. This confirms the intuition that a larger page endurance variability and a greater number of pages per block (double for C2 compared to C1) increase the benefit of the presented techniques. In the next section, we translate the block lifetime extension into a device lifetime extension.

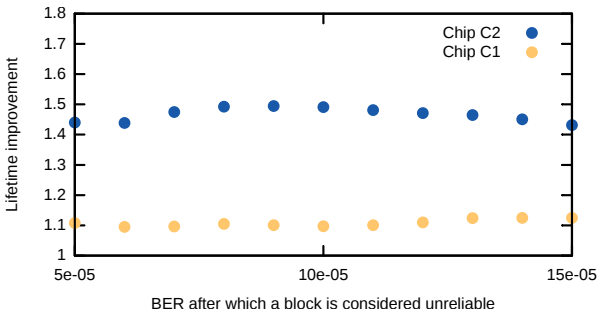


Figure 10: **Lifetime improvement w.r.t. BER threshold.** The BER threshold that indicates when a block is considered unreliable directly affects a device lifetime. Large BER thresholds increase the baseline lifetime and remove room to improvement at the cost of a more expensive ECC.

6.3 Device Lifetime Extension

We now evaluate the lifetime extension for a set of blocks when relieving the weakest pages. The three grey areas of Figure 9 represent the total amount of data we could write the device during its lifetime using the baseline and our relief techniques. Assuming that the device dies whenever 10% of its blocks turn bad, the ratio of a relief gray area with the baseline area represents the additional fraction of data that we could write: for C2, our *reactive* and *proactive* techniques show a lifetime improvement of more than 30% and 50%, respectively. These results are obtained from a sample of 100 blocks, which are enough to provide an error margin of less than 3% for a 95% confidence level. From this figure, we can also make a quantitative comparison between the error rate leveling technique proposed by Pan et al. [20]. If we were to perfectly predict the endurance of every block, we would have a device lifetime that is equal to each individual block lifetime and which corresponds to the total area below the baseline curve. Accordingly, we would get an extra lifetime of 5% and 11% for C1 and C2, respectively, which is an optimistic estimate, yet significantly lower than what the *proactive* approach can bring.

We performed a sensitivity analysis on several parameters that might have an effect on the lifetime extension. For the following results, we focus on the *proactive* strategy. The proportion of bad blocks tolerated by a device had negligible effect on the lifetime extension. As for the

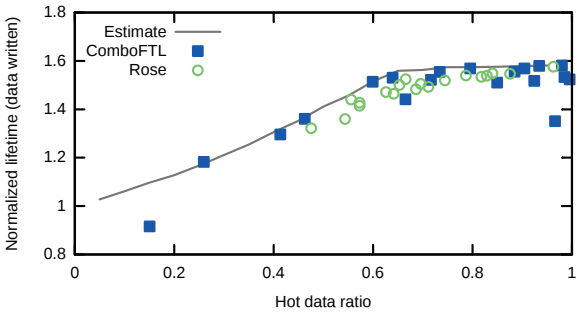


Figure 11: **Lifetime improvement w.r.t. hot write ratio.** The curve gives the expected lifetime extension provided by the *proactive* technique on chip C2. The data points represent results from benchmarks using two different FTLs. Those measurements take into account the writes overhead caused by the hot partition capacity loss. Apart from a couple of outliers, the results are consistent with our expectations.

BER threshold, the effect on lifetime extension is moderate, as illustrated in Figure 10. A larger BER gives more time to benefit from relieving pages, but it also increases the reference lifetime and makes the relative improvement smaller. Finally, the hot write ratio sets by how much our technique can be exploited and has a significant effect on the lifetime extension. The curve labeled "Estimate" in Figure 11 shows the lifetime of a device implementing the *proactive* technique (normalized to the baseline lifetime) as a function of the hot write ratio. We clearly see that the more writes are directed to the hot partition, the better the relief properties can be exploited, as one would expect. The data points on the figure represent the normalized lifetime extension when considering the actual execution of a set of benchmarks with real FTLs, which will be introduced in the next section; these measurements take into account all possible overheads derived from the implementation of the relief technique and match well the simpler estimate. All results show significant lifetime extensions for hot write ratios larger than 40% which is, in fact, in the range where most benchmarks (with very rare exceptions) are in practice.

6.4 Lifetime and Performance Evaluation

The temporary capacity reduction in the hot partition produced by relieving pages decreases its efficiency and is very likely to trigger more often the garbage collector. This effect is more critical for hybrid mapping FTLs that rely on block-level mapping for the cold partition: these FTLs will need to write a whole block even when a single page needs to be evicted from the page-level mapped hot partition (buffer partition) to the block-level mapped cold partition. To refine our estimations and understand the impact on performance, we developed a trace-driven flash simulator and implemented two hybrid FTLs, namely ComboFTL [9] and ROSE [5]. Both FTLs have a hot partition that is mapped to the page level, however their cold partitions are mapped differently. ROSE maps its cold data at the block level, while ComboFTL divide its cold partition in sets of blocks, each being mapped at the page level. Additionally, ComboFTL has a warm partition; we will consider this third partition hot as well, in the sense that pages of blocks allocated to the warm partition will be subject to relief cycles when appropriate. Thanks to the block level mapping, ROSE requires significantly less memory than ComboFTL to be implemented but pays the cost with an execution time 25% larger and a 20% smaller lifetime in average.

In our experimental setup, we assume a hot partition allocating 5% of the total device size and we limit the maximum ratio of relieved pages to 25%, which represents a maximal loss of 1.25% of the total device capacity. Hence, the page relief cost can either be considered as extra capacity requirement (1.25% here) or in a garbage collection overhead that we will now evaluate for two different FTLs.

We selected a large set of disk traces to be executed by both FTLs. First the trace *homesrv* is a disk trace that we collected during eight days on a small Linux home server hosting various services (e.g., mail, file server, web server). The traces *fin1* and *fin2* [2] are gathered from OLTP applications running at two large financial institutions. Lastly, we selected 15 traces that have a significant amount of writes from the MSR Cambridge traces [19]. In our simulation, we assume a total capacity of 16 GBytes and a flash device with the characteristics of C2 (see Table 1). While most of the traces were acquired on disks of a larger capacity, their footprint are all smaller and by considering only the referenced logical blocks (2 MBytes for C2), every selected benchmark fitted in the simulated disk. Importantly, when simulating a smaller device, the hot partition size gets proportionally scaled down, which effectively reduces the hot write ratio and the potential of our approaches and renders the following results conservative.

For the experiments, we considered again a maximum BER of 10^{-4} and a bad blocks limit of 10%. We report in Figure 12 the performance and lifetime results for both chips and of both FTLs executing all the benchmarks with the *proactive* technique. The results are normalized to their baseline counterpart, that is implementing the same FTL without relieving weak pages. (Note that this makes the results for ComboFTL and ROSE not comparable between themselves, but our purpose here is not to compare different FTLs but rather to show that, ir-

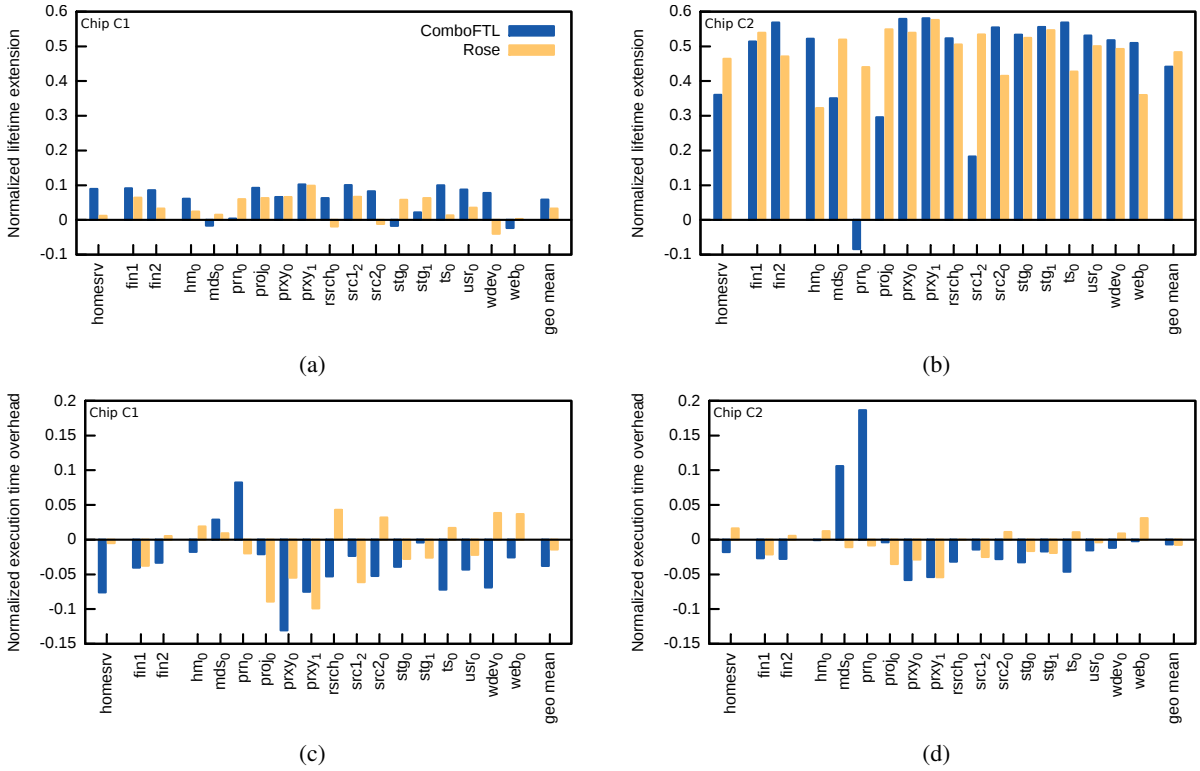


Figure 12: **Performance and lifetime evaluation of our *proactive* technique for various benchmarks running on both chips.** (a) Our relief technique gets at most 10% lifetime extension for the chip C1, (b) whereas for C2 it gives regularly an extra 50% lifetime, but for rare exceptions. In (c) and (d), we see that the execution time is stable for most of the benchmarks despite the capacity loss in the hot buffer. Thanks to the half relief efficiency, several benchmarks even sport a better performance.

respective of the particular FTL, our technique remains perfectly effective). Most of the benchmarks result in a hot write ratio larger than 50% and show a lifetime extension between 30% and 60% for C2. In particular, we observed that ComboFTL frequently fails to correctly identify hot data from the prn_0 trace; this results in a large amount of garbage collection, a poor hot data ratio, and a performance drop of 20% when relieving weak pages—ROSE performs significantly better here. Overall, despite this pathological case, the *proactive* relief technique brings an average lifetime extension of 45% and a execution time *improvement* within 1%. The execution time improvement comes thanks to the half relief efficiency, which provides significantly smaller write latencies. In summary, the *proactive* approach provides a significant lifetime extension with a stable performance and a negligible memory overhead.

7 Conclusion

In this paper, we exploit large variations in cell quality and sensitivity occurring in modern flash devices to extend the device lifetime. We better exploit the endurance of the strongest cells by putting more stress on them while periodically relieving the weakest ones of their duty. This gain comes at a moderate cost in memory requirements and without any loss in performance. The proposed techniques are a first attempt to benefit from page-relief mechanisms. While we already show a lifetime improvement of up to 60% at practically no cost, we believe that further investigation of the effects of our method on data retention as well as research on other wear unleveling techniques could help to further balance the endurance of every page and block. In future flash technology nodes, process variations will only become more critical and we are convinced that techniques such as the ones presented here could help overcome the upcoming challenges.

References

[1] AUCLAIR, D., CRAIG, J., GUTERMAN, D., MANGAN, J., MEHROTRA, S., AND NORMAN, R. Soft errors handling in EEPROM devices, Aug. 12 1997. US Patent 5,657,332.

[2] BATES, K., AND MCNUTT, B. OLTP application I/O, June 2007. http://traces.cs.umass.edu/index.php/Storage/Storage.

[3] CAI, Y., HARATSCH, E., MUTLU, O., AND MAI, K. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Design, Automation & Test in Europe Conf. & Exhibition* (Dresden, Germany, Mar. 2012), pp. 521–26.

[4] CHANG, L.-P. A hybrid approach to NAND-flash-based solid-state disks. *IEEE Trans. Computers 59*, 10 (Oct. 2010), 1337–49.

[5] CHIAO, M.-L., AND CHANG, D.-W. ROSE: A novel flash translation layer for NAND flash memory based on hybrid address translation. *IEEE Trans. Computers 60*, 6 (June 2011), 753–66.

[6] CHO, H., SHIN, D., AND EOM, Y. I. KAST: K-associative sector translation for NAND flash memory in real-time systems. In *Design Automation and Test in Europe* (Nice, France, Apr. 2009), pp. 507–12.

[7] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: Anomalies, observations, and applications. In *ACM/IEEE Int. Symp. Microarchitecture* (New York, NY, USA, Dec. 2009), pp. 24–33.

[8] HETZLER, S. R. Flash endurance and retention monitoring. In *Flash Memory Summit* (Santa Clara, CA, USA, Aug. 2013).

[9] IM, S., AND SHIN, D. ComboFTL: Improving performance and lifespan of MLC flash memory using SLC flash buffer. *Journal of Systems Architecture 56*, 12 (Dec. 2010), 641–53.

[10] JIMENEZ, X., NOVO, D., AND IENNE, P. Software controlled cell bit-density to improve NAND flash lifetime. In *Design Automation Conf.* (San Francisco, California, USA, June 2012), pp. 229–34.

[11] JIMENEZ, X., NOVO, D., AND IENNE, P. Phœnix: Reviving MLC blocks as SLC to extend NAND flash devices lifetime. In *Design, Automation & Test in Europe Conf. & Exhibition* (Grenoble, France, Mar. 2013), pp. 226–29.

[12] LEE, S., SHIN, D., KIM, Y.-J., AND KIM, J. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review 42*, 6 (Oct. 2008), 36–42.

[13] LEE, S.-W., PARK, D.-J., CHUNG, T.-S., LEE, D.-H., PARK, S., AND SONG, H.-J. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embedded Computing Systems 6*, 3 (July 2007).

[14] LIN, W., AND CHANG, L. Dual greedy: Adaptive garbage collection for page-mapping solid-state disks. In *Design, Automation & Test in Europe Conf. & Exhibition* (Dresden, Germany, Mar. 2012), pp. 117–22.

[15] LIU, R., YANG, C., AND WU, W. Optimizing NAND flash-based SSDs via retention relaxation. *Target 11*, 10 (2012).

[16] LUE, H.-T., DU, P.-Y., CHEN, C.-P., CHEN, W.-C., HSIEH, C.-C., HSIAO, Y.-H., SHIH, Y.-H., AND LU, C.-Y. Radically extending the cycling endurance of flash memory (to >100M cycles) by using built-in thermal annealing to self-heal the stress-induced damage. In *IEEE Int. Electron Devices Meeting* (San Francisco, California, USA, Dec. 2012), pp. 9.1.1–4.

[17] MICHELONI, R., CRIPPA, L., AND MARELLI, A. *Inside NAND Flash Memories*. Springer, 2010.

[18] MOHAN, V., SIDDIQUA, T., GURUMURTHI, S., AND STAN, M. R. How I learned to stop worrying and love flash endurance. In *Proc. USENIX Conf. Hot Topics in Storage and File Systems* (Boston, Massachusetts, USA, June 2010).

[19] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write off-loading: Practical power management for enterprise storage. In *Proc. USENIX Conf. File and Storage Technologies* (San Jose, California, USA, Feb. 2008), pp. 253–67.

[20] PAN, Y., DONG, G., AND ZHANG, T. Error rate-based wear-leveling for NAND flash memory at highly scaled technology nodes. *IEEE Trans. Very Large Scale Integration Systems 21*, 7 (July 2013), 1350–54.

[21] PARK, D., DEBNATH, B., NAM, Y., DU, D. H. C., KIM, Y., AND KIM, Y. HotDataTrap: a sampling-based hot data identification scheme for flash memory. In *ACM Int. Symp. Applied Computing* (Riva del Garda, Italy, Mar. 2012), pp. 1610–17.

[22] PARK, J.-W., PARK, S.-H., WEEMS, C. C., AND KIM, S.-D. A hybrid flash translation layer design for SLC-MLC flash memory based multibank solid state disk. *Microprocessors & Microsystems 35*, 1 (Feb. 2011), 48–59.

[23] SCHWARZ, T., XIN, Q., MILLER, E., LONG, D. D. E., HOSPODOR, A., AND NG, S. Disk scrubbing in large archival storage systems. In *IEEE Int. Symp. Modeling, Analysis, and Simulation of Computer and Telecommunications Systems* (Volendam, Netherlands, Oct. 2004), pp. 409–18.

[24] WANG, C., AND WONG, W.-F. Extending the lifetime of NAND flash memory by salvaging bad blocks. In *Design, Automation & Test in Europe Conf. & Exhibition* (Dresden, Germany, Mar. 2012), pp. 260–63.

[25] WU, M., AND ZWAENEPOEL, W. eNVy: a non-volatile, main memory storage system. In *Sixth Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA, Oct. 1994), pp. 86–97.

[26] ZAMBELLI, C., INDACO, M., FABIANO, M., DI CARLO, S., PRINETTO, P., OLIVO, P., AND BERTOZZI, D. A cross-layer approach for new reliability-performance trade-offs in MLC NAND flash memories. In *Design, Automation & Test in Europe Conf. & Exhibition* (Dresden, Germany, 2012), pp. 881–86.

Lifetime Improvement of NAND Flash-based Storage Systems Using Dynamic Program and Erase Scaling

Jaeyong Jeong*, Sangwook Shane Hahn*, Sungjin Lee†, and Jihong Kim*

*Dept. of CSE, Seoul National University, {jyjeong, shanehahn, jihong}@davinci.snu.ac.kr
†CSAIL, Massachusetts Institute of Technology, chamdoo@csail.mit.edu

Abstract

The cost-per-bit of NAND flash memory has been continuously improved by semiconductor process scaling and multi-leveling technologies (e.g., a 10 nm-node TLC device). However, the decreasing lifetime of NAND flash memory as a side effect of recent advanced technologies is regarded as a main barrier for a wide adoption of NAND flash-based storage systems. In this paper, we propose a new system-level approach, called dynamic program and erase scaling (DPES), for improving the lifetime (particularly, endurance) of NAND flash memory. The DPES approach is based on our key observation that changing the erase voltage as well as the erase time significantly affects the NAND endurance. By slowly erasing a NAND block with a lower erase voltage, we can improve the NAND endurance very effectively. By modifying NAND chips to support multiple write and erase modes with different operation voltages and times, DPES enables a flash software to exploit the new tradeoff relationships between the NAND endurance and erase voltage/speed under dynamic program and erase scaling. We have implemented the first DPES-aware FTL, called autoFTL, which improves the NAND endurance with a negligible degradation in the overall write throughput. Our experimental results using various I/O traces show that autoFTL can improve the maximum number of P/E cycles by 61.2% over an existing DPES-unaware FTL with less than 2.2% decrease in the overall write throughput.

1 Introduction

NAND flash-based storage devices are increasingly popular from mobile embedded systems (e.g., smartphones and smartpads) to large-scale high-performance enterprise servers. Continuing semiconductor process scaling (e.g., 10 nm-node process technology) combined with various recent advances in flash technology (such as a TLC device [1] and a 3D NAND device [2]) is expected to further accelerate an improvement of the cost-per-bit of NAND devices, enabling a wider adoption of NAND flash-based storage systems. However, the poor endurance of NAND flash memory, which deteriorates further as a side effect of recent advanced technologies, is still regarded as a main barrier for sustainable growth in the NAND flash-based storage market. (We represent the NAND endurance by the maximum number of program/erase (P/E) cycles that a flash memory cell can tolerate while preserving data integrity.) Even though the NAND density doubles every two years, the storage lifetime does not increase as much as expected in a recent device technology [3]. For example, the NAND storage lifetime was increased by only 20% from 2009 to 2011 because the maximum number of P/E cycles was decreased by 40% during that period. In particular, in order for NAND flash memory to be widely adopted in high-performance enterprise storage systems, the deteriorating NAND endurance problem should be adequately resolved.

Since the lifetime L_C of a NAND flash-based storage device with the total capacity C is proportional to the maximum number $MAX_{P/E}$ of P/E cycles, and is inversely proportional to the total written data W_{day} per day, L_C (in days) can be expressed as follows (assuming a perfect wear leveling):

$$L_C = \frac{MAX_{P/E} \times C}{W_{day} \times WAF}, \quad (1)$$

where WAF is a write amplification factor which represents the efficiency of an FTL algorithm. Many existing lifetime-enhancing techniques have mainly focused on reducing WAF by increasing the efficiency of an FTL algorithm. For example, by avoiding unnecessary data copies during garbage collection, WAF can be reduced [4]. In order to reduce W_{day}, various architectural/system-level techniques were proposed. For example, data de-duplication [5], data compression [6] and write traffic throttling [7] are such examples. On the other hand, few system/software-level techniques were proposed for actively increasing the max-

imum number $MAX_{P/E}$ of P/E cycles. For example, a recent study [8] suggests $MAX_{P/E}$ can be indirectly improved by a self-recovery property of a NAND cell but no specific technique was proposed yet.

In this paper, we propose a new approach, called dynamic program and erase scaling (DPES), which can significantly improve $MAX_{P/E}$. The key intuition of our approach, which is motivated by a NAND device physics model on the endurance degradation, is that changing the erase voltage as well as the erase time significantly affects the NAND endurance. For example, slowly erasing a NAND block with a lower erase voltage can improve the NAND endurance significantly. By modifying a NAND device to support multiple write and erase modes (which have different voltage/speed and different impacts on the NAND endurance) and allowing a firmware/software module to choose the most appropriate write and erase mode (e.g., depending on a given workload), DPES can significantly increase $MAX_{P/E}$.

The physical mechanism of the endurance degradation is closely related to stress-induced damage in the tunnel oxide of a NAND memory cell [9]. Since the probability of stress-induced damage has an exponential dependence on the stress voltage [10], reducing the stress voltage (particularly, the erase voltage) is an effective way of improving the NAND endurance. Our measurement results with recent 20 nm-node NAND chips show that when the erase voltage is reduced by 14% during P/E cycles, $MAX_{P/E}$ can increase on average by 117%. However, in order to write data to a NAND block erased with the lower erase voltage (which we call a shallowly erased block in the paper), it is necessary to form narrow threshold voltage distributions after program operations. Since shortening the width of a threshold voltage distribution requires a fine-grained control during a program operation, the program time is increased if a lower erase voltage was used for erasing a NAND block.

Furthermore, for a given erase operation, since a nominal erase voltage (e.g., 14 V) tends to damage the cells more than necessary in the beginning period of an erase operation [11], starting with a lower (than the nominal) erase voltage and gradually increasing to the nominal erase voltage can improve the NAND endurance. However, gradually increasing the erase voltage increases the erase time. For example, our measurement results with recent 20 nm-node NAND chips show that when the initial erase voltage of 10 V is used instead of 14 V during P/E cycles, $MAX_{P/E}$ can increase on average by 17%. On the other hand, the erase time is increased by 300%.

Our DPES approach exploits the above two tradeoff relationships between the NAND endurance and erase voltage/speed at the firmware-level (or the software level in general) so that the NAND endurance is improved while the overall write throughput is not affected. For example, since the maximum performance of NAND flash memory is not always needed in real workloads, a DPES-based technique can exploit idle times between consecutive write requests for shortening the width of threshold voltage distributions so that shallowly erased NAND blocks, which were erased by lower erase voltages, can be used for most write requests. Idle times can be also used for slowing down the erase speed. If such idle times can be automatically estimated by a firmware/system software, the DPES-based technique can choose the most appropriate write speed for each write request or select the most suitable erase voltage/speed for each erase operation. By aggressively selecting endurance-enhancing erase modes (i.e., a slow erase with a lower erase voltage) when a large idle time is available, the NAND endurance can be significantly improved because less damaging erase operations are more frequently used.

In this paper, we present a novel NAND endurance model which accurately captures the tradeoff relationship between the NAND endurance and erase voltage/speed under dynamic program and erase scaling. Based on our NAND endurance model, we have implemented the first DPES-aware FTL, called *autoFTL*, which dynamically adjusts write and erase modes in an automatic fashion, thus improving the NAND endurance with a negligible degradation in the overall write throughput. In autoFTL, we also revised key FTL software modules (such as garbage collector and wear-leveler) to make them DPES-aware for maximizing the effect of DPES on the NAND endurance. Since no NAND chip currently allows an FTL firmware to change its program and erase voltages/times dynamically, we evaluated the effectiveness of autoFTL with the *FlashBench* emulation environment [12] using a DPES-enabled NAND simulation model (which supports multiple write and erase modes). Our experimental results using various I/O traces show that autoFTL can improve $MAX_{P/E}$ by 61.2% over an existing DPES-unaware FTL with less than 2.2% decrease in the overall write throughput.

The rest of the paper is organized as follows. Section 2 briefly explains the basics of NAND operations related to our proposed approach. In Section 3, we present the proposed DPES approach in detail. Section 4 describes our DPES-aware autoFTL. Experimental results follow in Section 5, and related work is summarized in Section 6. Finally, Section 7 concludes with a summary and future work.

2 Background

In order to improve the NAND endurance, our proposed DPES approach exploits key reliability and performance parameters of NAND flash memory during run time. In this section, we review the basics of various reliability parameters and their impact on performance and en-

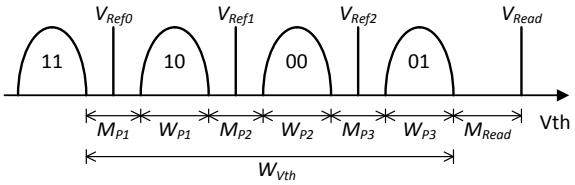


Figure 1: An example of threshold voltage distributions for multi-level NAND flash memory and primary reliability parameters.

durance of NAND cells.

2.1 Threshold Voltage Distributions of NAND Flash Memory

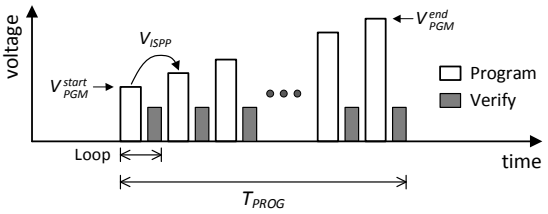
Multi-level NAND flash memory stores 2 bits in a cell using four distinct threshold voltage levels (or states) as shown in Figure 1. Four states are distinguished by different reference voltages, V_{Ref0}, V_{Ref1} and V_{Ref2}. The threshold voltage gap M_{Pi} between two adjacent states and the width W_{Pi} of a threshold voltage distribution are mainly affected by data retention and program time requirements [13, 14], respectively. As a result, the total width W_{Vth} of threshold voltage distributions should be carefully designed to meet all the NAND requirements. In order for flash manufacturers to guarantee the reliability and performance requirements of NAND flash memory throughout its storage lifespan, all the reliability parameters, which are highly inter-related each other, are usually *fixed* during device design times under the worst-case operating conditions of a storage product.

However, if one performance/reliability requirement can be relaxed under specific conditions, it is possible to drastically improve the reliability or performance behavior of the storage product by exploiting tradeoff relationships among various reliability parameters. For example, Liu *et al*. [13] suggested a system-level approach that improves the NAND write performance when most of written data are short-lived (i.e., frequently updated data) by sacrificing M_{Pi}'s which affect the data retention capability[1]. Our proposed DPES technique exploits W_{Pi}'s (which also affect the NAND write performance) so that the NAND endurance can be improved.

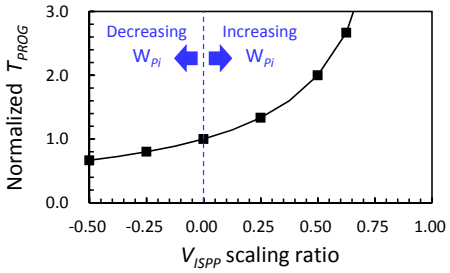
2.2 NAND Program Operations

In order to form a threshold voltage distribution within a desired region, NAND flash memory generally uses the incremental step pulse programming (ISPP) scheme. As shown in Figure 2(a), the ISPP scheme gradually increases the program voltage by the V_{ISPP} step until all the memory cells in a page are located in a desired threshold

[1] Since short-lived data do not need a long data retention time, M_{Pi}'s are maintained loosely so that the NAND write performance can be improved.



(a) A conceptual timing diagram of the ISPP scheme.



(b) Normalized T_{PROG} variations over different V_{ISPP} scaling ratios.

Figure 2: An overview of the incremental step pulse programming (ISPP) scheme for NAND flash memory.

voltage region. While repeating ISPP loops, once NAND cells are verified to have been sufficiently programmed, those cells are excluded from subsequent ISPP loops.

Since the program time is proportional to the number of ISPP loops (which are inversely proportional to V_{ISPP}), the program time T_{PROG} can be expressed as follows:

$$T_{PROG} \propto \frac{V_{PGM}^{end} - V_{PGM}^{start}}{V_{ISPP}}. \quad (2)$$

Figure 2(b) shows normalized T_{PROG} variations over different V_{ISPP} scaling ratios. (When a V_{ISPP} scaling ratio is set to $x\%$, V_{ISPP} is reduced by $x\%$ of the nominal V_{ISPP}.) When a narrow threshold voltage distribution is needed, V_{ISPP} should be reduced for a fine-grained control, thus increasing the program time. Since the width of a threshold voltage distribution is proportional to V_{ISPP} [14], for example, if the nominal V_{ISPP} is 0.5 V and the width of a threshold voltage distribution is reduced by 0.25 V, V_{ISPP} also needs to be reduced by 0.25 V (i.e., a V_{ISPP} scaling ratio is 0.5), thus increasing T_{PROG} by 100%.

3 Dynamic Program and Erase Scaling

The DPES approach is based on our key observation that slowly erasing (i.e., erase time scaling) a NAND block with a lower erase voltage (i.e., erase voltage scaling) significantly improves the NAND endurance. In this section, we explain the effect of erase voltage scaling on improving the NAND endurance and describe the dynamic program scaling method for writing data to a shallowly erased NAND block (i.e., a NAND block erased with

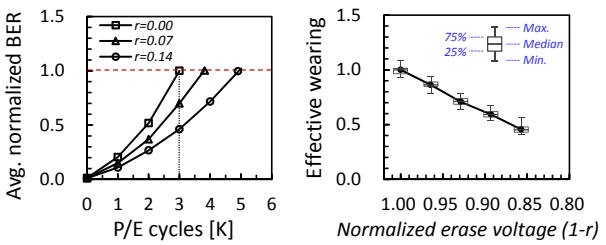
a lower erase voltage). We also present the concept of erase time scaling and its effect on improving the NAND endurance. Finally, we present a novel NAND endurance model which describes the effect of DPES on the NAND endurance based on an empirical measurement study using 20 nm-node NAND chips.

3.1 Erase Voltage Scaling and its Effect on NAND Endurance

The time-to-breakdown T_{BD} of the oxide layer decreases exponentially as the stress voltage increases because the higher stress voltage accelerates the probability of stress-induced damage which degrades the oxide reliability [10]. This phenomenon implies that the NAND endurance can be improved by lowering the stress voltage (e.g., program and erase voltages) during P/E cycles because the reliability of NAND flash memory primarily depends on the oxide reliability [9]. Although the maximum program voltage to complete a program operation is usually larger than the erase voltage, the NAND endurance is mainly degraded during erase operations because the stress time interval of an erase operation is about 100 times longer than that of a program operation. Therefore, if the erase voltage can be lowered, its impact on the NAND endurance improvement can be significant.

In order to verify our observation, we performed NAND cycling tests by changing the erase voltage. In a NAND cycling test, program and erase operations are repeated 3,000 times (which are roughly equivalent to $MAX_{P/E}$ of a recent 20 nm-node NAND device [3]). Our cycling tests for each case are performed with more than 80 blocks which are randomly selected from 5 NAND chips. In our tests, we used the NAND retention BER (i.e., a BER after 10 hours' baking at 125°C) as a measure for quantifying the wearing degree of a NAND chip [9]. (This is a standard NAND retention evaluation procedure specified by JEDEC [15].) Figure 3(a) shows how the retention BER changes, on average, as the number of P/E cycles increases while varying erase voltages. We represent different erase voltages using an voltage scaling ratio r ($0 \leq r \leq 1$). When r is set to x, the erase voltage is reduced by $(x \times 100)$% of the nominal erase voltage. The retention BERs were normalized over the retention BER after 3K P/E cycles when the nominal erase voltage was used. As shown in Figure 3(a), the more the erase voltage is reduced (i.e., the higher r's), the less the retention BERs. For example, when the erase voltage is reduced by 14% of the nominal erase voltage, the normalized retention BER is reduced by 54% after 3K P/E cycles over the nominal erase voltage case.

Since the normalized retention BER reflects the degree of the NAND wearing, higher r's lead to less endurance degradations. Since different erase voltages degrade the NAND endurance by different amounts, we introduce a new endurance metric, called *effective wearing per PE* (in short, *effective wearing*), which represents the effective degree of NAND wearing after a P/E cycle. We represent the effective wearing by a normalized retention BER after 3K P/E cycles[2]. Since the normalized retention BER is reduced by 54% when the erase voltage is reduced by 14%, the effective wearing becomes 0.46. When the nominal erase voltage is used, the effective wearing is 1.



(a) Average BER variations over different P/E cycles under varying erase voltage scaling ratios (r's)

(b) Effective wearing over different erase voltage scaling ratios (r's)

Figure 3: The effect of lowering the erase voltage on the NAND endurance.

As shown in Figure 3(b), the effective wearing decreases near-linearly as r increases. Based on a linear regression model, we can construct a linear equation for the effective wearing over different r's. Using this equation, we can estimate the effective wearing for a different r. After 3K P/E cycles, for example, the total sum of the effective wearing with the nominal erase voltage is 3K. On the other hand, if the erase voltage was set to 14% less than the nominal voltage, the total sum of the effective wearing is only 1.38K because the effective wearing with r of 0.14 is 0.46. As a result, $MAX_{P/E}$ can be increased more than twice as much when the erase voltage is reduced by 14% over the nominal case. In this paper, we will use a NAND endurance model with five different erase voltage modes (as described in Section 3.5).

Since we did not have access to NAND chips from different manufacturers, we could not prove that our test results can be generalized. However, since our tests are based on widely-known device physics which have been investigated by many device engineers and researchers, we are convinced that the consistency of our results would be maintained as long as NAND flash memories use the same physical mechanism (i.e., FN-tunneling) for program and erase operations. We believe that our results will also be effective for future NAND devices as long as

[2] In this paper, we use a linear approximation model which simplifies the wear-out behavior over P/E cycles. Our current linear model can overestimate the effective wearing under low erase voltage scaling ratios while it can underestimate the effective wearing under high erase voltage scaling ratios. We verified that, by the combinations of over-/under-estimations of the effective wearing in our model, the current linear model achieves a reasonable accuracy with an up to 10% overestimation [16] while supporting a simple software implementation.

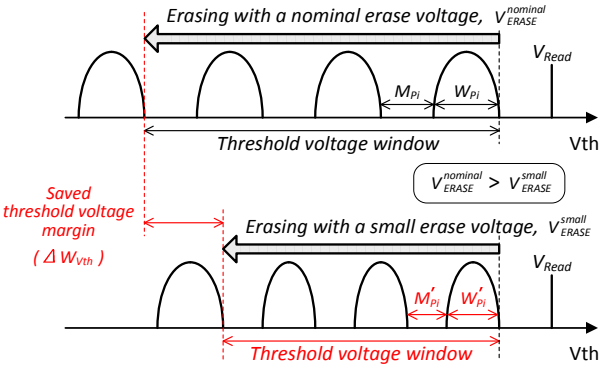


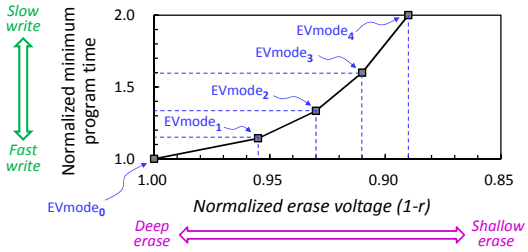
Figure 4: An example of program voltage scaling for writing data to a shallowly erased NAND block.

their operations are based on the FN-tunneling mechanism. It is expected that current 2D NAND devices will gradually be replaced by 3D NAND devices, but the basis of 3D NAND is still the FN-tunneling mechanism.

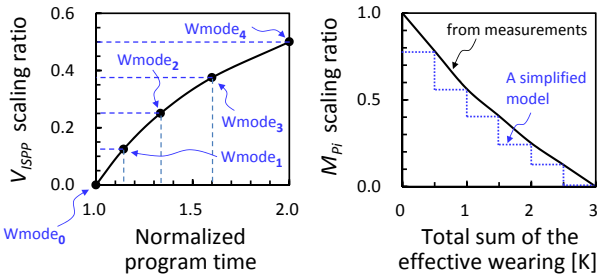
3.2 Dynamic Program Scaling

In order to write data to a shallowly erased NAND block, it is necessary to change program bias conditions dynamically so that narrow threshold voltage distributions can be formed after program operations. If a NAND block was erased with a lower erase voltage, a threshold voltage window for a program operation is reduced by the decrease in the erase voltage because the value of the erase voltage decides how deeply a NAND block is erased. For example, as shown in Figure 4, if a NAND block is shallowly erased with a lower erase voltage V_{ERASE}^{small} (which is lower than the nominal erase voltage $V_{ERASE}^{nominal}$), the width of a threshold voltage window is reduced by a saved threshold voltage margin ΔW_{Vth} (which is proportional to the voltage difference between $V_{ERASE}^{nominal}$ and V_{ERASE}^{small}). Since threshold voltage distributions can be formed only within the given threshold voltage window when a lower erase voltage is used, a fine-grained program control is necessary, thus increasing the program time of a shallowly erased block.

In our proposed DPES technique, we use five different erase voltage modes, $\texttt{EVmode}_0, \cdots, \texttt{EVmode}_4$. $\texttt{EVmode}_0$ uses the highest erase voltage V_0 while $\texttt{EVmode}_4$ uses the lowest erase voltage V_4. After a NAND block is erased, when the erased block is programmed again, there is a strict requirement on the minimum interval length of the program time which depends on the erase voltage mode used for the erased block. (As explained above, this minimum program time requirement is necessary to form threshold voltage distributions within the reduced threshold voltage window.) Figure 5(a) shows these minimum program times for five erase voltage modes. For example, if a NAND block were erased by $\texttt{EVmode}_4$, where the erase voltage is 89% of the nominal erase voltage, the



(a) An example relationship between erase voltages and the normalized minimum program times when the total sum of effective wearing is in the range of 0.0 ∼ 0.5K.



(b) V_{ISPP} scaling ratios (c) M_{Pi} scaling ratios

Figure 5: The relationship between the erase voltage and the minimum program time, and V_{ISPP} scaling and M_{Pi} scaling for dynamic program scaling.

erased block would need at least twice longer program time than the nominal program time. On the other hand, if a NAND block were erased by $\texttt{EVmode}_0$, where the erase voltage is same as the nominal erase voltage, the erased block can be programmed with the same nominal program time.

In order to satisfy the minimum program time requirements of different $\texttt{EVmode}_i$'s, we define five different write modes, $\texttt{Wmode}_0, \cdots, \texttt{Wmode}_4$ where $\texttt{Wmode}_i$ satisfies the minimum program time requirement of the blocks erased by $\texttt{EVmode}_i$. Since the program time of $\texttt{Wmode}_j$ is longer than that of $\texttt{Wmode}_i$ (where $j > i$), $\texttt{Wmode}_k$, $\texttt{Wmode}_{(k+1)}, \cdots, \texttt{Wmode}_4$ can be used when writing to the blocks erased by $\texttt{EVmode}_k$. Figure 5(b) shows how V_{ISPP} should be scaled for each write mode so that the minimum program time requirement can be satisfied. The program time is normalized over the nominal T_{PROG}.

In order to form threshold voltage distributions within a given threshold voltage window, a fine-grained program control is necessary by reducing M_{Pi}'s and W_{Pi}'s. As described in Section 2.2, we can reduce W_{Pi}'s by scaling V_{ISPP} based on the program time requirement. Figure 5(b) shows the tradeoff relationship between the program time and V_{ISPP} scaling ratio based on our NAND characterization study. The program time is normalized over the nominal T_{PROG}. For example, in the case of $\texttt{Wmode}_4$, when the program time is two times longer than the nominal T_{PROG}, V_{ISPP} can be maximally reduced. Dynamic program scaling can be easily integrated into an

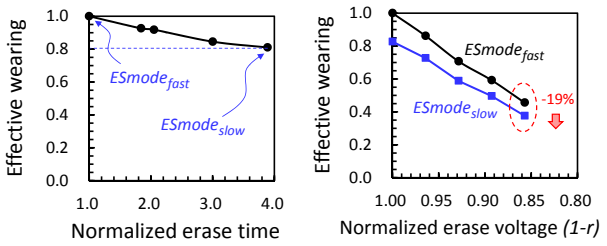
existing NAND controller with a negligible time overhead (e.g., less than 1% of T_{PROG}) and a very small space overhead (e.g., 4 bits per block). On the other hand, in conventional NAND chips, M_{Pi} is kept large enough to preserve the data retention requirement under the worst-case operating condition (e.g., 1-year data retention after 3,000 P/E cycles). However, since the data retention requirement is proportional to the total sum of the effective wearing [9], M_{Pi} can be relaxed by removing an unnecessary data retention capability. Figure 5(c) shows our M_{Pi} scaling model over different total sums of the effective wearing based on our measurement results. In order to reduce the management overhead, we change the M_{Pi} scaling ratio every 0.5-K P/E cycle interval (as shown by the dotted line in Figure 5(c)).

3.3 Erase Time Scaling and its Effect on NAND Endurance

When a NAND block is erased, a high nominal erase voltage (e.g., 14 V) is applied to NAND memory cells. In the beginning period of an erase operation, since NAND memory cells are not yet sufficiently erased, an excessive high voltage (i.e., the nominal erase voltage plus the threshold voltage in a programmed cell) is inevitably applied across the tunnel oxide. For example, if 14 V is required to erase NAND memory cells, when an erase voltage (i.e., 14 V) is applied to two programmed cells whose threshold voltages are 0 V and 4 V, the total erase voltages applied to two memory cells are 14 V and 18 V, respectively [16]. As described in Section 3.1, since the probability of damage is proportional to the erase voltage, the memory cell with a high threshold voltage is damaged more than that with a low threshold voltage, resulting in unnecessarily degrading the memory cell with a high threshold voltage.

In order to minimize unnecessary damage in the beginning period of an erase operation, it is an effective way to start the erase voltage with a sufficiently low voltage (e.g., 10 V) and gradually increase to the nominal erase voltage [11]. For example, if we start with the erase voltage of 10 V, the memory cell whose threshold voltage is 4 V may be partially erased because the erase voltage is 14 V (i.e., 10 V plus 4 V) without excessive damage to the memory cell. As we increase the erase voltage in subsequent ISPE (incremental step pulse erasing [17]) loops, the threshold voltage in the cell is reduced by each ISPE step, thus avoiding unnecessary damage during an erase operation. In general, the lower the starting erase voltage, the less damage to the cells.

However, as an erase operation starts with a lower voltage than the nominal voltage, the erase time increases because more erase loops are necessary for completing the erase operation. Figure 6(a) shows how the effective wearing decreases, on average, as the erase time increases. The longer the erase time (i.e., the lower the starting erase voltage), the less the effective wearing (i.e., the higher NAND endurance.). We represent the fast erase mode by $\texttt{ESmode}_{fast}$ and the slow erase mode by $\texttt{ESmode}_{slow}$. Our measurement results with 20 nm-node NAND chips show that if we increase the erase time by 300% by starting with a lower erase voltage, the effective wearing is reduced, on average, by 19%. As shown in Figure 6(b), the effect of the slow erase mode on improving the NAND endurance can be exploited regardless of the erase voltage scaling ratio r. Since the erase voltage modes are continuously changed depending on the program time requirements, the endurance-enhancing erase mode (i.e., the lowest erase voltage mode) cannot be used under an intensive workload condition. On the other hand, the erase time scaling can be effective even under an intensive workload condition, if slightly longer erase times do not affect the overall write throughput.



(a) Effective wearing variations over different erase times

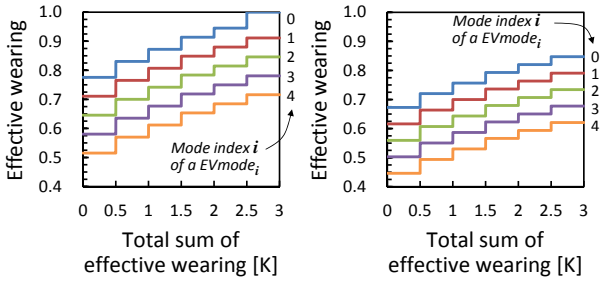
(b) Effective wearing variations over varying erase voltage scaling ratios (r's) under two different erase time settings

Figure 6: The effect of erase time scaling on the NAND endurance.

3.4 Lazy Erase Scheme

As explained in Section 3.2, when a NAND block was erased with $\texttt{EVmode}_i$, a page in the shallowly erased block can be programmed using specific $\texttt{Wmode}_j$'s (where $j \geq i$) only because the requirement of the saved threshold voltage margin cannot be satisfied with a faster write mode $\texttt{Wmode}_k$ $(k < i)$. In order to write data with a faster write mode to the shallowly erased NAND block, the shallowly erased block should be erased further before it is written. We propose a lazy erase scheme which additionally erases the shallowly erased NAND block, when necessary, with a small extra erase time (i.e., 20% of the nominal erase time). Since the effective wearing mainly depends on the maximum erase voltage used, erasing a NAND block by a high erase voltage in a lazy fashion does not incur any extra damage than erasing it with the initially high erase voltage[3]. Since a lazy erase

[3] Although it takes a longer erase time, the total sum of the effective wearing by lazily erasing a shallowly erased block is less than that by erasing with the initially high erase voltage. This can be explained in a



(a) The endurance model for $\texttt{ESmode}_{fast}$.

(b) The endurance model for $\texttt{ESmode}_{slow}$.

Figure 7: The proposed NAND endurance model for DPES-enabled NAND blocks.

cancels an endurance benefit of a shallow erase while introducing a performance penalty, it is important to accurately estimate the write speed of future write requests so that correct erase modes can be selected when erasing NAND blocks, thus avoiding unnecessary lazy erases.

3.5 NAND Endurance Model

Combining erase voltage scaling, program time scaling and erase time scaling, we developed a novel NAND endurance model that can be used with DPES-enabled NAND chips. In order to construct a DPES-enabled NAND endurance model, we calculate saved threshold voltage margins for each combination of write modes (as shown in Figure 5(b)) and M_{Pi} scaling ratios (as shown in Figure 5(c)). Since the effective wearing has a near-linear dependence on the erase voltage and time as shown in Figures 3(b) and 6(b), respectively, the values of the effective wearing for each saved threshold voltage margin can be estimated by a linear equation as described in Section 3.1. All the data in our endurance model are based on measurement results with recent 20 nm-node NAND chips. For example, when the number of P/E cycles is less than 500, and a block is slowly erased before writing with the slowest write mode, a saved threshold voltage margin can be estimated to 1.06 V (which corresponds to the erase voltage scaling ratio r of 0.14 in Figure 6(b)). As a result, we can estimate the value of the effective wearing as 0.45 by a linear regression model for the solid line with squared symbols in Figure 6(b).

Figure 7 shows our proposed NAND endurance model with five erase voltage modes (i.e., $\texttt{EVmode}_0 \sim \texttt{EVmode}_4$) and two erase speed modes (i.e., $\texttt{ESmode}_{slow}$ and $\texttt{ESmode}_{fast}$). $\texttt{EVmode}_0$ (which uses the largest erase voltage) supports the fastest write mode (i.e., $\texttt{Wmode}_0$) with no slowdown in the write speed while $\texttt{EVmode}_4$ (which uses the smallest erase voltage) supports only the slowest write mode (i.e., $\texttt{Wmode}_4$) with the largest wearing gain. Similarly, $\texttt{ESmode}_{fast}$ is the fast erase mode with no additional wearing gain while $\texttt{ESmode}_{slow}$ represents the slow erase mode with the improved wearing gain. Our proposed NAND endurance model takes account of both V_{ISPP} scaling and M_{Pi} scaling described in Figures 5(b) and 5(c).

similar fashion as why the erase time scaling is effective in improving the NAND endurance as discussed in the previous section. The endurance gain from using two different starting erase voltages is higher than the endurance loss from a longer erase time.

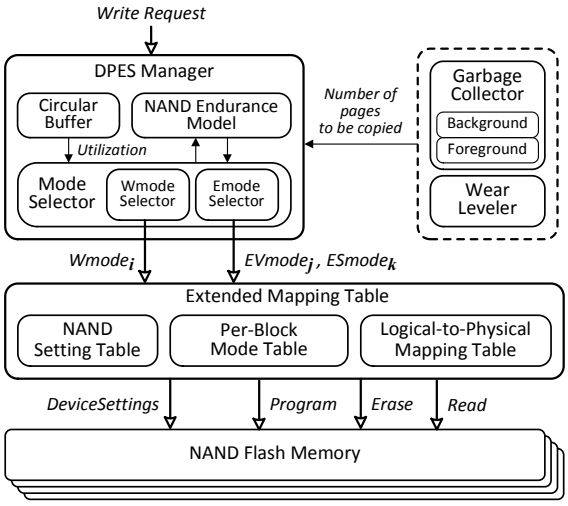


Figure 8: An organizational overview of autoFTL.

4 Design and Implementation of AutoFTL

4.1 Overview

Based on our NAND endurance model presented in Section 3.5, we have implemented autoFTL, the first DPES-aware FTL, which automatically changes write and erase modes depending on write throughput requirements. AutoFTL is based on a page-level mapping FTL with additional modules for DPES support. Figure 8 shows an organizational overview of autoFTL. The DPES manager, which is the core module of autoFTL, selects a write mode $\texttt{Wmode}_i$ for a write request and decides both an appropriate erase voltage mode $\texttt{EVmode}_j$ and erase speed mode $\texttt{ESmode}_k$ for each erase operation. In determining appropriate modes, the mode selector bases its decisions on the estimated write throughput requirement using a circular buffer. AutoFTL maintains per-block mode information and NAND setting information as well as logical-to-physical mapping information in the extended mapping table. The per-block mode table keeps track of the current write mode and the total sum of the effective wearing for each block. The NAND setting table is used to choose appropriate device settings for the selected write and erase modes, which are sent to NAND chips via a new interface *DeviceSettings* between autoFTL and NAND chips. AutoFTL also extends both the garbage collector and wear leveler to be DPES-aware.

Table 1: The write-mode selection rules used by the DPES manager.

Buffer utilization u	Write mode
$u > 80\%$	$\texttt{Wmode}_0$
$60\% < u \leq 80\%$	$\texttt{Wmode}_1$
$40\% < u \leq 60\%$	$\texttt{Wmode}_2$
$20\% < u \leq 40\%$	$\texttt{Wmode}_3$
$u \leq 20\%$	$\texttt{Wmode}_4$

As semiconductor technologies reach their physical limitations, it is necessary to use cross-layer optimization between system software and NAND devices. As a result, some of internal device interfaces are gradually opened to public in the form of additional 'user interface'. For example, in order to track bit errors caused by data retention, a new 'device setting interface' which adjusts the internal reference voltages for read operations is recently opened to public [18, 19]. There are already many set and get functions for modifying or monitoring NAND internal configurations in the up-to-date NAND specifications such as the toggle mode interface and ONFI. For the measurements presented here, we were fortunately able to work in conjunction with a flash manufacturer to adjust erase voltage as we wanted.

4.2 Write Mode Selection

In selecting a write mode for a write request, the Wmode selector of the DPES manager exploits idle times between consecutive write requests so that autoFTL can increase $MAX_{P/E}$ without incurring additional decrease in the overall write throughput. In autoFTL, the Wmode selector uses a simple circular buffer for estimating the maximum available program time (i.e., the minimum required write speed) for a given write request. Table 1 summarizes the write-mode selection rules used by the Wmode selector depending on the utilization of a circular buffer. The circular buffer queues incoming write requests before they are written, and the Wmode selector adaptively decides a write mode for each write request. The current version of the Wmode selector, which is rather conservative, chooses the write mode, $\texttt{Wmode}_i$, depending on the buffer utilization u. The buffer utilization u represents how much of the circular buffer is filled by outstanding write requests. For example, if the utilization is lower than 20%, the write request in the head of the circular buffer is programmed to a NAND chip with $\texttt{Wmode}_4$.

4.3 Extended Mapping Table

Since erase operations are performed at the NAND block level, the per-block mode table maintains five linked lists of blocks which were erased using the same erase voltage mode. When the DPES manager decides a write mode for a write request, the corresponding linked list is consulted to locate a destination block for the write request. Also, the DPES manager informs a NAND chip how to configure appropriate device settings (e.g., ISPP/ISPE voltages, the erase voltage, and reference voltages for read/verify operations) for the current write mode using the per-block mode table. Once NAND chips are set to a certain mode, an additional setting is not necessary as long as the write and the erase modes are maintained. For a read request, since different write modes require different reference voltages for read operations, the per-block mode table keeps track of the current write mode for each block so that a NAND chip changes its read references before serving a read request.

4.4 Erase Voltage Mode Selection

Since the erase voltage has a significant impact on the NAND endurance as described in Section 3.1, selecting a right erase voltage is the most important step in improving the NAND endurance using the DPES technique. As explained in Section 4.2, since autoFTL decides a write mode of a given write request based on the utilization of the circular buffer of incoming write requests, when deciding the erase voltage mode of a victim block, autoFTL takes into account of the future utilization of the circular buffer. If autoFTL could accurately predict the future utilization of the circular buffer and erase the victim block with the erase voltage that can support the future write mode, the NAND endurance can be improved without a lazy erase operation. In the current version, we use the average buffer utilization of 10^5 past write requests for predicting the future utilization of the circular buffer. In order to reduce the management overhead, we divide 10^5 past write requests into 100 subgroups where each subgroup consists of 1000 write requests. For each subgroup, we compute the average utilization of 1000 write requests in the subgroup, and use the average of 100 subgroup's utilizations to calculate the estimate of the future utilization of the buffer.

When a foreground garbage collection is invoked, since the write speed of a near-future write request is already chosen based on the current buffer utilization, the victim block can be erased with the corresponding erase voltage mode. On the other hand, when a background garbage collection is invoked, it is difficult to use the current buffer utilization because the background garbage collector is activated when there are no more write requests waiting in the buffer. For this case, we use the estimated average buffer utilization of the circular buffer to predict the buffer utilization when the next phase of write requests (after the background garbage collection) fills in the circular buffer.

4.5 Erase Speed Mode Selection

In selecting an erase speed mode for a block erase operation, the DPES manager selects an erase speed mode which does not affect the write throughput. An erase speed mode for erasing a NAND block is determined by estimating the effect of a block erase time on the buffer utilization. Since write requests in the circular buffer cannot be programmed while erasing a NAND block, the buffer utilization is effectively increased by the block erase time. The effective buffer utilization u' considering the effect of the block erase time can be expressed as follows:

$$u' = u + \Delta u^{erase}, \quad (3)$$

where u is the current buffer utilization and Δu^{erase} is the increment in the buffer utilization by the block erase time. In order to estimate the effect of a block erase operation on the buffer utilization, we convert the block erase time to a multiple M of the program time of the current write mode. Δu^{erase} corresponds to the increment in the buffer utilization for these M pages. For selecting an erase speed mode of a NAND block, the mode selector checks if $\texttt{ESmode}_{slow}$ can be used. If erasing with $\texttt{ESmode}_{slow}$ does not increase u' larger than 100% (i.e., no buffer overflow), $\texttt{ESmode}_{slow}$ is selected. Otherwise, the fast erase mode $\texttt{ESmode}_{fast}$ is selected. On the other hand, when the background garbage collection is invoked, $\texttt{ESmode}_{slow}$ is always selected in erasing a victim block. Since the background garbage collection is invoked when an idle time between consecutive write requests is sufficiently long, the overall write throughput is not affected even with $\texttt{ESmode}_{slow}$.

4.6 DPES-Aware Garbage Collection

When the garbage collector is invoked, the most appropriate write mode for copying valid data to a free block is determined by using the same write-mode selection rules summarized in Table 1 with a slight modification to computing the buffer utilization u. Since the write requests in the circular buffer cannot be programmed while copying valid pages to a free block by the garbage collector, the buffer utilization is effectively increased by the number of valid pages in a victim block. By using the information from the garbage collector, the mode selector recalculates the effective buffer utilization u^* as follows:

$$u^* = u + \Delta u^{copy}, \quad (4)$$

where u is the current buffer utilization and Δu^{copy} is the increment in the buffer utilization taking the number of valid pages to be copied into account. The mode selector decides the most appropriate write mode based on the write-mode selection rules with u^* instead of u. After copying all the valid pages to a free block, a victim block is erased by the erase voltage mode (selected by the rules described in Section 4.4) with the erase speed (chosen by the rules described in Section 4.5). For example, as shown in the case 1 of Table 2, if garbage collection is invoked when u is 70%, and the number of valid pages to be copied is 30 (i.e., $\Delta u^{copy} = 30/200 = 15\%$), $\texttt{Wmode}_0$ is selected because u^* is 85% ($= 70\% + 15\%$), and $\texttt{ESmode}_{slow}$ is selected because erasing with $\texttt{ESmode}_{slow}$ does not overflow the circular buffer. (We assume that Δu^{erase} for $\texttt{ESmode}_{slow}$ and Δu^{erase} for $\texttt{ESmode}_{fast}$ are 8% and 2%, respectively.) On the other hand, as shown in the case 2 of Table 2, when the number of valid pages to be copied is 50 (i.e., $\Delta u^{copy} = 50/200 = 25\%$), $\texttt{ESmode}_{slow}$ cannot be selected because u' becomes larger than 100%. As shown in the case 1, $\texttt{ESmode}_{slow}$ can still be used even when the buffer utilization is higher than 80%. When the buffer utilization is higher than 80% (i.e., an intensive write workload condition), the erase voltage scaling is not effective because the highest erase voltage is selected. On the other hand, even when the buffer utilization is above 90%, the erase speed scaling can be still useful.

Table 2: Examples of selecting write and erase modes in the garbage collector assuming that the circular buffer has 200 pages and the current buffer utilization u is 70%.

(Case 1) The number of valid pages in a victim block is 30.

u^{copy}	u^*	Δu^{erase}		u'	Selected modes
15%	85%	Slow	8%	93%	$\texttt{EVmode}_0$ & $\texttt{ESmode}_{slow}$
		Fast	2%	87%	$\texttt{Wmode}_0$

(Case 2) The number of valid pages in a victim block is 50.

u^{copy}	u^*	Δu^{erase}		u'	Selected modes
25%	95%	Slow	8%	103%	$\texttt{EVmode}_0$ & $\texttt{ESmode}_{fast}$
		Fast	2%	97%	$\texttt{Wmode}_0$

4.7 DPES-Aware Wear Leveling

Since different erase voltage/time affects the NAND endurance differently as described in Section 3.1, the reliability metric (based on the number of P/E cycles) of the existing wear leveling algorithm [20] is no longer valid in a DPES-enabled NAND flash chip. In autoFTL, the DPES-aware wear leveler uses the total sum of the effective wearing instead of the number of P/E cycles as a reliability metric, and tries to evenly distribute the total sum of the effective wearing among NAND blocks.

5 Experimental Results

5.1 Experimental Settings

In order to evaluate the effectiveness of the proposed autoFTL, we used an extended version of a unified development environment, called *FlashBench* [12], for NAND flash-based storage devices. Since the efficiency of our DPES is tightly related to the temporal characteristics of write requests, we extended the existing FlashBench to be timing-accurate. Our extended FlashBench emulates the key operations of NAND flash memory in a timing-accurate fashion using high-resolution timers (or hrtimers) (which are available in a recent Linux kernel [21]). Our validation results on an 8-core Linux server system show that the extended FlashBench is very accurate. For example, variations on the program time and erase time of our DRAM-based NAND emulation models are less than 0.8% of T_{PROG} and 0.3% of T_{ERASE}, respectively.

For our evaluation, we modified a NAND flash model in FlashBench to support DPES-enabled NAND flash chips with five write modes, five erase voltage modes, and two erase speed modes as shown in Figure 7. Each NAND flash chip employed 128 blocks which were composed of 128 8-KB pages. The maximum number of P/E cycles was set to 3,000. The nominal page program time (i.e., T_{PROG}) and the nominal block erase time (i.e., T_{ERASE}) were set to 1.3 *m*s and 5.0 *m*s, respectively.

We evaluated the proposed autoFTL in two different environments, mobile and enterprise environments. Since the organizations of mobile storage systems and enterprise storage systems are quite different, we used two FlashBench configurations for different environments as summarized in Table 3. For a mobile environment, FlashBench was configured to have two channels, and each channel has a single NAND chip. Since mobile systems are generally resource-limited, the size of a circular buffer for a mobile environment was set to 80 KB only (i.e., equivalently 10 8-KB pages). For an enterprise environment, FlashBench was configured to have eight channels, each of which was composed of four NAND chips. Since enterprise systems can utilize more resources, the size of a circular buffer was set to 32 MB (which is a typical size of data buffer in HDD) for enterprise environments.

We carried out our evaluations with two different techniques: baseline and autoFTL. Baseline is an existing DPES-unaware FTL that always uses the highest erase voltage mode and the fast erase mode for erasing NAND blocks, and the fastest write mode for writing data to NAND blocks. AutoFTL is the proposed DPES-aware FTL which decides the erase voltage and the erase time depending on the characteristic of a workload and fully utilizes DPES-aware techniques, described in Sections 3 and 4, so it can maximally exploit the benefits of dynamic program and erase scaling.

Table 3: Summary of two FlashBench configurations.

Environments	Channels	Chips	Buffer
Mobile	2	2	80 KB
Enterprise	8	32	32 MB

Our evaluations were conducted with various I/O traces from mobile and enterprise environments. (For more details, please see Section 5.2). In order to replay I/O traces on top of the extended FlashBench, we developed a trace replayer. The trace replayer fetches I/O commands from I/O traces and then issues them to the extended FlashBench according to their inter-arrival times to a storage device. After running traces, we measured the maximum number of P/E cycles, $MAX_{P/E}$, which was actually conducted until flash memory became unreliable. We then compared it with that of baseline. The overall write throughput is an important metric that shows the side-effect of autoFTL on storage performance. For this reason, we also measured the overall write throughput while running each I/O trace.

5.2 Benchmarks

We used 8 different I/O traces collected from Android-based smartphones and real-world enterprise servers. The `m_down` trace was recorded while downloading a system installation file (whose size is about 700 MB) using a mobile web-browser through 3G network. The `m_p2p1` trace included I/O activities when downloading multimedia files using a mobile P2P application from a lot of rich seeders. Six enterprise traces, `hm_0`, `proj_0`, `prxy_0`, `src1_2`, `stg_0`, and `web_0`, were from the MS-Cambridge benchmarks [22]. However, since enterprise traces were collected from old HDD-based server systems, their write throughputs were too low to evaluate the performance of modern NAND flash-based storage systems. In order to partially compensate for low write throughput of old HDD-based storage traces, we accelerated all the enterprise traces by 100 times so that the peak throughput of the most intensive trace (i.e., `src1_2`) can fully consume the maximum write throughput of our NAND configuration. (In our evaluations, therefore, all the enterprise traces are 100x-accelerated versions of the original traces.)

Since recent enterprise SSDs utilize lots of inter-chip parallelism (multiple channels) and intra-chip parallelism (multiple planes), peak throughput is significantly higher than that of conventional HDDs. We tried to find appropriate enterprise traces which satisfied our requirements to (1) have public confidence; (2) can fully consume the maximum throughput of our NAND configura-

Table 4: Normalized inter-arrival times of write requests for 8 traces used for evaluations.

Trace	Distributions of normalized inter-arrival times t over $T_{PROG}^{effective}$ [%]		
	$t \leq 1$	$1 < t \leq 2$	$t > 2$
proj_0	40.6%	47.0%	12.4%
src1_2	41.0%	55.6%	3.4%
hm_0	14.2%	72.1%	13.7%
prxy_0	8.9%	34.6%	56.5%
stg_0	7.1%	81.5%	11.4%
web_0	5.4%	36.7%	56.9%
m_down	45.9%	0.0%	54.1%
m_p2p1	49.5%	0.0%	50.5%

tion; (3) reflect real user behaviors in enterprise environments; (4) are extracted from under SSD-based storage systems. To the best of our knowledge, we could not find any workload which met all of the requirements at the same time. In particular, there are few enterprise SSD workloads which are opened to public.

Table 4 summarizes the distributions of inter-arrival times of our I/O traces. Inter-arrival times were normalized over $T_{PROG}^{effective}$ which reflects parallel NAND operations supported by multiple channels and multiple chips per channel in the extended FlashBench. For example, for an enterprise environment, since up to 32 chips can serve write requests simultaneously, $T_{PROG}^{effective}$ is about 40 *u*s (i.e., 1300 *u*s of T_{PROG} is divided by 32 chips.). On the other hand, for a mobile environment, since there are only 2 chips can serve write requests at the same time, $T_{PROG}^{effective}$ is 650 *u*s. Although the mobile traces collected from Android smartphones (i.e., m_down [23] and m_p2p1) exhibit very long inter-arrival times, normalized inter-arrival times over $T_{PROG}^{effective}$ are not much different from the enterprise traces, except that the mobile traces show distinct bimodal distributions which no write requests in $1 < t \leq 2$.

5.3 Endurance Gain Analysis

In order to understand how much $MAX_{P/E}$ is improved by DPES, each trace was repeated until the total sum of the effective wearing reached 3K. Measured $MAX_{P/E}$ values were normalized over that of baseline. Figure 9 shows normalized $MAX_{P/E}$ ratios for eight traces with two different techniques. Overall, the improvement on $MAX_{P/E}$ is proportional to inter-arrival times as summarized in Table 4; the longer inter-arrival times are, the more likely slow write modes are selected.

AutoFTL improves $MAX_{P/E}$ by 69%, on average, over baseline for the enterprise traces. For proj_0 and src1_2 traces, improvements on $MAX_{P/E}$ are less than 50% because inter-arrival times of more than 40% of write requests are shorter than $T_{PROG}^{effective}$ so that it is difficult to use the lowest erase voltage mode. For the other enterprise traces, $MAX_{P/E}$ is improved by 79%, on average, over baseline.

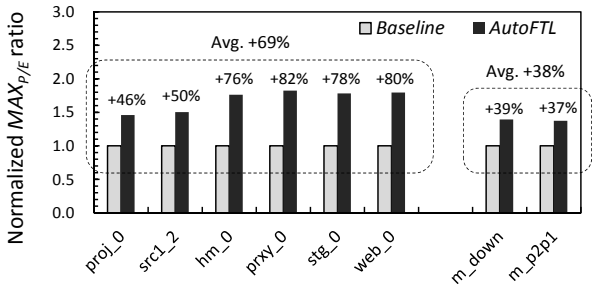


Figure 9: Comparisons of normalized $MAX_{P/E}$ ratios for eight traces.

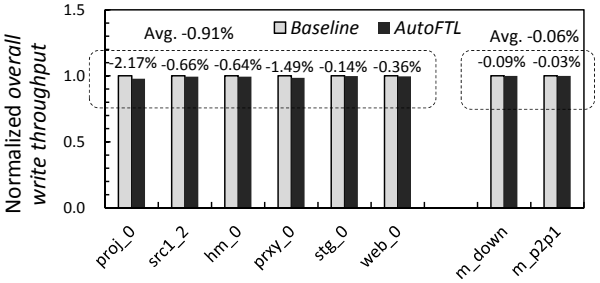


Figure 10: Comparisons of normalized overall write throughputs for eight traces.

On the other hand, for the mobile traces, AutoFTL improves $MAX_{P/E}$ by only 38%, on average, over baseline. Although more than 50% of write requests have inter-arrival times twice longer than $T_{PROG}^{effective}$, autoFTL could not improve $MAX_{P/E}$ as much as expected. This is because the size of the circular buffer is too small for buffering the increase in the buffer utilization caused by the garbage collection. For example, when a NAND block is erased by the fast speed erase mode, the buffer utilization is increased by 40% for the mobile environment while the effect of the fast erase mode on the buffer utilization is less than 0.1% for the enterprise environment. Moreover, by the same reason, the slow erase speed mode cannot be used in the mobile environment.

5.4 Overall Write Throughput Analysis

Although autoFTL uses slow write modes frequently, the decrease in the overall write throughput over baseline is less than 2.2% as shown in Figure 10. For proj_0 trace, the overall write throughput is decreased by 2.2%. This is because, in proj_0 trace, the circular buffer may become full by highly clustered write requests. When the circular buffer becomes full, if the foreground garbage collection should be invoked, the write response time of NAND chips can be directly affected. Although inter-arrival times in prxy_0 trace are relatively long over other enterprise traces, the overall write throughput is

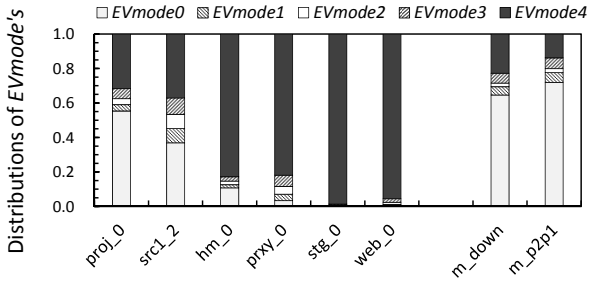


Figure 11: Distributions of EVmode's used.

degraded more than the other enterprise traces. This is because almost all the write requests exhibit inter-arrival times shorter than 10 *m*s so that the background garbage collection is not invoked at all[4]. As a result, the foreground garbage collection is more frequently invoked, thus increasing the write response time.

We also evaluated if there is an extra delay from a host in sending a write request to the circular buffer because of DPES. Although autoFTL introduced a few extra queueing delay for the host, the increase in the average queueing delay per request was negligible compared to $T_{PROG}^{effective}$. For example, for src1_2 trace, 0.4% of the total programmed pages were delayed, and the average queueing delay per request was 2.6 *u*s. For stg_0 trace, less than 0.1% of the total programmed pages were delayed, and the average queueing delay per request was 0.1 *u*s.

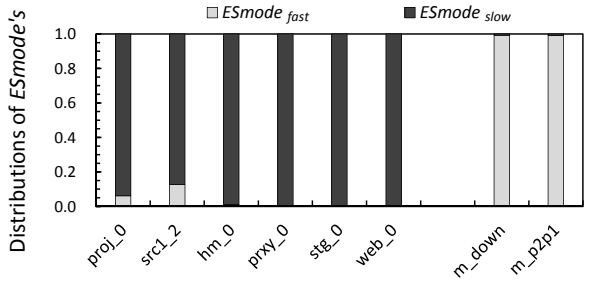
5.5 Detailed Analysis

We performed a detailed analysis on the relationship between the erase voltage/speed modes and the improvement of $MAX_{P/E}$. Figure 11 presents distributions of EVmode's used for eight I/O traces. Distributions of EVmode's exactly correspond to the improvements of $MAX_{P/E}$ as shown in Figure 9; the more frequently a low erase voltage mode is used, the higher the endurance gain is. In our evaluations for eight I/O traces, lazy erases are rarely used for all the traces.

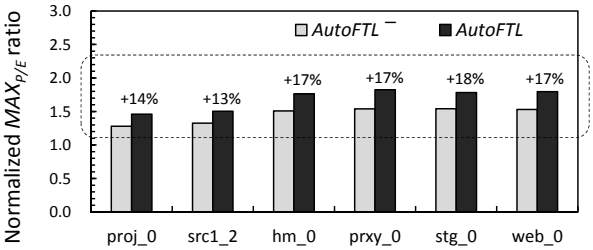
Figure 12(a) shows distributions of ESmode's for eight I/O traces. Since the slow erase mode is selected by using the effective buffer utilization, there are little chances for selecting the slow erase mode for the mobile traces because the size of the circular buffer is only 80 KB. On the other hand, for the enterprise environment, there are more opportunities for selecting the slow erase mode. Even for the traces with short inter-arrival times such as proj_0 and src1_2, only 5%∼10% of block erases used the fast erase mode.

We also evaluated the effect of the slow erase mode on the improvement of $MAX_{P/E}$. For this for evaluation,

[4]In our autoFTL setting, the background garbage collection is invoked when a idle time between two consecutive requests is longer than 300 *m*s.



(a) Distributions of ESmode's used.



(b) The effect of $\texttt{ESmode}_{slow}$ on improving $MAX_{P/E}$.

Figure 12: Distributions of ESmode's used and the effect of ESmode's on $MAX_{P/E}$.

we modified our autoFTL so that $\texttt{ESmode}_{fast}$ is always used when NAND blocks are erased. (We represent this technique by autoFTL$^-$.) As shown in Figure 12(b), the slow erase mode can improve the NAND endurance gain up to 18%. Although the slow erase mode can increase the buffer utilization, its effect on the write throughput was almost negligible.

6 Related Work

As the endurance of recent high-density NAND flash memory is continuously reduced, several system-level techniques which exploit the physical characteristics of NAND flash memory have been proposed for improving the endurance and lifetime of flash-based storage systems [8, 7, 24, 25].

Mohan *et al*. investigated the effect of the damage recovery on the SSD lifetime for enterprise servers [8]. They showed that the overall endurance of NAND flash memory can be improved with its recovery nature. Our DPES technique does not consider the self-recovery effect, but it can be easily extended to exploit the physical characteristic of the self-recovery of flash memory cells.

Lee *et al*. proposed a novel lifetime management technique that guarantees the lifetime of storage devices by intentionally throttling write performance [7]. They also exploited the self-recovery effect of NAND devices, so as to lessen the performance penalty caused by write throttling. Unlike Lee's work (which sacrifices write performance for guaranteeing the storage lifetime), our DPES technique improves the lifetime of NAND devices without degrading the performance of NAND-based stor-

age systems.

Wu *et al*. presented a novel endurance enhancement technique that boosts recovery speed by heating a flash chip under high temperature [24]. By leveraging the temperature-accelerated recovery, it improved the endurance of SSDs up to five times. The major drawback of this approach is that it requires extra energy consumption to heat flash chips and lowers the reliability of a storage device. Our DPES technique improves the endurance of NAND devices by lowering the erase voltage and slowing down the erase speed without any serious side effect.

Jeong *et al*. proposed an earlier version of the DPES idea and demonstrated that DPES can improve the NAND endurance significantly without sacrificing the overall write throughput [25]. Unlike their work, however, our work treats the DPES approach in a more complete fashion, extensively extending the DPES approach in several dimensions such as the erase speed scaling, shallow erasing and lazy erase scheme. Furthermore, more realistic and detailed evaluations using the timing-accurate emulator are presented in this paper.

7 Conclusions

We have presented a new system-level approach for improving the lifetime of flash-based storage systems using dynamic program and erase scaling (DPES). Our DPES approach actively exploits the tradeoff relationship between the NAND endurance and the erase voltage/speed so that directly improves the NAND endurance with a minimal decrease in the write performance. Based on our novel NAND endurance model and the newly defined interface for changing the NAND behavior, we have implemented autoFTL, which changes the erase voltage and speed in an automatic fashion. Moreover, by making the key FTL modules (such as garbage collection and wear leveling) DPES-aware, autoFTL can significantly improve the NAND endurance. Our experimental results show that autoFTL can improve the maximum number of P/E cycles by 69% for enterprise traces and 38% for mobile traces, on average, over an existing DPES-unaware FTL.

The current version of autoFTL can be further improved in several ways. For example, we believe that the current mode selection rules are rather too conservative without adequately reflecting the varying characteristics of I/O workload. As an immediate future task, we plan to develop more *adaptive* mode selection rules that may adaptively adjust the buffer utilization boundaries for selecting write modes.

Acknowledgements

We would like to thank Erik Riedel, our shepherd, and anonymous referees for valuable comments that greatly improved our paper. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT and Future Planning (MSIP) (NRF-2013R1A2A2A01068260). This research was also supported by Next-Generation Information Computing Development Program through NRF funded by MSIP (No. 2010-0020724). The ICT at Seoul National University and IDEC provided research facilities for this study.

References

[1] S.-H. Shin *et al*., "A New 3-bit Programming Algorithm Using SLC-to-TLC Migration for 8 MB/s High Performance TLC NAND Flash Memory," in *Proc. IEEE Symp. VLSI Circuits*, 2012.

[2] J. Choi *et al*., "3D Approaches for Non-volatile Memory," in *Proc. IEEE Symp. VLSI Technology*, 2011.

[3] A. A. Chien *et al*., "Moore's Law: The First Ending and A New Beginning," *Tech. Report, Dept. of Computer Science, the Univ. of Chicago*, TR-2012-06.

[4] J.-W. Hsieh *et al*., "Efficient Identification of Hot Data for Flash Memory Storage Systems," *ACM Trans. Storage*, vol. 2, no. 1, pp. 22-40, 2006.

[5] F. Chen *et al*., "CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory Based Solid State Drives," in *Proc. USENIX Conf. File and Storage Tech*., 2011.

[6] S. Lee *et al*., "Improving Performance and Lifetime of Solid-State Drives Using Hardware-Accelerated Compression," *IEEE Trans. Consum. Electron*., vol. 57, no. 4, pp. 1732-1739, 2011.

[7] S. Lee *et al*., "Lifetime Management of Flash-Based SSDs Using Recovery-Aware Dynamic Throttling," in *Proc. USENIX Conf. File and Storage Tech*., 2012.

[8] V. Mohan *et al*., "How I Learned to Stop Worrying and Love Flash Endurance," in *Proc. USENIX Workshop Hot Topics in Storage and File Systems*, 2010.

[9] N. Mielke *et al*., "Bit Error Rate in NAND Flash Memories," in *Proc. IEEE Int. Reliability Physics Symp*., 2008.

[10] K. F. Schuegraf *et al.,* "Effects of Temperature and Defects on Breakdown Lifetime of Thin SiO_2 at Very Low Voltages," *IEEE Trans. Electron Devices*, vol. 41, no. 7, pp. 1227-1232, 1994.

[11] S. Cho, "Improving NAND Flash Memory Reliability with SSD Controllers," in *Proc. Flash Memory Summit*, 2013.

[12] S. Lee *et al.,* "FlashBench: A Workbench for a Rapid Development of Flash-Based Storage Devices," in *Proc. IEEE Int. Symp. Rapid System Prototyping*, 2012.

[13] R.-S. Liu *et al.,* "Optimizing NAND Flash-Based SSDs via Retention Relaxation," in *Proc. USENIX Conf. File and Storage Tech.*, 2012.

[14] K.-D. Suh *et al.,* "A 3.3 V 32 Mb NAND Flash Memory with Incremental Step Pulse Programming Scheme," *IEEE J. Solid-State Circuits*, vol. 30, no. 11, pp. 1149-1156, 1995.

[15] JEDEC Standard, "Stress-Test-Driven Qualification of Integrated Circuits," JESD47H.01, 2011.

[16] J. Jeong and J. Kim, "Dynamic Program and Erase Scaling in NAND Flash-based Storage Systems," *Tech. Report, Seoul National Univ.*, http://cares.snu.ac.kr/download/TR-CARES-01-14, 2014.

[17] D.-W. Lee *et al.,* "The Operation Algorithm for Improving the Reliability of TLC (Triple Level Cell) NAND Flash Characteristics," in *Proc. IEEE Int. Memory Workshop*, 2011.

[18] J. Yang, "High-Efficiency SSD for Reliable Data Storage Systems," in *Proc. Flash Memory Summit*, 2011.

[19] R. Frickey, "Data Integrity on 20 nm NAND SSDs," in *Proc. Flash Memory Summit*, 2012.

[20] L.-P. Chang, "On Efficient Wear Leveling for Large-Scale Flash-Memory Storage Systems," in *Proc. ACM Symp. Applied Computing*, 2007.

[21] IBM "Kernel APIs, Part 3: Timers and Lists in the 2.6 Kernel," http://www.ibm.com/developerworks/library/l-timers-list/.

[22] D. Narayanan *et al.,* "Write Off-Loading: Practical Power Management for Enterprise Storage," in *Proc. USENIX Conf. File and Storage Tech.*, 2008.

[23] http://www.ubuntu.com/download

[24] Q. Wu *et al.,* "Exploiting Heat-Accelerated Flash Memory Wear-Out Recovery to Enable Self-Healing SSDs," in *Proc. USENIX Workshop Hot Topics in Storage and File Systems*, 2011.

[25] J. Jeong *et al.,* "Improving NAND Endurance by Dynamic Program and Erase scaling," in *Proc. USENIX Workshop Hot Topics in Storage and File Systems*, 2013.

ReconFS: A Reconstructable File System on Flash Storage

Youyou Lu Jiwu Shu* Wei Wang
Department of Computer Science and Technology, Tsinghua University
Tsinghua National Laboratory for Information Science and Technology
* *Corresponding author: shujw@tsinghua.edu.cn*
{*luyy09, wangwei11*}*@mails.tsinghua.edu.cn*

Abstract

Hierarchical namespaces (directory trees) in file systems are effective in indexing file system data. However, the update patterns of namespace metadata, such as intensive writeback and scattered small updates, exaggerate the writes to flash storage dramatically, which hurts both performance and endurance (i.e., limited program/erase cycles of flash memory) of the storage system.

In this paper, we propose a reconstructable file system, ReconFS, to reduce namespace metadata writeback size while providing hierarchical namespace access. ReconFS decouples the volatile and persistent directory tree maintenance. Hierarchical namespace access is emulated with the volatile directory tree, and the consistency and persistence of the persistent directory tree are provided using two mechanisms in case of system failures. First, consistency is ensured by embedding an inverted index in each page, eliminating the writes of the pointers (indexing for directory tree). Second, persistence is guaranteed by compacting and logging the scattered small updates to the metadata persistence log, so as to reduce write size. The inverted indices and logs are used respectively to reconstruct the structure and the content of the directory tree on reconstruction. Experiments show that ReconFS provides up to 46.3% performance improvement and 27.1% write reduction compared to ext2, a file system with low metadata overhead.

1 Introduction

In recent years, flash memory is gaining popularity in storage systems for its high performance, low power consumption and small size [11, 12, 13, 19, 23, 28]. However, flash memory has limited program/erase (P/E) cycles, and the reliability is weakened as P/E cycles approach the limit, which is known as the endurance problem [10, 14, 17, 23]. The recent trend of denser flash memory, which increases storage capacity by multiple-level cell (MLC) or triple-level cell (TLC) technologies, makes the endurance problem even worse [17].

File system design evolves slowly in the past few decades, yet it has a marked impact on I/O behaviors of the storage subsystems. Recent studies have proposed to revisit the namespace structure of file systems, e.g., flexible indexing for search-friendly file systems [33] and table structured metadata management for better metadata access performance [31]. Meanwhile, leveraging the internal storage management of flash translation layer (FTL) of solid state drives (SSDs) to improve storage management efficiency has also been discussed [19, 23, 25, 37]. But namespace management also impacts flash-based storage performance and endurance, especially when considering metadata-intensive workloads. This however has not been well researched.

Namespace metadata are intensively written back to persistent storage due to system consistency or persistence guarantees [18, 20]. Since the no-overwrite property of flash memory requires writes to be updated in free pages, frequent writeback introduces a large *dynamic update size* (i.e., the total write size of free pages that are used). Even worse, a single file system operation may scatter updates to different metadata pages (e.g., the create operation writes both the inode and the directory entry), and the average update size to each metadata page is far less than one page size (e.g., an inode in ext2 has the size of 128 bytes). A whole page needs to be written even though only a small part in the page is updated. Endurance, as well as performance, of flash storage systems is affected by namespace metadata accesses due to frequent and scattered small write patterns.

To address these problems, we propose a reconstructable file system, ReconFS, which provides a volatile hierarchical namespace and relaxes the write-back requirements. ReconFS decouples the maintenance of the volatile and persistent directory trees. Metadata

pages are written back to their home locations only when they are evicted or checkpointed (i.e., the operation to update the persistent directory tree the same as the volatile directory tree) from main memory. Consistency and persistence of the persistent directory tree are guaranteed using two new mechanisms. First, we use *embedded connectivity* mechanism to embed an inverted index in each page and track the unindexed pages. Since the namespace is tree-structured, the inverted indices are used for directory tree structure reconstruction. Second, we log the differential updates of each metadata page to the metadata persistence log and compact them into fewer pages, and we call it *metadata persistence logging* mechanism. These logs are used for directory tree content update on reconstruction.

Fortunately, flash memory properties can be leveraged to keep overhead of the two mechanisms low. First, page metadata, the spare space alongside each flash page, is used to store the inverted index. The inverted index is atomically accessed with its page data without extra overhead [10]. Second, unindexed pages are tracked in the unindexed zone by limiting new allocations to a continuous logical space. The address mapping table in FTL redirects the writes to different physical pages, and the performance is not affected even though the logical layout is changed. Third, high random read performance makes the compact logging possible, as the reads of corresponding base pages are fast during recovery. As such, ReconFS can efficiently gain performance and endurance benefits with rather low overhead.

Our contributions are summarized as follows:

- We propose a reconstructable file system design to avoid the high overhead of maintaining a persistent directory tree and emulate hierarchical namespace access using a volatile directory tree in memory.
- We provide namespace consistency by embedding an inverted index with the indexed data and eliminate the pointer update in the parent node (in the directory tree view) to reduce writeback frequency.
- We also provide metadata persistence by logging and compacting dirty parts from multiple metadata pages to the metadata persistence log, and the compact form reduces metadata writeback size.
- We implement ReconFS based on ext2 and evaluate it against different file systems, including ext2, ext3, btrfs and f2fs. Results show an up to 46.3% performance increase and 27.1% endurance improvement compared to ext2, a file system with low metadata overhead.

The rest of this paper is organized as follows. Section 2 gives the background of flash memory and namespace management. Section 3 describes the ReconFS design, including the decoupled volatile and persistent directory tree maintenance, the embedded connectivity and metadata persistence logging mechanisms, as well as the reconstruction. We present the implementation in Section 4 and evaluate ReconFS in Section 5. Related work is given in Section 6, and the conclusion is made in Section 7.

2 Background

2.1 Flash Memory Basics

Programming in flash memory is performed in one direction. Flash memory cells need to be erased before overwritten. The read/write unit is a flash page (e.g., 4KB), and the erase unit is a flash block (e.g., 64 pages). In each flash page, there is a spare area for storing the metadata of the page, which is called page metadata or out-of-band (OOB) area [10]. The page metadata is used to store error correction codes (ECC). And it has been proposed to expose the page metadata to software in NVMe standard [6].

Flash translation layers (FTLs) are used in flash-based solid state drives (SSDs) to export the block interface [10]. FTLs translate the logical page number in the software to the physical page number in flash memory. The address mapping hides the no-overwrite property from the system software. FTLs also perform garbage collection to reclaim space and wear leveling to extend the lifetime of the device.

Flash-based SSDs provide higher bandwidth and IOPS compared to hard disk drives (HDDs) [10]. Multiple chips are connected through multiple channels inside an SSD to provide internal parallelism, providing high aggregated bandwidth. Due to elimination of mechanical moving part, an SSD provides high IOPS. Endurance is another element that makes flash-based SSDs different from HDDs [10, 14, 17, 23]. Each flash memory cell has limited program/erase (P/E) cycles. As the P/E cycles approach the limit, the reliability of each cell drops dramatically. As such, endurance is a critical issue in system designs on flash-based storage.

2.2 Hierarchical Namespaces

Directory trees have been used in different file systems for over three decades to manage data in a hierarchical way. But hierarchical namespaces introduce high overhead to provide consistency and persistence for the directory tree. Also, static metadata organization amplifies the metadata write size.

Namespace Consistency and Persistence. Directories and files are indexed in a tree structure, the directory tree. Each page uses pointers to index its children in the directory tree. To keep the consistency of the directory

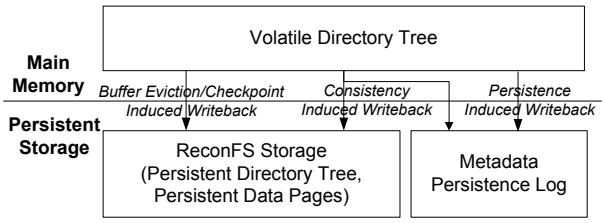


Figure 1: ReconFS Framework

tree, the page that has the pointer and the pointed page should be updated atomically. Different mechanisms, such as journaling [4, 7, 8, 34, 35] and copy-on-write (COW) [2, 32], are used to provide atomicity, but introduce a large amount of extra writes. In addition, the persistence requires the pointers to be in a durable state even after power failures, and this demands in-time writeback of these pages. This increases the writeback frequency, which also has a negative impact on endurance.

In this paper, we focus on the consistency of the directory tree, i.e., the metadata consistency. Data consistency can be achieved by incorporating transactional flash techniques [22, 23, 28, 29].

Metadata Organization. Namespace metadata are clustered and stored in the storage media, which we refer to as *static compacting*. Static compacting is commonly used in file systems. In ext2, index nodes in each block group are stored continuously. Since each index node is of 128 bytes in ext2, a 4KB page can store as many inodes as 32. Directory entries are organized in the similar way except that each directory entry is of variable length. Multiple directory entries with the same parent directory may share the same directory entry page. This kind of metadata organization improves the metadata performance in hard disk drives, as the metadata can be easily located.

Unfortunately, this kind of metadata organization has not addressed the endurance problem. For each file system operation, multiple metadata pages may be written but with only small parts updated in each page. E.g., a file create operation creates an inode in the inode page and writes a directory entry to the directory entry page. Since the flash-based storage is written in the unit of pages, the write amount is exaggerated by comparing the sum of all updated pages' size (from the view of storage device) with the updated metadata size (from the view of file system operations).

3 Design

ReconFS is designed to reduce the writes to flash storage while providing hierarchical namespace access. In this section, we first present the overall design of ReconFS, including the decoupled volatile and persistent directory tree maintenance and four types of metadata writeback. We then describe two mechanisms, *embedded connectivity* and *metadata persistence logging*, which provide consistency and persistence of the persistent directory tree with reduced writes, respectively. Finally, we discuss the ReconFS reconstruction.

3.1 Overview of ReconFS

ReconFS decouples the maintenance of the volatile and persistent directory trees. ReconFS emulates a volatile directory tree in main memory to provide the hierarchical namespace access. Metadata pages are updated to the volatile directory tree without being written back to the persistent directory tree. While the reduced writeback can benefit both performance and endurance of flash storage, consistency and persistence of the persistent directory tree need to be provided in case of unexpected system failures. Instead of writing back metadata pages directly to their home locations, ReconFS either embeds the inverted index with the indexed data for namespace consistency or compacts and writes back the scattered small updates in a log-structured way.

As shown in Figure 1, ReconFS is composed of three parts: the Volatile Directory Tree, the ReconFS Storage, and the Metadata Persistence Log. The Volatile Directory Tree manages namespace metadata pages in main memory to provide hierarchical namespace access. The ReconFS Storage is the persistent storage for ReconFS file system. It stores both the data and metadata, including the persistent directory tree, of the file system. The Metadata Persistence Log is a continuously allocated space in the persistent storage which is mainly used for the metadata persistence.

3.1.1 Decoupled Volatile and Persistent Directory Tree Maintenance

Since ReconFS emulates the hierarchical namespace access in main memory using a volatile directory tree, three issues are raised. First, namespace metadata pages need replacement when memory pressure is high. Second, namespace consistency is not guaranteed once system crashes without namespace metadata written back in time. Third, updates to the namespace metadata may get lost after unexpected system failures.

For the first issue, ReconFS writes back the namespace metadata to their home locations in ReconFS storage when they are evicted from the buffer, which we call *write-back on eviction*. This guarantees the metadata in persistent storage that do not have copies in main memory are the latest. Therefore, there are three kinds

of metadata in persistent storage (denoted as M_{disk}): the up-to-date metadata written back on eviction (denoted as $M_{up-to-date}$), the untouched metadata that have not been read into memory (denoted as $M_{untouched}$) and the obsolete metadata that have copies in memory (denoted as $M_{obsolete}$). Note $M_{obsolete}$ includes both pages that have dirty or clean copies in memory. Let M_{vdt}, M_{pdt} respectively be the namespace metadata of the volatile and persistent directory trees and M_{memory} be the volatile namespace metadata in main memory, we have

$$M_{vdt} = M_{memory} + M_{up-to-date} + M_{untouched},$$

$$M_{pdt} = M_{disk} = M_{obsolete} + M_{up-to-date} + M_{untouched}.$$

Since $M_{up-to-date}$ and M_{memory} are the latest, M_{vdt} is the latest. In contrast, M_{pdt} is not up-to-date, as ReconFS does not write back the metadata that still have copies in main memory. Volatile metadata are written back to their home locations for three cases: (1) file system unmount, (2) unindexed zone switch (Section 3.2), and (3) log truncation (Section 3.3). We call the operation that makes $M_{pdt} = M_{vdt}$ the *checkpoint* operation. When the volatile directory tree is checkpointed on unmount, it can be reconstructed by directly reading the persistent directory tree for later system booting.

The second and third issues are raised from unexpected system crashes, in which cases, $M_{vdt} \neq M_{pdt}$. The writeback of namespace metadata not only provides namespace connectivity for updated files or directories, but also keeps the descriptive metadata in metadata pages (e.g., owner, access control list in an inode) up-to-date. The second issue is caused by the loss of connectivity. To overcome this problem, ReconFS embeds an inverted index in each page for connectivity reconstruction (Section 3.2). The third issue is from the loss of metadata update. This problem is addressed by logging the metadata that need persistence (e.g., fsync) to the metadata persistence log (Section 3.3). In this way, the metadata of volatile directory tree can be reconstructed by first the connectivity reconstruction and then the descriptive metadata update even after system crashes.

3.1.2 Metadata Writeback

Metadata writeback to persistent storage, including the file system storage and the metadata persistence log, can be classified into four types as follows:

- *Buffer eviction induced writeback*: Metadata pages that are evicted due to memory pressure are written back to their home locations, so that these pages can be directly read out for later accesses without looking up the logs.
- *Checkpoint induced writeback*: Metadata pages are written back to their home locations for checkpoint operations, in order to reduce the reconstruction overhead.
- *Consistency induced writeback*: Writeback of pointers (used as the indices) is eliminated by embedding an inverted index with the indexed data of the flash storage, so as to reduce the writeback frequency.
- *Persistence induced writeback*: Metadata pages written back due to persistence requirements are compacted and logged to the metadata persistence log in a compact form to reduce the metadata writeback size.

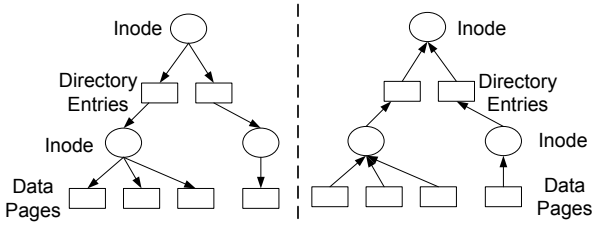


Figure 2: Normal Indexing (left) and Inverted Indexing (right) in a Directory Tree

3.2 Embedded Connectivity

Namespace consistency is one of the reasons why namespace metadata need frequent writeback to persistent storage. In the normal indexing of a directory tree as shown in the left half of Figure 2, the pointer and the pointed page of each link should be written back atomically for namespace consistency in each metadata operation. This not only requires the two pages to be updated but also demands journaling or ordered update for consistency. Instead, ReconFS provides namespace consistency using inverted indexing, which embeds the inverted index with the indexed data, as shown in the right half of Figure 2. Since the pointer is embedded with the pointed page, the consistency can be easily achieved. As well as the journal writes, the pointer updates are eliminated. In this way, the embedded connectivity lowers the frequency of metadata writeback and ensures the metadata consistency.

Embedded Inverted Index: In a directory tree, there are two kinds of links: links from directory entries to inodes (*dirent-inode links*) and links from inodes to data pages (*inode-data links*). Since directory entries are stored as data pages of directories in Unix/Linux, links from inodes to directory entries are classified as the inode-data links. For an inode-data link, the inverted index is the inode number and the data's location (i.e., the offset and length) in the file or directory. Since the inverted index is of several bytes, it is stored in the page

metadata of each flash page. For a dirent-inode link, the inverted index is the file or directory name and its inode number. Because the name is of variable length and is difficult to fit into the page metadata, an operation record, which is composed of the inverted index, the inode content and the operation type, is generated and stored in the metadata persistence log. The operation type in the operation record is set to 'creation' for create operations and 'link' for hard link operations. During reconstruction, the 'link' type does not invalidate previous creation records, while the 'creation' does.

An inverted index is also associated with a version number for identifying the correct version in case of inode number or directory entry reuses. When an inode number or a directory entry is reused after it is deleted, pages that belong to the deleted file or directory may still reside in persistent storage with their inverted indices. During reconstruction, these pages may be wrongly regarded as valid. To avoid this ambiguity, each directory entry is extended with a version number, and each inode is extended with the version pair $< V_{born}, V_{cur} >$, which indicates the liveness of the inode. V_{born} is the version number when the inode is created or reused. For a delete operation, V_{born} is set by increasing one to V_{cur}. Because all pages at that time have version numbers no larger than V_{cur}, all data pages of the deleted inode are set invalid. As same as the create and hard link operations, a delete operation generates a deletion record and appends it to the metadata persistence log, which is used to disconnect the inode from the directory tree and invalid all its children pages.

Unindexed Zone: Pages whose indices have not been written back are not accessible in the directory tree after system failures. These pages are called *unindexed pages* and need to be tracked for reconstruction. ReconFS divides the logical space into several zones and restricts the writes to one zone in each stage. This zone is called the *unindexed zone*, and it tracks all unindexed pages at one stage. A stage is the time period when the unindexed zone is used for allocation. When the zone is used up, the unindexed zone is switched to another. Before the zone switch, a checkpoint operation is performed to write the dirty indices back to their home locations. The restriction of writes to the unindexed zone incurs little performance penalty. This is because the FTL inside an SSD remaps logical addresses to physical addresses, and data layout in the logical space view does little impact on system performance while data layout in the physical space view is critical.

In addition to namespace connectivity, bitmap write-back is another source of frequent metadata persistence. The bitmap updates are frequently written back to keep the space allocation consistent. ReconFS only keeps the volatile bitmap in main memory, which is used for logical space allocation, and does not keep the persistent bitmap up-to-date. Once system crashes, bitmaps are reconstructed. Since new allocations are performed only in the unindexed zone, the bitmap in the unindexed zone is reconstructed using the valid and invalid statuses of the pages. Bitmaps in other zones are only updated when pages are deleted, and these updates can be reconstructed using deletion records in the metadata persistence log.

3.3 Metadata Persistence Logging

Metadata persistence causes frequent metadata write-back. The scattered small update pattern of the writeback amplifies the metadata writes, which are written back in the unit of pages. Instead of using static compacting (as mentioned in Section 2), ReconFS dynamically compacts the metadata updates and writes them to the metadata persistence log. While static compacting requires the metadata updates written back to their home locations, dynamic compacting is able to cluster the small updates in a compact form. Dynamic compacting only writes the dirty parts rather than the whole pages, so as to reduce write size.

In metadata persistence logging, writeback is triggered when persistence is needed, e.g., explicit synchronization or the wake up of pdflush daemon. The metadata persistence logging mechanism keeps track of the dirty parts of each metadata page in main memory and compacts those parts into the logs:

- Memory Dirty Tagging: For each metadata operation, metadata pages are first updated in the main memory. ReconFS records the location metadata (i.e., the offset and the length) of the dirty parts in each updated metadata page. The location metadata are attached to the buffer head of the metadata page to track the dirty parts for each page.
- Writeback Compacting: During writeback, ReconFS travels multiple metadata pages and appends their dirty parts to the log pages. Each dirty part has its location metadata (i.e., the base page address, the offset and length in the page) attached in the head of each log page.

Log truncation is needed when the metadata persistence log runs short of space. Instead of merging the small updates in the log with base metadata pages, ReconFS performs a checkpoint operation to write back all dirty metadata pages to their home locations. To mitigate the writeback cost, the checkpoint operation is performed in an asynchronous way using a writeback daemon, and the daemon starts when the log space drops below a pre-defined threshold. As such, the log is truncated without costly merging operations.

Multi-page Update Atomicity. Multi-page update atomicity is needed for an operation record which size

is larger than one page (e.g., a file creation operation with a 4KB file name). To provide the consistency of the metadata operation, these pages need to be updated atomically. Single-page update atomicity is guaranteed in flash storage, because the no-overwrite property of flash memory requires the page to be updated in a new place followed by atomic mapping entry update in the FTL mapping table.

Multi-page update atomicity is simply achieved using a flag bit in each page. Since a metadata operation record is written in continuously allocated log pages, the atomicity is achieved by tagging the start and end of these pages. The last page is tagged with flag '1', and the others are tagged with '0'. The bit is stored in the head of each log page. It is set when the log page is written back, and it does not require extra writes. During recovery, the flag bit '1' is used to determine the atomicity. Pages between two '1's belong to complete operations, while pages at the log tail without an ending '1' belong to an incomplete operation. In this way, multi-page update atomicity is achieved.

3.4 ReconFS Reconstruction

During normal shutdowns, the volatile directory tree writes the checkpoint to the persistent directory tree in persistent storage, which is simply read into main memory to reconstruct the volatile directory tree for the next system start. But once the system crashes, ReconFS needs to reconstruct the volatile directory tree using the metadata recorded by the embedded connectivity and the metadata persistence logging mechanisms. Since the persistent directory tree is the checkpoint of volatile directory tree when the unindexed zone is switched or the log is truncated, all page allocations are performed in the unindexed zone, and all metadata changes have been logged to the persistent metadata logs. Therefore, ReconFS only needs to update the directory tree by scanning the unindexed zone and the metadata persistence log. ReconFS reconstruction includes:

1. *File/directory reconstruction*: Each page in the unindexed zone is connected to its index node using its inverted index. And then, each page checks the version number in its inverted index with the $< V_{born}, V_{cur} >$ in its index node. If this matches, the page is indexed to the file or directory. Otherwise, the page is discarded because the page has been invalidated. After this, all pages, including file data pages and directory entry pages, are indexed to their index nodes.
2. *Directory tree connectivity reconstruction*: The metadata persistence log is scanned to search the dirent-inode links. These links are used to connect those inodes to the directory tree, so as to update the directory tree structure.
3. *Directory tree content update*: Log records in the metadata persistence log are used to update the metadata pages in the directory tree, so the content of the directory tree is updated to the latest.
4. *Bitmap reconstruction*: The bitmap in the unindexed zone is reset by checking the valid status of each page, which can be identified using version numbers. Bitmaps in other zones are not changed except for deleted pages. With the deletion or truncation log records, the bitmaps are updated.

After the reconstruction, those obsolete metadata pages in persistent directory tree are updated to the latest, and the recent allocated pages are indexed into the directory tree. The volatile directory tree is reconstructed to provide hierarchical namespace access.

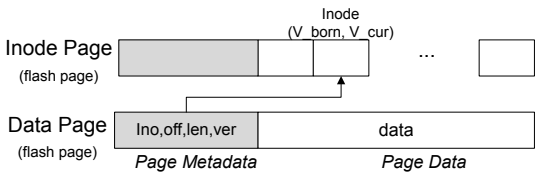


Figure 3: An Inverted Index for an Inode-Data Link

4 Implementation

ReconFS is implemented based on ext2 file system in Linux kernel 3.10.11. ReconFS shares both on-disk and in-memory data structures of ext2 but modifies the namespace metadata writeback flows.

In volatile directory directory tree, ReconFS employs two dirty flags for each metadata buffer page: persistence dirty and checkpoint dirty. *Persistence dirty* is tagged for the writeback to the metadata persistence log. *Checkpoint dirty* is tagged for the writeback to the persistent directory tree. Both of them are set when the buffer page is updated. The persistence dirty flag is cleared only when the metadata page is written to the metadata persistence log for metadata persistence. The checkpoint dirty flag is cleared only when the metadata are written back to its home location. ReconFS uses the double dirty flags to separate metadata persistence (the metadata persistence log) from metadata organization (the persistent directory tree).

In embedded connectivity, inverted indices for inode-data and dirent-inode links are stored in different ways. The inverted index of an *inode-data link* is stored in the page metadata of each flash page. It has the form of (ino, off, len, ver), in which *ino* is the inode number, off and len are the offset and the valid data length in the file or directory, respectively, and *ver* is the version number of the inode. The inverted index of a *dirent-*

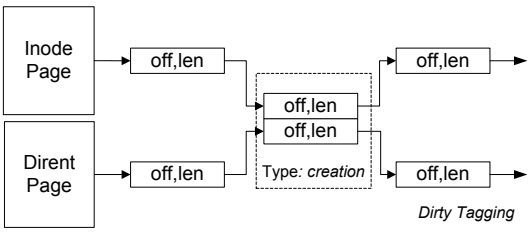


Figure 4: Dirty Tagging in Main Memory

inode link is stored as a log record with the record type *type* set to 'creation' in the metadata persistence log. The log record contains both the directory entry and the inode content and keeps an (off, len, lba, ver) extent for each of them. *lba* is the logical block address of the base metadata page. The log record acts as the inverted index for the inode, which is used to reconnect it to the directory tree. Unindexed zone in ReconFS is set by clustering multiple block groups in ext2. ReconFS limits the new allocations to these block groups, thus making these block groups as the unindexed zone. The addresses of these block groups are kept in file system super block and are made persistent on each zone switch.

In metadata persistence logging, ReconFS tags the dirty parts of each metadata page using a linked list, as shown in Figure 4. Each node in the linked list is a pair of (off, len) to indicate which part is dirty. Before each insertion, the list is checked to merge the overlapped dirty parts. The persistent log record also associates the type *type*, the version number *ver* and the logical block address *lba* for each metadata page with the linked list pairs, followed by the dirty content. In current implementation, ReconFS writes the metadata persistence log as a file in the root file system. Checkpoint is performed for file system unmount, unindexed zone switch or log truncation. Checkpoint for file system unmount is performed when the unmount command is issued, while checkpoint for the other two is triggered when the free space in the unindexed zone or the metadata persistence log drops below 5%.

Reconstruction of ReconFS is performed in three phases:

1. Scan Phase: Page metadata from all flash pages in the unindexed zone and log records from the metadata persistence log are read into memory. After this, all addresses of the metadata pages that appear in either of them are collected. And then, all these metadata pages are read into memory.
2. Zone Processing Phase: In the unindexed zone, each flash page is connected to its inode using the inverted index in its page metadata. Structures of files and directories are reconstructed, but they may have obsolete pages.
3. Log Processing Phase: Each log record is used either to connect a file or directory to the directory tree or to update the metadata page content. For a creation or hard link log record, the directory entry is updated for the inode. For a deletion or truncation log record, the corresponding bitmaps are read and updated. The other log records are used to update the page content. And finally, versions in the pages and inodes are checked to discard the obsolete pages, files and directories.

Table 1: File Systems

ext2	a traditional file system without journaling
ext3	a traditional journaling file system (journaled version of ext2)
btrfs[2]	a recent copy-on-write (COW) file system
f2fs[12]	a recent log-structured file system optimized for flash

5 Evaluation

We evaluate the performance and endurance of ReconFS against previous file systems, including ext2, ext3, btrfs and F2FS, and aim to answer the following four questions:

1. How does ReconFS compare with previous file systems in terms of performance and endurance?
2. What kind of operations gain more benefits from ReconFS? What are the benefits from embedded connectivity and metadata persistence logging?
3. What is the impact of changes in memory size?
4. What is the overhead of checkpoint and reconstruction in ReconFS?

In this section, we first describe the experimental setup before answering the above questions.

5.1 Experimental Setup

We implement ReconFS in Linux kernel 3.10.11, and evaluate the performance and endurance of ReconFS against the file systems listed in Table 1.

We use four workloads from filebench benchmark [3]. They emulate different types of servers. Operations and read-write ratio [21] of each workload are illustrated as follows:

- *fileserver* emulates a file server, which performs a sequence of create, delete, append, read, write and attribute operations. The read-write ratio is 1:2.
- *webproxy* emulates a web proxy server, which performs a mix of create-write-close, open-read-close and delete operations, as well as log appends. The read-write ratio is 5:1.

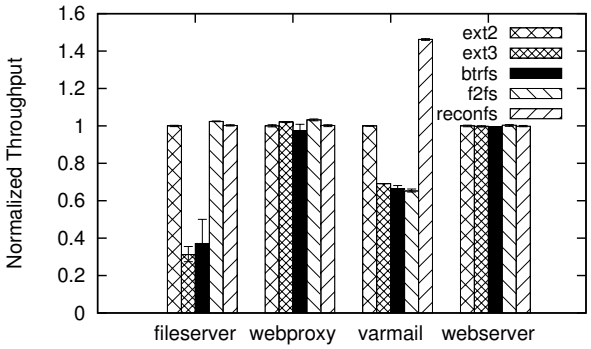


Figure 5: System Comparison on Performance

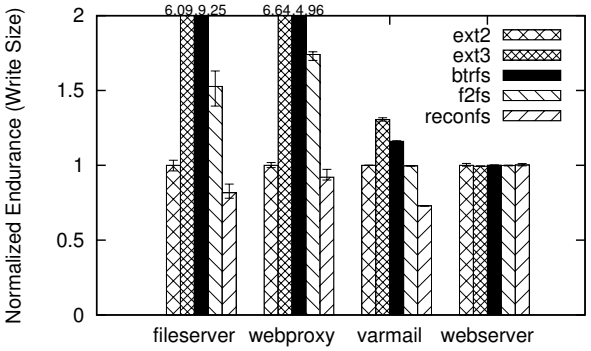


Figure 6: System Comparison on Endurance

Table 2: SSD Specification

Capacity	128 GB
Seq. Read Bandwidth	260 MB/s
Seq. Write Bandwidth	200 MB/s
Rand. Read IOPS (4KB)	17,000
Rand. Write IOPS (4KB)	5,000

- *varmail* emulates a mail server, which performs a set of create-append-sync, read-append-sync, read and delete operations. The read-write ratio is 1:1.
- *webserver* emulates a web server, which performs open-read-close operations, as well as log appends. The read-write ratio is 10:1.

Experiments are carried out on Fedora 10 using Linux kernel 3.10.11, and the computer is equipped with 4-core 2.50GHz processor and 12GB memory. We evaluate all file systems on a 128GB SSD, and its specification is shown in Table 2. All file systems are mounted with default options.

5.2 System Comparison

5.2.1 Overall Comparison

We evaluate the performance of all file systems by measuring the throughput reported by the benchmark, and the endurance by measuring the write size to storage. The write size to storage is collected from the block level trace using blktrace tool [1].

Figure 5 shows the throughput normalized to the throughput of ext2 to evaluate the performance. As shown in the figure, ReconFS is among the best of all file systems for all evaluated workloads, and gains performance improvement up to 46.3% than ext2 for varmail, the metadata intensive workload. For read intensive workloads, such as webproxy and webserver, all evaluated file systems do not show a big difference. But for write intensive workloads, such as fileserver and varmail, they show different performance. Ext2 shows comparatively higher performance than other file systems excluding ReconFS. Both ext3 and btrfs have provided namespace consistency with different mechanisms, e.g., waiting until the data reach persistent storage before writing back the metadata, but with poorer performance compared to ext2. F2FS, the file system with data layout optimized for flash, shows a comparable performance to ext2, but has inferior performance in varmail workload, which is metadata intensive and has frequent fsyncs. Comparatively, ReconFS achieves the performance of ext2 in all evaluated workloads, nearly the best performance of all previous file systems, and is even better than ext2 in varmail workload. Moreover, ReconFS provides namespace consistency with embedded connectivity while ext2 does not.

Figure 6 shows the write size to storage normalized to that of ext2 to evaluate the endurance. From the figure, we can see ReconFS effectively reduces write size for metadata and reduces write size by up to 27.1% compared to ext2. As same as the performance, the endurance of ext2 is the best of all file systems excluding ReconFS. On the while, ext3, btrfs and F2FS uses journaling or copy-on-write to provide consistency, which introduces extra writes. For instance, btrfs has the write size 9 times as large as that of ext2 in the fileserver workload. ReconFS provides namespace consistency using embedded connectivity without incurring extra writes, and further reduces write size by compacting metadata writeback. As shown in the figure, ReconFS shows a write size reduction of 18.4%, 7.9% and 27.1% even compared with ext2 respectively for fileserver, webproxy and varmail workloads.

5.2.2 Performance

To understand the performance impact of ReconFS, we evaluate four different operations that have to update the index node page and/or directory entry page. The four operations are file creation, deletion, append and append with fsyncs. They are evaluated using micro-

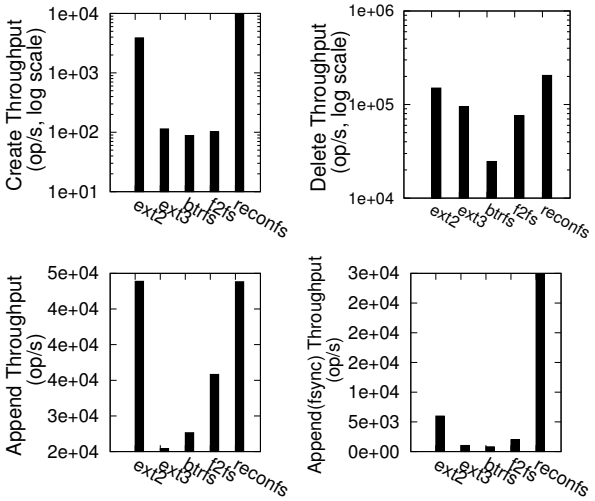


Figure 7: Performance Evaluation of Operations (File create, delete, append and append with fsync)

benchmarks. The file creation and deletion benchmarks create or delete 100K files spread over 100 directories. *fsync* is performed following each creation. The append benchmark appends 4KB pages to a file, and it inserts a fsync for every 1,000 (one fsync per 4MB) and 10 (one fsync per 40KB) append operations respectively for evaluating append and append with fsyncs.

Figure 7 shows the throughput of the four operations. ReconFS shows a significant throughput increase in file creation and append with fsyncs. File creation throughput in ReconFS doubles the throughput in ext2. This is because only one log page is appended in the metadata persistence log, while multiple pages need to be written back in ext2. Other file systems have even worse file creation performance due to consistency overheads. File deletion operations in ReconFS also show better performance than the others. File append throughput in ReconFS almost equals that in ext2 for append operations with one fsync per 1,000 append operations. But file append (with fsyncs) throughput in ext2 drops dramatically as the fsync frequency increases from 1/1000 to 1/10, as well as in the other journaling or log-structured file systems. In comparison, file append (with fsyncs) throughput in ReconFS only drops to half of previous throughput. When fsync frequency is 1/10, ReconFS has file append throughput 5 times better than ext2 and orders of magnitude better than the other file systems.

5.2.3 Endurance

To further investigate the endurance benefits of ReconFS, we measure the write size of ext2, ReconFS without log compacting (denoted as ReconFS-EC) and ReconFS.

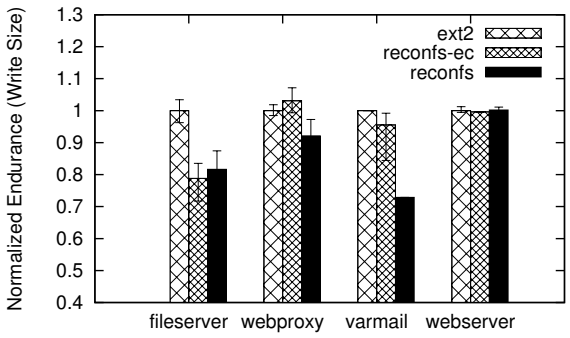


Figure 8: Endurance Evaluation for Embedded Connectivity and Metadata Persistence Logging

Figure 8 shows write sizes of the three file systems. We compare the write sizes of ext2 and ReconFS-EC to evaluate the benefit from embedded connectivity, since ReconFS-EC implements the embedded connectivity but without log compacting. From the figure, we observe that the fileserver workload shows a remarkable drop in write size from ext2 to ReconFS-EC. The benefit mainly comes from the intensive file creates and appends in the fileserver workload, which otherwise requires index pointers to be updated for namespace connectivity. Embedded connectivity in ReconFS eliminates updates to these index pointers. We also compare the write sizes of ReconFS-EC and ReconFS to evaluate the benefit from log compacting in metadata persistence logging. As shown in the figure, ReconFS shows a large write reduction in varmail workload. This is because frequent fsyncs reduce the effects of buffering, in other words, the updates to metadata pages are small when written back. As a result, the log compacting gains more improvement than other workloads.

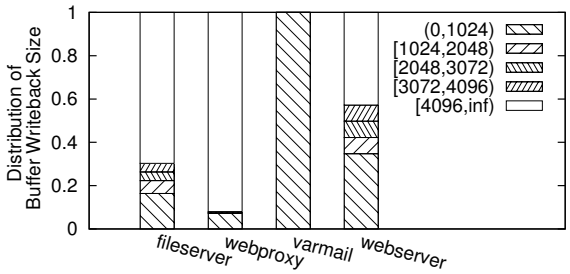
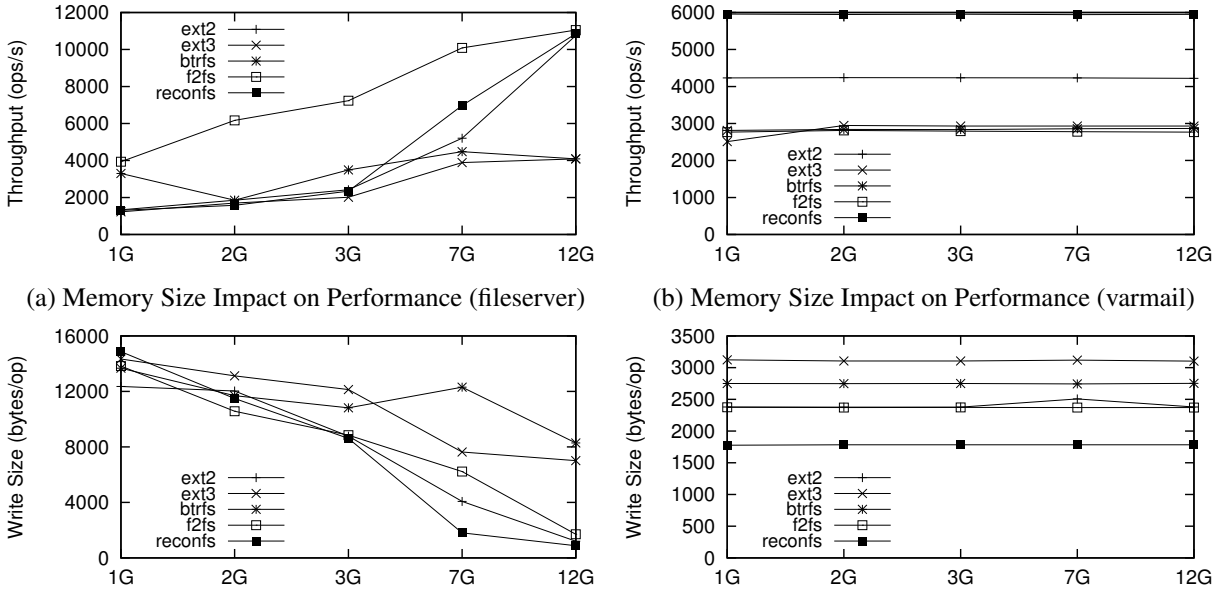


Figure 9: Distribution of Buffer Page Writeback Size

Figure 9 also shows the distribution of buffer page writeback size, which is the size of dirty parts in each page. As shown in the figure, over 99.9% of the dirty data for each page in metadata writeback of varmail workload are less than 1KB due to frequent fsyncs, while the others have the fraction varied from 7.3% to 34.7%



(a) Memory Size Impact on Performance (fileserver)

(b) Memory Size Impact on Performance (varmail)

(c) Memory Size Impact on Endurance (fileserver)

(d) Memory Size Impact on Endurance (varmail)

Figure 10: Memory Size Impact on Performance and Endurance

Table 3: Comparision of Full-Write and Compact-Write

Workloads	Full Write Size (KB)	Comp. Write Size (KB)	Compact Ratio
fileserver	108,143	48,624	44.96%
webproxy	45,133	21,325	47.25%
varmail	3,060,116	117,235	3.83%
webserver	374	143	38.36%

for dirty size less than 1KB. In addition, we calculate the compact ratio by dividing the full page update size with the compact write size, as shown in Table 3. The compact ratio of varmail workload achieves as low as 3.83%.

5.3 Impact of Memory Size

To study the memory size impact, we set the memory size to 1, 2, 3, 7 and 12 gigabytes[1] and measure both performance and endurance of all evaluated file systems. We measure performance in the unit of the operations per second (ops/s), and endurance in the unit of bytes per operation (bytes/op) by dividing the total write size with the number of operations. Results of webproxy and webserver workloads are not shown due to space limitation, as they are read intensive workloads and show little difference between file systems.

[1]We limit the memory size to 1, 2, 4, 8 and 12 gigabytes in the GRUB. The recognized memory sizes (shown in /proc/meminfo) are 997, 2,005, 3,012, 6,980 and 12,044 megabytes, respectively.

Figure 10 (a) shows the throughput of fileserver workload for all file systems under different memory sizes. As shown in the figure, ReconFS gains more when memory size becomes larger, in which case data pages are written back less frequently and the writeback of metadata pages has larger impact. When memory size is small and memory pressure is high, the impact of data writes dominates. ReconFS has poorer performance than F2FS, which has optimized data layout. When memory size increases, the impact from the metadata writes increases. Little improvement is gained in ext3 and btrfs when memory size increases from 7GB to 12GB. In contrast, ReconFS and ext2 gain significant improvement for their low metadata overhead and approach the performance of F2FS. Figure 10 (c) shows the endurance measured in bytes per operation of fileserver. In the figure, ReconFS has comparable or less write size than other file systems.

Figure 10 (b) shows the throughput of varmail workload. Performance is stable under different memory sizes, and ReconFS achieves the best performance. This is because varmail workload is metadata intensive workload and has frequent fsync operations. Figure 10 (d) shows the endurance of varmail workload. ReconFS achieves the best in all file systems.

5.4 Reconstruction Overhead

We measure the unmount time to evaluate the overhead of checkpoint, which writes back all dirty metadata to make the persistent directory tree equivalent to the

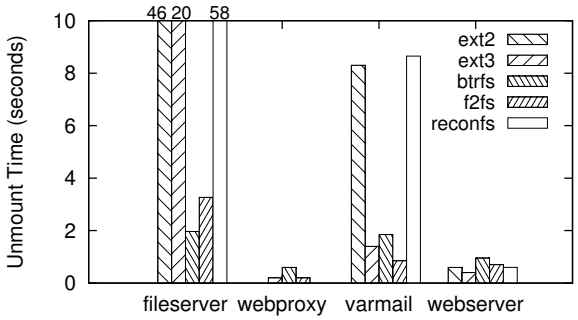


Figure 11: Unmount Time (Immediate Unmount)

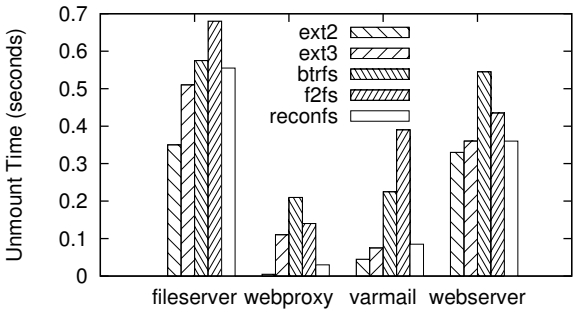


Figure 12: Unmount Time (Unmount after 90s)

volatile directory tree, as well as the reconstruction time.

Unmount Time. We use *time* command to measure the time of unmount operations and use the elapsed time reported by the time command.

Figure 11 shows the unmount time when the unmount is performed immediately when each benchmark completes. The read intensive workloads, webproxy and webserver, have unmount time less than one second for all file systems. But the write intensive workloads have various unmount time for different file systems. The unmount time in ext2 is 46 seconds, while that of ReconFS is 58. All the unmount time values are less than one minute, and they include the time used for both data and metadata writeback. Figure 12 shows the unmount time when the unmount is performed 90 seconds later after each benchmark completes. All of them are less than one second, and ReconFS does not show a noticeable difference with others.

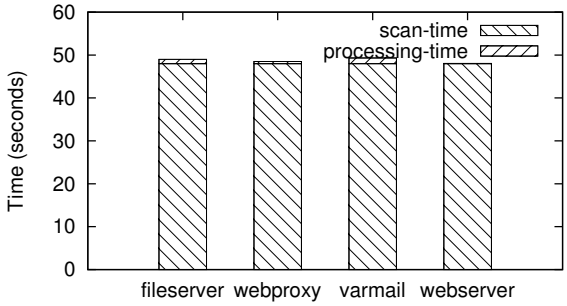


Figure 13: Recovery Time

Reconstruction Time. Reconstruction time has two main parts: scan time and processing time. The scan time includes the time of the unindexed zone scan and the log scan. The scan is the sequential read, which performance is bounded by the device bandwidth. The processing time is the time used to read the base metadata pages in the directory tree to be updated in addition to the recovery logic processing time. As shown in Figure 13, the scan time is 48 seconds for an 8GB zone on the SSD, and the processing time is around one second. The scan time is expected to be reduced with PCIe SSDs. E.g., the scan time for a 32GB zone on a PCIe SSD with 3GB/s is around ten seconds. Therefore, with high read bandwidth and IOPS, the reconstruction of ReconFS can complete in tens of seconds.

6 Related Work

File System Namespace. Research on file system namespace has been long for efficient and effective namespace metadata management. Relational database or table-based technologies have been used to manage namespace metadata for either consistency or performance. Inversion file system [26] manages namespace metadata using PostGRES database system to provide transaction protection and crash recovery to the metadata. TableFS [31] stores namespace metadata in LevelDB [5] to improve metadata access performance by leveraging the log-structured merge tree (LSM-tree) [27] implemented in LevelDB.

The hierarchical structure of namespace has also been discussed to be implemented in a flexible way to provide semantic accesses. Semantic file system [16] removes the tree-structured namespace and accesses files and directories using attributes. hFAD [33] proposes a similar approach, which prefers a search-friendly file system to a hierarchical file system.

Pilot [30] proposes an even aggressive way and eliminates all indexing in file systems, in which files are accessed only through a 64-bit universal identifier (UID). And Pilot does not provide tree-structured file access. Comparatively, ReconFS removes only the indexing of persistent storage to lower the metadata cost, and it emulates the tree-structured file access using the volatile directory tree.

Backpointers and Inverted Indices. Backpointers have been used in storage systems for different purposes. BackLog [24] uses backpointer in data blocks to reduce

the pointer updates when data blocks are moved due to advanced file system features, such as snapshots, clones. NoFS [15] uses backpointer for consistency checking on each read to provide consistency. Both of them use backpointer as the assistant to enhance new functions, but ReconFS uses backpointers (inverted indices) as the only indexing (without forward pointers).

In flash-based SSDs, backpointer (e.g., the logical page addresses) is stored in the page metadata of each flash page, which is atomically accessed with the page data, to recover the FTL mapping table [10]. On each device booting, all pages are scanned, and the FTL mapping table is recovered using the backpointer. OFSS [23] uses backpointer in page metadata in a similar way. OFSS uses an object-based FTL, and the backpointer in each page records the information of the object, which is used to delay the persistence of the object indexing. ReconFS extends the use of backpointer in flash storage to the file system namespace management. Instead of maintaining the indexing (forward pointers), ReconFS embeds only the reverse index (backward pointers) with the indexed data, and the reverse indices are used for reconstruction once system fails unexpectedly.

File System Logging. File systems have used logging in two different ways. One is the journaling, which updates metadata and/or data in the journaling area before updating them to their home locations, and is widely used in modern file systems to provide file system consistency [4, 7, 8, 34, 35]. Log-structured file systems use logging in the other way [32]. Log-structured file systems write all data and metadata in a logging way, making random writes sequential for better performance.

ReconFS employs the logging mechanism for metadata persistence. Unlike journaling file systems or log-structured file systems, which require tracking of valid and invalid pages for checkpoint and garbage cleaning, the metadata persistence log in ReconFS is simply discarded after the writeback of all volatile metadata. ReconFS also enables compact logging, because the base metadata pages can be read quickly during reconstruction due to high random read performance of flash storage.

File Systems on Flash-based Storage. In addition to embedded flash file systems [9, 36], researchers are proposing new general-purpose file systems for flash storage. DFS [19] is a file system that directly manages flash memory by leveraging functions (e.g., block allocation, atomic update) provided by FusionIO's ioDrive. Nameless Write [37] also removes the space allocation function in the file system and leverage the FTL space management for space allocation. OFSS [23] proposes to directly manage flash memory using an object-based FTL, in which the object indexing, free space management and data layout can be optimized with the flash memory characteristics. F2FS [12] is a promising log-structured file system which is designed for flash storage. It optimizes data layout in flash memory, e.g., the hot/cold data grouping. But these file systems have paid little attention to the high overhead of namespace metadata, which are frequently written back and are written in the scattered small write pattern. ReconFS is the first to address the namespace metadata problem on flash storage.

7 Conclusion

Properties of namespace metadata, such as intensive writeback and scattered small updates, make the overhead of namespace management high on flash storage in terms of both performance and endurance. ReconFS removes maintenance of the persistent directory tree and emulates hierarchical access using a volatile directory tree. ReconFS is reconstructable after unexpected system failures using both *embedded connectivity* and *metadata persistence logging* mechanisms. Embedded connectivity enables directory tree structure reconstruction by embedding the reverted index with the indexed data. With elimination of updates to parent pages (in the directory tree) for pointer updating, the consistency maintenance is simplified and the writeback frequency is reduced. Metadata persistence logging provides persistence to metadata pages, and the logged metadata are used for directory tree content reconstruction. Since only the dirty parts of metadata pages are logged and compacted in the logs, the writeback size is reduced. Reconstruction is fast due to high bandwidth and IOPS of flash storage. Through the new namespace management, ReconFS improves both performance and endurance of flash-based storage system without compromising consistency or persistence.

Acknowledgments

We would like to thank our shepherd Remzi Arpaci-Dusseau and the anonymous reviewers for their comments and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 60925006), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), Shanghai Key Laboratory of Scalable Computing and Systems, Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, Huawei Technologies Co. Ltd., and Tsinghua University Initiative Scientific Research Program.

References

[1] blktrace(8) - linux man page. http://linux.die.net/man/8/blktrace.

[2] Btrfs. http://btrfs.wiki.kernel.org.

[3] Filebench benchmark. http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Main_Page.

[4] Journaled file system technology for linux. http://jfs.sourceforge.net/.

[5] LevelDB, a fast and lightweight key/value database library by Google. https://code.google.com/p/leveldb/.

[6] The NVM express standard. http://www.nvmexpress.org.

[7] ReiserFS. http://reiser4.wiki.kernel.org.

[8] XFS: A high-performance journaling filesystem. http://oss.sgi.com/projects/xfs/.

[9] Yaffs. http://www.yaffs.net.

[10] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of 2008 USENIX Annual Technical Conference (USENIX'08)*, 2008.

[11] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, 2009.

[12] Neil Brown. An F2FS teardown. http://lwn.net/Articles/518988/.

[13] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, 2009.

[14] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, 2011.

[15] Vijay Chidambaram, Tushar Sharma, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[16] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. Semantic file systems. In *Proceedings of the thirteenth ACM Symposium on Operating Systems Principles (SOSP'91)*, 1991.

[17] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[18] Tyler Harter, Charlotte Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, 2011.

[19] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. DFS: a file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, 2010.

[20] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[21] Eunji Lee, Hyokyung Bahn, and Sam H Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[22] Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions. In *Proceedings of the 31st IEEE International Conference on Computer Design (ICCD'13)*, 2013.

[23] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[24] Peter Macko, Margo I Seltzer, and Keith A Smith. Tracking back references in a write-anywhere file system. In *Proceedings of the*

8th USENIX Conference on File and storage technologies (FAST'10), 2010.

[25] David Nellans, Michael Zappe, Jens Axboe, and David Flynn. ptrim ()+ exists (): Exposing new FTL primitives to applications. In *the 2nd Annual Non-Volatile Memory Workshop*, 2011.

[26] Michael A Olson. The design and implementation of the inversion file system. In *USENIX Winter*, 1993.

[27] 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.

[28] 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.

[29] 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.

[30] 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. *Communications of the ACM*, 23(2):81–92, 1980.

[31] Kai Ren and Garth Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of 2013 USENIX Annual Technical Conference (USENIX'13)*, 2013.

[32] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[33] Margo I Seltzer and Nicholas Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS XII)*, 2009.

[34] Stephen Tweedie. Ext3, journaling filesystem. In *Ottawa Linux Symposium*, 2000.

[35] Stephen C Tweedie. Journaling the linux ext2fs filesystem. In *The Fourth Annual Linux Expo*, 1998.

[36] David Woodhouse. Jffs2: The journalling flash file system, version 2. `http://sourceware.org/jffs2`.

[37] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

Toward strong, usable access control for shared distributed data

Michelle L. Mazurek, Yuan Liang, William Melicher, Manya Sleeper,
Lujo Bauer, Gregory R. Ganger, Nitin Gupta, and Michael K. Reiter*
*Carnegie Mellon University, *University of North Carolina at Chapel Hill*

Abstract

As non-expert users produce increasing amounts of personal digital data, usable access control becomes critical. Current approaches often fail, because they insufficiently protect data or confuse users about policy specification. This paper presents Penumbra, a distributed file system with access control designed to match users' mental models while providing principled security. Penumbra's design combines semantic, tag-based policy specification with logic-based access control, flexibly supporting intuitive policies while providing high assurance of correctness. It supports private tags, tag disagreement between users, decentralized policy enforcement, and unforgeable audit records. Penumbra's logic can express a variety of policies that map well to real users' needs. To evaluate Penumbra's design, we develop a set of detailed, realistic case studies drawn from prior research into users' access-control preferences. Using microbenchmarks and traces generated from the case studies, we demonstrate that Penumbra can enforce users' policies with overhead less than 5% for most system calls.

1 Introduction

Non-expert computer users produce increasing amounts of personal digital data, distributed across devices (laptops, tablets, phones, etc.) and the cloud (Gmail, Facebook, Flickr, etc.). These users are interested in accessing content seamlessly from any device, as well as sharing it with others. Thus, systems and services designed to meet these needs are proliferating [6,37,42,43,46,52].

In this environment, access control is critical. News headlines repeatedly feature access-control failures with consequences ranging from embarrassing (e.g., students accessing explicit photos of their teacher on a classroom iPad [24]) to serious (e.g., a fugitive's location being revealed by geolocation data attached to a photo [56]). The potential for such problems will only grow. Yet, at the same time, access-control configuration is a secondary task most users do not want to spend much time on.

Access-control failures generally have two sources: ad-hoc security mechanisms that lead to unforeseen behavior, and policy authoring that does not match users' mental models. Commercial data-sharing services sometimes fail to guard resources entirely [15]; often they manage access in ad-hoc ways that lead to holes [33]. Numerous studies report that users do not understand privacy settings or cannot use them to create desired policies (e.g., [14,25]). Popular websites abound with advice for these confused users [38,48].

Many attempts to reduce user confusion focus only on improving the user interface (e.g., [26, 45, 54]). While this is important, it is insufficient—a full solution also needs the underlying access-control infrastructure to provide principled security while aligning with users' understanding [18]. Prior work investigating access-control infrastructure typically either does not support the flexible policies appropriate for personal data (e.g., [20]) or lacks an efficient implementation with system-call-level file-system integration (e.g., [31]).

Recent work (including ours) has identified features that are important for meeting users' needs but largely missing in deployed access-control systems: for example, support for semantic policies, private metadata, and interactive policy creation [4, 28, 44]. In this paper, we present Penumbra, a distributed file system with access control designed to support users' policy needs while providing principled security. Penumbra provides for flexible policy specification meant to support real access-control policies, which are complex, frequently include exceptions, and change over time [8, 34, 35, 44, 53]. Because Penumbra operates below the user interface, we do not evaluate it directly with a user study; instead, we develop a set of realistic case studies drawn from prior work and use them for evaluation. We define "usability" for this kind of non-user-facing system as supporting specific policy needs and mental models that have been previously identified as important.

Penumbra's design is driven by three important factors. First, users often think of content in terms of its attributes, or *tags*—photos of my sister, budget spreadsheets, G-rated movies—rather than in traditional hierarchies [28, 47, 49]. In Penumbra, both content and policy are organized using tags, rather than hierarchically. Second, because tags are central to managing content, they must be treated accordingly. In Penumbra, tags are cryptographically signed first-class objects, specific to a

single user's namespace. This allows different users to use different attribute values to describe and make policy about the same content. Most importantly, this design ensures tags used for policy specification are resistant to unauthorized changes and forgery. Policy for accessing tags is set independently of policy for files, allowing for private tags. Third, Penumbra is designed to work in a distributed, decentralized, multi-user environment, in which users access files from various devices without a dedicated central server, an increasingly important environment [47]. We support multi-user devices; although these devices are becoming less common [13], they remain important, particularly in the home [27, 34, 61]. Cloud environments are also inherently multi-user.

This paper makes three main contributions. First, it describes Penumbra, the first file-system access-control architecture that combines semantic policy specification with logic-based credentials, providing an intuitive, flexible policy model without sacrificing correctness. Penumbra's design supports distributed file access, private tags, tag disagreement between users, decentralized policy enforcement, and unforgeable audit records that describe who accessed what content and why that access was allowed. Penumbra's logic can express a variety of flexible policies that map well to real users' needs.

Second, we develop a set of realistic access-control case studies, drawn from user studies of non-experts' policy needs and preferences. To our knowledge, these case studies, which are also applicable to other personal-content-sharing systems, are the first realistic policy benchmarks with which to assess such systems. These case studies capture users' desired policy goals in detail; using them, we can validate our infrastructure's efficacy in supporting these policies.

Third, using our case studies and a prototype implementation, we demonstrate that semantic, logic-based policies can be enforced efficiently enough for the interactive uses we target. Our results show enforcement also scales well with policy complexity.

2 Related work

In this section, we discuss four related areas of research.

Access-control policies and preferences. Users' access-control preferences for personal data are nuanced, dynamic, and context-dependent [3, 35, 44]. Many policies require fine-grained rules, and exceptions are frequent and important [34, 40]. Users want to protect personal data from strangers, but are perhaps more concerned about managing access and impressions among family, friends, and acquaintances [4, 12, 25, 32]. Furthermore, when access-control mechanisms are ill-suited to users' policies or capabilities, they fall back on clumsy, ad-hoc coping mechanisms [58]. Penumbra is designed to support personal polices that are complex, dynamic, and drawn from a broad range of sharing preferences.

Tags for access control. Penumbra relies on tags to define access-control policies. Researchers have prototyped tag-based access-control systems for specific contexts, including web photo albums [7], corporate desktops [16], microblogging services [17], and encrypting portions of legal documents [51]. Studies using role-playing [23] and users' own tags [28] have shown that tag-based policies are easy to understand and accurate policies can be created from existing tags.

Tags for personal distributed file systems. Many distributed file systems use tags for file management, an idea introduced by Gifford et al. [22]. Many suggest tags will eclipse hierarchical management [49]. Several systems allow tag-based file management, but do not explicitly provide access control [46, 47, 52]. Homeviews provides capability-based access control, but remote files are read-only and each capability governs files local to one device [21]. In contrast, Penumbra provides more principled policy enforcement and supports policy that applies across devices. Cimbiosys offers partial replication based on tag filtering, governed by fixed hierarchical access-control policies [60]. Research indicates personal policies do not follow this fixed hierarchical model [34]; Penumbra's more flexible logic builds policies around non-hierarchical, editable tags, and does not require a centralized trusted authority.

Logic-based access control. An early example of logic-based access control is Taos, which mapped authentication requests to proofs [59]. Proof-carrying authentication (PCA) [5], in which proofs are submitted together with requests, has been applied in a variety of systems [9, 11, 30]. PCFS applies PCA to a local file system and is evaluated using a case study based on government policy for classified data [20]. In contrast, Penumbra supports a wider, more flexible set of distributed policies targeting personal data. In addition, while PCFS relies on constructing and caching proofs prior to access, we consider the efficiency of proof generation.

One important benefit of logic-based access control is meaningful auditing; logging proofs provides unforgeable evidence of which policy credentials were used to allow access. This can be used to reduce the trusted computing base, to assign blame for unintended accesses, and to help users detect and fix policy misconfigurations [55].

3 System overview

This section describes Penumbra's architecture as well as important design choices.

3.1 High-level architecture

Penumbra encompasses an ensemble of devices, each storing files and tags. Users on one device can remotely access files and tags on other devices, subject to access control. Files are managed using semantic (i.e., tag-based) object naming and search, rather than a directory hierarchy. Users query local and remote files using tags, e.g., *type=movie* or *keyword=budget*. Access-control policy is also specified semantically, e.g., Alice might allow Bob to access files with the tags *type=photo* and *album=Hawaii*. Our concept of devices can be extended to the cloud environment. A cloud service can be thought of as a large multi-user device, or each cloud user as being assigned her own logical "device." Each user runs a *software agent*, associated with both her global public-key identity and her local uid, on every device she uses. Among other tasks, the agent stores all the *authorization credentials*, or cryptographically signed statements made by principals, that the user has received.

Each device in the ensemble uses a file-system-level *reference monitor* to control access to files and tags. When a system call related to accessing files or tags is received, the monitor generates a challenge, which is formatted as a logical statement that can be proved true only if the request is allowed by policy. To gain access, the requesting user's agent must provide a logical proof of the challenge. The reference monitor will verify the proof before allowing access. To make a proof, the agent assembles a set of relevant authorization credentials. The credentials, which are verifiable and unforgeable, are specified as formulas in an access-control logic, and the proof is a derivation demonstrating that the credentials are sufficient to allow access. Penumbra uses an intuitionistic first-order logic with predicates and quantification over base types, described further in Sections 3.3 and 4.

The challenges generated by the reference monitors have seven types, which fall into three categories: authority to read, write, or delete an existing file; authority to read or delete an existing tag; and authority to create content (files or tags) on the target device. The rationale for this is explained in Section 3.2. Each challenge includes a nonce to prevent replay attacks; for simplicity, we omit the nonces in examples. The logic is not exposed directly to users, but abstracted by an interface that is beyond the scope of this paper.

For both local and remote requests, the user must prove to her local device that she is authorized to access the content. If the content is remote, the local device (acting as client) must additionally prove to the remote device that the local device is trusted to store the content and enforce policy about it. This ensures that users of untrusted devices cannot circumvent policy for remote data. Figure 1 illustrates a remote access.

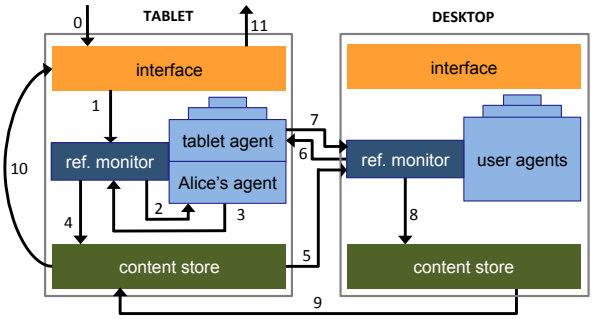


Figure 1: Access-control example. (0) Using her tablet, Alice requests to open a file stored on the desktop. (1) The interface component forwards this request to the reference monitor. (2) The local monitor produces a challenge, which (3) is proved by Alice's local agent, then (4) asks the content store for the file. (5) The content store requests the file from the desktop, (6) triggering a challenge from the desktop's reference monitor. (7) Once the tablet's agent proves the tablet is authorized to receive the file, (8) the desktop's monitor instructs the desktop's content store to send it to the tablet. (9–11) The tablet's content store returns the file to Alice via the interface component.

3.2 Metadata

Semantic management of access-control policy, in addition to file organization, gives new importance to tag handling. Because we base policy on tags, they must not be forged or altered without authorization. If Alice gives Malcolm access to photos from her Hawaiian vacation, he can gain unauthorized access to her budget if he can change its type from *spreadsheet* to *photo* and add the tag *album=Hawaii*. We also want to allow users to keep tags private and to disagree about tags for a shared file.

To support private tags, we treat each tag as an object independent of the file it describes. Reading a tag requires a proof of access, meaning that assembling a file-access proof that depends on tags will often require first assembling proofs of access to those tags (Figure 2).

For tag integrity and to allow users to disagree about tags, we implement tags as cryptographically signed credentials of the form *principal* signed tag(*attribute, value, file*). For clarity in examples, we use descriptive file names; in reality, Penumbra uses globally unique IDs. For example, Alice can assign the song "Thriller" a four-star rating by signing a credential: Alice signed tag(rating, 4, "Thriller"). Alice, Bob, and Caren can each assign different ratings to "Thriller." Policy specification takes this into account: if Alice grants Bob permission to listen to songs where Alice's rating is three stars or higher, Bob's rating is irrelevant. Because tags are signed, any principal is free to make any tag about any file. Principals

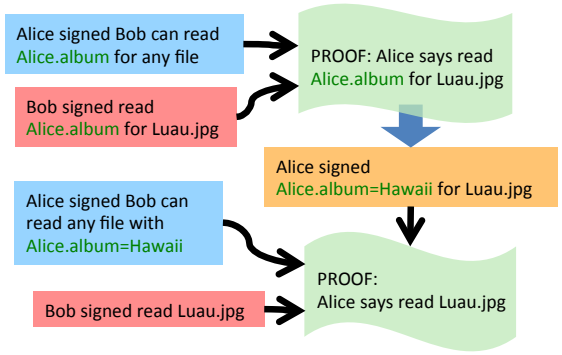


Figure 2: Example two-stage proof of access, expressed informally. In the first stage, Bob's agent asks which album Alice has placed the photo Luau.jpg in. After making the proof, Bob's agent receives a metadata credential saying the photo is in the album *Hawaii*. By combining this credential with Bob's authority to read some files, Bob's agent can make a proof that will allow Bob to open Luau.jpg.

can be restricted from storing tags on devices they do not own, but if Alice is allowed to create or store tags on a device then those tags may reference any file.

Some tags are naturally written as attribute-value pairs (e.g., *type=movie*, *rating=PG*). Others are commonly value-only (e.g., photos tagged with *vacation* or with people's names). We handle all tags as name-value pairs; value-only tags are transformed into name-value pairs, e.g., from "vacation" to *vacation=true*.

Creating tags and files. Because tags are cryptographically signed, they cannot be updated; instead, the old credential is revoked (Section 4.4) and a new one is issued. As a result, there is no explicit write-tag authority.

Unlike reading and writing, in which authority is determined per file or tag, authority to create files and tags is determined per device. Because files are organized by their attributes rather than in directories, creating one file on a target device is equivalent to creating any other. Similarly, a user with authority to create tags can always create any tag in her own namespace, and no tags in any other namespace. So, only authority to create any tags on the target device is required.

3.3 Devices, principals, and authority

We treat both users and devices as principals who can create policy and exercise authority granted to them. Each principal has a public-private key pair, which is consistent across devices. This approach allows multi-user devices and decisions based on the combined trustworthiness of a user and a device. (Secure initial distribution of a user's private key to her various devices is outside the scope of this paper.)

Access-control logics commonly use A signed F to describe a principal cryptographically asserting a statement F. A says F describes beliefs or assertions F that can be derived from other statements that A has signed or, using modus ponens, other statements that A believes (says):

$$\frac{A \text{ says } F \quad A \text{ says } (F \rightarrow G)}{A \text{ says } G}$$

Statements that principals can make include both delegation and use of authority. In the following example, principal A grants authority over some action F to principal B, and B wants to perform action F.

$$A \text{ signed deleg } (B, F) \quad (1)$$
$$B \text{ signed } F \quad (2)$$

These statements can be combined, as a special case of modus ponens, to prove that B's action is supported by A's authority:

$$\frac{(1) \quad (2)}{A \text{ says } F}$$

Penumbra's logic includes these rules, other constructions commonly used in access control (such as defining groups of users), and a few minor additions for describing actions on files and tags (see Section 4).

In Penumbra, the challenge statements issued by a reference monitor are of the form *device* says *action*, where *action* describes the access being attempted. For Alice to read a file on her laptop, her software agent must prove that *AliceLaptop* says readfile(f).

This design captures the intuition that a device storing some data ultimately controls who can access it: sensitive content should not be given to untrusted devices, and trusted devices are tasked with enforcing access-control policy. For most single-user devices, a default policy in which the device delegates all of its authority to its owner is appropriate. For shared devices or other less common situations, a more complex device policy that gives no user full control may be necessary.

3.4 Threat model

Penumbra is designed to prevent unauthorized access to files and tags. To prevent spoofed or forged proofs, we use nonces to prevent replay attacks and rely on standard cryptographic assumptions that signatures cannot be forged unless keys are leaked. We also rely on standard network security techniques to protect content from observation during transit between devices.

Penumbra employs a language for capturing and reasoning about trust assertions. If trust is misplaced, violations of intended policy may occur—for example, an authorized user sending a copy of a file to an unauthorized user. In contrast to other systems, Penumbra's flexibility allows users to encode limited trust precisely, minimizing vulnerability to devices or users who prove untrustworthy; for example, different devices belonging to the same owner can be trusted differently.

4 Expressing semantic policies

This section describes how Penumbra expresses and enforces semantic policies with logic-based access control.

4.1 Semantic policy for files

File accesses incur challenges of the form *device* says *action*(f), where f is a file and *action* can be one of readfile, writefile, or deletefile.

A policy by which Alice allows Bob to listen to any of her music is implemented as a conditional delegation: If Alice says a file has *type=music*, then Alice delegates to Bob authority to read that file. We write this as follows:

$$\text{Alice signed } \forall f: \text{tag(type,music,}f\text{)} \rightarrow \text{deleg(Bob,readfile(}f\text{))} \quad (3)$$

To use this delegation to listen to "Thriller," Bob's agent must show that Alice says "Thriller" has *type=music*, and that Bob intends to open "Thriller" for reading, as follows:

$$\text{Alice signed tag(type,music,"Thriller")} \quad (4)$$
$$\text{Bob signed readfile("Thriller")} \quad (5)$$

$$\frac{(3) \qquad (4)}{\text{Alice says deleg(Bob,readfile("Thriller"))} \qquad (5)}{\text{Alice says readfile("Thriller")}}$$

In this example, we assume Alice's devices grant her access to all of her files; we elide proof steps showing that the device assents once Alice does. We similarly elide instantiation of the quantified variable.

We can easily extend such policies to multiple attributes or to groups of people. To allow the group "co-workers" to view her vacation photos, Alice would assign users to the group (which is also a principal) by issuing credentials as follows:

$$\text{Alice signed speaksfor(Bob, Alice.co-workers)} \quad (6)$$

Then, Alice would delegate authority to the group rather than to individuals:

$$\text{Alice signed } \forall f: \text{tag(type,music,}f\text{)} \rightarrow \text{deleg(Alice.co-workers,readfile(}f\text{))} \quad (7)$$

4.2 Policy about tags

Penumbra supports private tags by requiring a proof of access before allowing a user or device to read a tag. Because tags are central to file and policy management, controlling access to them without impeding file system operations is critical.

Tag policy for queries. Common accesses to tags fall into three categories. A *listing query* asks which files belong to a category defined by one or more attributes, e.g., list all Alice's files with *type=movie* and *genre=comedy*. An *attribute query* asks the value of an attribute for a specific file, e.g., the name of the album to which a photo belongs. This kind of query can be made directly by users or by their software agents as part of two-stage proofs (Figure 2). A *status query*, which requests all the system metadata for a given file—last modify time, file size, etc.—is a staple of nearly every file access in most file systems (e.g., the POSIX *stat* system call).

Tag challenges have the form *device* says *action*(*attribute list*,*file*), where *action* is either readtags or deletetags. An *attribute list* is a set of (principal,attribute,value) triples representing the tags for which access is requested. Because tag queries can apply to multiple values of one attribute or multiple files, we use the wildcard * to indicate all possible completions. The listing query example above, which is a search on multiple files, would be specified with the attribute list [(Alice,type,movie), (Alice,genre,comedy)] and the target file *. The attribute query example identifies a specific target file but not a specific attribute value, and could be written with the attribute list [(Alice,album,*)] and target file "Luau.jpg." A status query for the same file would contain an attribute list like [(AliceLaptop,*,*)].

Credentials for delegating and using authority in the listing query example can be written as:

$$\text{Alice signed } \forall f: \text{deleg(Bob,readtags([(Alice,type,movie),(Alice,genre,comedy)],}f\text{))} \quad (8)$$
$$\text{Bob signed readtags([(Alice,type,movie),(Alice,genre,comedy)],*)} \quad (9)$$

These credentials can be combined to prove Bob's authority to make this query.

Implications of tag policy. One subtlety inherent in tag-based delegation is that delegations are not separable. If Alice allows Bob to list her Hawaii photos (e.g., files with *type=photo* and *album=Hawaii*), that should not imply that he can list all her photos or non-photo files related to Hawaii. However, tag delegations should be additive: a user with authority to list all photos and authority to list all Hawaii files could manually compute the intersection of the results, so a request for Hawaii photos should be allowed. Penumbra supports this subtlety.

Another interesting issue is limiting the scope of queries. Suppose Alice allows Bob to read the album name only when *album=Hawaii*, and Bob wants to know the album name for "photo127." If Bob queries the album name regardless of its value (attributelist[(Alice,album,*)]), no proof can be made and the request will fail. If Bob limits his request to the attribute list [(Alice,album,Hawaii)], the proof succeeds. If "photo127" is not in the Hawaii album, Bob cannot learn which album it is in.

Users may sometimes make broader-than-authorized queries: Bob may try to list all of Alice's photos when

he only has authority for Hawaii photos. Bob's agent will then be asked for a proof that cannot be constructed. A straightforward option is for the query to simply fail. A better outcome is for Bob to receive an abridged list containing only Hawaii photos. One way to achieve this is for Bob's agent to limit his initial request to something the agent can prove, based on available credentials—in this case, narrowing its scope from all photos to Hawaii photos. We defer implementing this to future work.

4.3 Negative policies

Negative policies, which forbid access rather than allow it, are important but often challenging for access-control systems. Without negative policies, many intuitively desirable rules are difficult to express. Examples taken from user studies include denying access to photos tagged with *weird* or *strange* [28] and sharing all files other than financial documents [34].

The first policy could naively be formulated as forbidding access to files tagged with *weird=true*; or as allowing access when the tag *weird=true* is not present. In our system, however, policies and tags are created by many principals, and there is no definitive list of all credentials. In such contexts, the inability to find a policy or tag credential does not guarantee that no such credential exists; it could simply be located somewhere else on the network. In addition, policies of this form could allow users to make unauthorized accesses by interrupting the transmission of credentials. Hence, we explore alternative ways of expressing deny policies.

Our solution has two parts. First, we allow delegation based on tag inequality: for example, to protect financial documents, Alice can allow Bob to read any file with *topic≠financial*. This allows Bob to read a file if his agent can find a tag, signed by Alice, placing that file into a topic other than financial. If no credential is found, access is still denied, which prevents unauthorized access via credential hiding. This approach works best for tags with non-overlapping values—e.g., restricting children to movies not rated R. If, however, a file is tagged with both *topic=financial* and *topic=vacation*, then this approach would still allow Bob to access the file.

To handle situations with overlapping and less-well-defined values, e.g., denying access to weird photos, Alice can grant Bob authority to view files with *type=photo* and *weird=false*. In this approach, every non-weird photo must be given the tag *weird=false*. This suggests two potential difficulties. First, we cannot ask the user to keep track of these negative tags; instead, we assume the user's policymaking interface will automatically add them (e.g., adding *weird=false* to any photo the user has not marked with *weird=true*). As we already assume the interface tracks tags to help the user maintain consistent labels and avoid typos, this is not an onerous requirement. Second, granting the ability to view files with *weird=false* implicitly leaks the potentially private information that some photos are tagged *weird=true*. We assume the policymaking interface can obfuscate such negative tags (e.g., by using a hash value to obscure *weird*), and maintain a translation to the user's original tags for purposes of updating and reviewing policy and tags. We discuss the performance impact of adding tags related to the negative policy (e.g., *weird=false*) in Section 7.

4.4 Expiration and revocation

In Penumbra, as in similar systems, the lifetime of policy is determined by the lifetimes of the credentials that encode that policy. To support dynamic policies and allow policy changes to propagate quickly, we have two fairly standard implementation choices.

One option is short credential lifetimes: the user's agent can be set to automatically renew each short-lived policy credential until directed otherwise. Alternatively, we can require all credentials used in a proof to be online countersigned, confirming validity [29]. Revocation is then accomplished by informing the countersigning authority. Both of these options can be expressed in our logic; we do not discuss them further.

5 Realistic policy examples

We discussed abstractly how policy needs can be translated into logic-based credentials. We must also ensure that our infrastructure can represent real user policies.

It is difficult to obtain real policies from users for new access-control capabilities. In lab settings, especially without experience to draw on, users struggle to articulate policies that capture real-life needs across a range of scenarios. Thus, there are no applicable standard policy or file-sharing benchmarks. Prior work has often, instead, relied on researcher experience or intuition [41,46,52,60]. Such an approach, however, has limited ability to capture the needs of non-expert users [36].

To address this, we develop the first set of access-control-policy case studies that draw from target users' needs and preferences. They are based on detailed results from in-situ and experience-sampling user studies [28, 34] and were compiled to realistically represent diverse policy needs. These case studies, which could also be used to evaluate other systems in this domain, are an important contribution of this work.

We draw on the HCI concept of *persona* development. Personas are archetypes of system users, often created to guide system design. Knowledge of these personas' characteristics and behaviors informs tests to ensure an application is usable for a range of people. Specifying

An access-control system should support ...	Sources	Case study
access-control policies on metadata	[4, 12]	All
policies for potentially overlapping groups of people, with varied granularity (e.g., family, subsets of friends, strangers, "known threats")	[4, 12, 25, 40, 44, 50]	All
policies for potentially overlapping groups of items, with varied granularity (e.g., health information, "red flag" items)	[25, 34, 40, 44]	All
photo policies based on photo location., people in photo	[4, 12, 28]	Jean, Susie
negative policies to restrict personal or embarrassing content	[4, 12, 28, 44]	Jean, Susie
policy inheritance for new and modified items	[4, 50]	All
hiding unshared content	[35, 44]	All
joint ownership of files	[34, 35]	Heather/Matt
updating policies and metadata	[4, 12, 50]	—

Table 1: Access control system needs from literature.

individuals with specific needs provides a face to types of users and focuses design and testing [62].

To make the case studies sufficiently concrete for testing, each includes a set of users and devices, as well as policy rules for at least one user. Each also includes a simulated trace of file and metadata actions; some actions loosely mimic real accesses, and others test specific properties of the access-control infrastructure. Creating this trace requires specifying many variables, including policy and access patterns, the number of files of each type, specific tags (access-control or otherwise) for each file, and users in each user group. We determine these details based on user-study data, and, where necessary, on inferences informed by HCI literature and consumer market research (e.g., [2, 57]). In general, the access-control policies are well-grounded in user-study data, while the simulated traces are more speculative.

In line with persona development [62], the case studies are intended to include a range of policy needs, especially those most commonly expressed, but not to completely cover all possible use cases. To verify coverage, we collated policy needs discussed in the literature. Table 1 presents a high-level summary. The majority of these needs are at least partially represented in all of our case studies. Unrepresented is only the ability to update policies and metadata over time, which Penumbra supports but we did not include in our test cases. The diverse policies represented by the case studies can all be encoded in Penumbra; this provides evidence that our logic is expressive enough to meet users' needs.

Case study 1: Susie. This case (Figure 3), drawn from a study of tag-based access control for photos [28], captures a default-share mentality: Susie is happy to share most photos widely, with the exception of a few containing either highly personal content or pictures of children she works with. As a result, this study exercises several somewhat-complex negative policies. This study focuses exclusively on Susie's photos, which she accesses from several personal devices but which other users access only via simulated "cloud" storage. No users besides Susie have write access or the ability to create files and tags. Because the original study collected detailed information on photo tagging and policy preferences, both the tagging and the policy are highly accurate.

Case study 2: Jean. This case study (Figure 3) is drawn from the same user study as Susie. Jean has a default-protect mentality; she only wants to share photos with people who are involved in them in some way. This includes allowing people who are tagged in photos to see those photos, as well as allowing people to see photos from events they attended, with some exceptions. Her policies include some explicit access-control tags—for example, restricting photos tagged *goofy*—as well as hybrid tags that reflect content as well as policy. As with the Susie case study, this one focuses exclusively on Jean's photos, which she accesses from personal devices and others access from a simulated "cloud." Jean's tagging scheme and policy preferences are complex; this case study includes several examples of the types of tags and policies she discussed, but is not comprehensive.

Case study 3: Heather and Matt. This case study (Figure 3) is drawn from a broader study of users' access-control needs [34]. Heather and Matt are a couple with a young daughter; most of the family's digital resources are created and managed by Heather, but Matt has full access. Their daughter has access to the subset of content appropriate for her age. The couple exemplifies a default-protect mentality, offering only limited, identified content to friends, other family members, and coworkers. This case study includes a wider variety of content, including photos, financial documents, work documents, and entertainment media. The policy preferences reflect Heather and Matt's comments; the assignment of non-access-control-related tags is less well-grounded, as they were not explicitly discussed in the interview.

Case study 4: Dana. This case study (Figure 3) is drawn from the same user study as Heather and Matt. Dana is a law student who lives with a roommate and has a strong default-protect mentality. She has confidential documents related to a law internship that must be

SUSIE

Individuals: Susie, mom
Groups: friends, acquaintances, older friends, public
Devices: laptop, phone, tablet, cloud
Tags per photo: 0-2 access-control, 1-5 other
Policies:
Friends can see all photos.
Mom can see all photos except mom-sensitive.
Acquaintances can see all photos except personal, very personal, or red flag.
Older friends can see all photos except red flag.
Public can see all photos except personal, very personal, red flag, or kids.

JEAN

Individuals: Jean, boyfriend, sister, Pat, supervisor, Dwight
Groups: volunteers, kids, acquaintances
Devices: phone, two cloud services
Tags per photo: 1-10, including mixed-use access control
Policies:
Anyone can see photos they are in.
Kids can only see kids photos.
Dwight can see photos of his wife.
Supervisor can see work photos.
Volunteers can see volunteering photos.
Boyfriend can see boyfriend, family reunion, and kids photos.
Acquaintances can see beautiful photos.
No one can see goofy photos.

HEATHER AND MATT

Individuals: Heather, Matt, daughter
Groups: friends, relatives, co-workers, guests
Devices: laptop, two phones, DVR, tablet
Tags per item: 1-3, including mixed-use access control
Policies:
Heather and Matt can see all files
Co-workers can see all photos and music
Friends and relatives can see all photos, TV shows, and music
Guests can see all TV shows and music
Daughter can see all photos; music, TV except inappropriate
Heather can update all files except TV shows
Matt can update TV shows

DANA

Individuals: Dana, sister, mom, boyfriend, roommate, boss
Groups: colleagues, friends
Devices: laptop, phone, cloud service
Tags per item: 1-3, including mixed-use access control
Policies:
Boyfriend and sister can see all photos
Friends can see favorite photos
Boyfriend, sister, friends can see all music and TV shows
Roommate can read and write household documents
Boyfriend and mom can see health documents
Boss can read and write all work documents
Colleagues can read and write work documents per project

Figure 3: Details of the four case studies

protected. This case study includes documents related to work, school, household management, and personal topics like health, as well as photos, e-books, television shows, and music. The policy preferences closely reflect Dana's comments; the non-access-control tags are drawn from her rough descriptions of the content she owns.

6 Implementation

This section describes our Penumbra prototype.

6.1 File system implementation

Penumbra is implemented in Java, on top of FUSE [1]. Users interact normally with the Linux file system; FUSE intercepts system calls related to file operations and redirects them to Penumbra. Instead of standard file paths, Penumbra expects semantic queries. For example, a command to list G-rated movies can be written 'ls "query:Alice.type=movie & Alice.rating=G".'

Figure 4 illustrates Penumbra's architecture. System calls are received from FUSE in the front-end interface, which also parses the semantic queries. The central controller invokes the reference monitor to create challenges and verify proofs, user agents to create proofs, and the file and (attribute) database managers to provide protected content. The controller uses the communications module to transfer challenges, proofs, and content between devices. We also implement a small, short-term authority cache in the controller. This allows users who

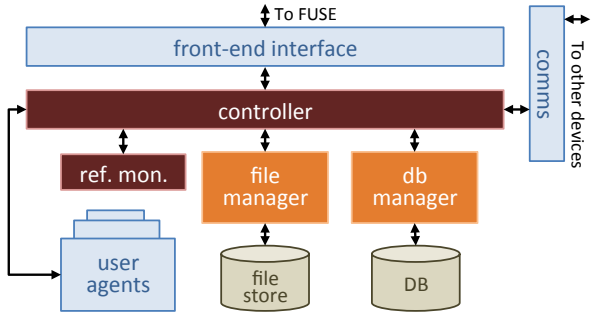


Figure 4: System architecture. The primary TCB (controller and reference monitor) is shown in red (darkest). The file and database managers (medium orange) also require some trust.

have recently proved access to content to access that content again without submitting another proof. The size and expiration time of the cache can be adjusted to trade off proving time with faster response to policy updates.

The implementation is about 15,000 lines of Java and 1800 lines of C. The primary trusted computing base (TCB) includes the controller (1800 lines) and the reference monitor (2500 lines)—the controller guards access to content, invoking the reference monitor to create challenges and verify submitted proofs. The file manager (400 lines) must be trusted to return the correct content for each file and to provide access to files only through the controller. The database manager (1600 lines) similarly must be trusted to provide access to tags only through the controller and to return only the requested

System call	Required proof(s)
mknod	create file, create metadata
open	read file, write file
truncate	write file
utime	write file
unlink	delete file
getattr	read tags: (system, *, *)
readdir	read tags: attribute list for *
getxattr	read tags: (principal, attribute, *)
setxattr	create tags
removexattr	delete tags: (principal, attribute, *)

Table 2: Proof requirements for file-related system calls

tags. The TCB also includes 145 lines of LF (logical framework) specification defining our logic.

Mapping system calls to proof goals. Table 2 shows the proof(s) required for each system call. For example, calling readdir is equivalent to a listing query—asking for all the files that have some attribute(s)—so it must incur the appropriate read-tags challenge.

Using "touch" to create a file triggers four system calls: getattr (the FUSE equivalent of stat), mknod, utime, and another getattr. Each getattr is a status query (see Section 4.2) and requires a proof of authority to read system tags. The mknod call, which creates the file and any initial metadata set by the user, requires proofs of authority to create files and metadata. Calling utime instructs the device to update its tags about the file. Updated system metadata is also a side effect of writing to a file, so we map utime to a write-file permission.

Disconnected operation. When a device is not connected to the Penumbra ensemble, its files are not available. Currently, policy updates are propagated immediately to all available devices; if a device is not available, it misses the new policy. While this is obviously impractical, it can be addressed by implementing eventual consistency (see for example Perspective [47] or Cimbiosys [43]) on top of the Penumbra architecture.

6.2 Proof generation and verification

Users' agents construct proofs using a recursive theorem prover loosely based on the one described by Elliott and Pfenning [19]. The prover starts from the goal (the challenge statement provided by the verifier) and works backward, searching through its store of credentials for one that either proves the goal directly or implies that if some additional goal(s) can be proven, the original goal will also be proven. The prover continues recursively solving these additional goals until either a solution is reached or a goal is found to be unprovable, in which case the prover backtracks and attempts to try again with another credential. When a proof is found, the prover returns it in a format that can be submitted to the reference monitor for checking. The reference monitor uses a standard LF checker implemented in Java.

The policy scenarios represented in our case studies generally result in a shallow but wide proof search: for any given proof, there are many irrelevant credentials, but only a few nested levels of additional goals. In enterprise or military contexts with strictly defined hierarchies of authority, in contrast, there may be a deeper but narrower structure. We implement some basic performance improvements for the shallow-but-wide environment, including limited indexing of credentials and simple fork-join parallelism, to allow several possible proofs to be pursued simultaneously. These simple approaches are sufficient to ensure that most proofs complete quickly; eliminating the long tail in proving time would require more sophisticated approaches, which we leave to future work.

User agents build proofs using the credentials of which they are aware. Our basic prototype pushes all delegation credentials to each user agent. (Tag credentials are guarded by the reference monitor and not automatically shared.) This is not ideal, as pushing unneeded credentials may expose sensitive information and increase proving time. However, if credentials are not distributed automatically, agents may need to ask for help from other users or devices to complete proofs (as in [9]); this could make data access slower or even impossible if devices with critical information are unreachable. Developing a strategy to distribute credentials while optimizing among these tradeoffs is left for future work.

7 Evaluation

To demonstrate that our design can work with reasonable efficiency, we evaluated Penumbra using the simulated traces we developed as part of the case studies from Section 5 as well as three microbenchmarks.

7.1 Experimental setup

We measured system call times in Penumbra using the simulated traces from our case studies. Table 3 lists features of the case studies we tested. We added users to each group, magnifying the small set of users discussed explicitly in the study interview by a factor of five. The set of files was selected as a weighted-random distribution among devices and access-control categories. For each case study, we ran a parallel control experiment with access control turned off—all access checks succeed immediately with no proving. These comparisons account for the overheads associated with FUSE, Java, and our database accesses—none of which we aggressively optimized—allowing us to focus on the overhead

Case study	Users	Files	Deleg. creds.	Proofs	System calls
Susie	60	2,349	68	46,646	212,333
Jean	65	2,500	93	30,755	264,924
Heather/Matt	60	3,098	101	39,732	266,501
Dana	60	3,798	89	27,859	74,593

Table 3: Case studies we tested. Proof and system call counts are averaged over 10 runs.

of access control. We ran each case study 10 times with and 10 times without access control.

During each automated run, each device in the case study was mounted on its own four-core (eight-thread) 3.4GHz Intel i7-4770 machine with 8GB of memory, running Ubuntu 12.04.3 LTS. The machines were connected on the same subnet via a wired Gigabit-Ethernet switch; 10 pings across each pair of machines had minimum, maximum, and median round-trip times of 0.16, 0.37, and 0.30 ms. Accounts for the people in the case study were created on each machine; these users then created the appropriate files and added a weighted-random selection of tags. Next, users listed and opened a weighted-random selection of files from those they were authorized to access. The weights are influenced by research on how the age of content affects access patterns [57]. Based on the file type, users read and wrote all or part of each file's content before closing it and choosing another to access. The specific access pattern is less important than broadly exercising the desired policy. Finally, each user attempted to access forbidden content to validate that the policy was set correctly and measure timing for failed accesses.

7.2 System call operations

Adding theorem proving to the critical path of file operations inevitably reduces performance. Usability researchers have found that delays of less than 100 ms are not noticeable to most users, who perceive times less than that as instantaneous [39]. User-visible operations consist of several combined system calls, so we target system call operation times well under the 100 ms limit.

Figure 5 shows the duration distribution for each system call, aggregated across all runs of all case studies, both with and without access control. Most system calls were well under the 100 ms limit, with medians below 2 ms for getattr, open, and utime and below 5 ms for getxattr. Medians for mknod and setxattr were 20 ms and 25 ms. That getattr is fast is particularly important, as it is called within nearly every user operation. Unfortunately, readdir (shown on its own axis for scale) did not perform as well, with a median of 66 ms. This arises from a combination of factors: readdir performs the most proofs (one local, plus one per remote device); polls each

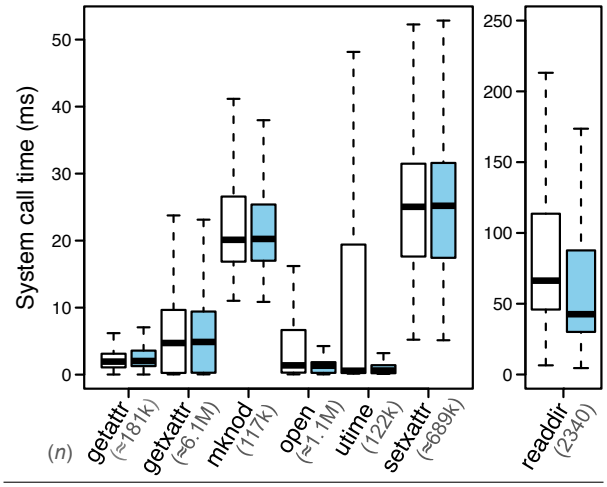


Figure 5: System call times with (white, left box of each pair) and without (shaded, right) access control, with the number of operations (n) in parentheses. ns vary up to 2% between runs with and without access control. Other than readdir (shown separately for scale), median system call times with access control are 1-25 ms and median overhead is less than 5%.

remote device; and must sometimes retrieve thousands of attributes from our mostly unoptimized database on each device. In addition, repeated readdirs are sparse in our case studies and so receive little benefit from proof caching. The results also show that access-control overhead was low across all system calls. For open and utime, the access control did not affect the median but did add more variance.

In general, we did little optimization on our simple prototype implementation; that most of our operations already fall well within the 100 ms limit is encouraging. In addition, while this performance is slower than for a typical local file system, longer delays (especially for remote operations like readdir) may be more acceptable for a distributed system targeting interactive data sharing.

7.3 Proof generation

Because proof generation is the main bottleneck inherent to our logic-based approach, it is critical to understand the factors that affect its performance. Generally system calls can incur up to four proofs (local and remote, for the proofs listed in Table 2). Most, however, incur fewer—locally opening a file for reading, for example, incurs one proof (or zero, if permission has already been cached). The exception is readdir, which can incur one local proof plus one proof for each device from which data is requested. However, if authority has already been cached no proof is required. (For these tests, authority cache entries expired after 10 minutes.)

Proving depth. Proving time is affected by prov-

ing depth, or the number of subgoals generated by the prover along one search path. Upon backtracking, proving depth decreases, then increases again as new paths are explored. Examples of steps that increase proving depth include using a delegation, identifying a member of a group, and solving the "if" clause of an implication. Although in corporate or military settings proofs can sometimes extend deeply through layers of authority, policies for personal data (as exhibited in the user studies we considered) usually do not include complex redelegation and are therefore generally shallow. In our case studies, the maximum proving depth (measured as the greatest depth reached during proof search, not the depth of the solution) was only 21; 11% of observed proofs (165,664 of 1,468,222) had depth greater than 10.

To examine the effects of proving depth, we developed a microbenchmark that tests increasingly long chains of delegation between users. We tested chains up to 60 levels deep. As shown in Figure 6a, proving time grew linearly with depth, but with a shallow slope—at 60 levels, proving time remained below 6 ms.

Red herrings. We define a *red herring* as an unsuccessful proving path in which the prover recursively pursues at least three subgoals before detecting failure and backtracking. To examine this, we developed a microbenchmark varying the number of red herrings; each red herring is exactly four levels deep. As shown in Figure 6b, proving time scaled approximately quadratically in this test: each additional red herring forces additional searches of the increasing credential space. In our case studies, the largest observed value was 43 red herrings; proofs with more than 20 red herrings made up only 0.5% of proofs (7,437 of 1,468,222). For up to 20 red herrings, proving time in the microbenchmark was generally less than 5 ms; at 40, it remained under 10 ms.

Proving time in the case studies. In the presence of real policies and metadata, changes in proving depth and red herrings can interact in complex ways that are not accounted for by the microbenchmarks. Figure 7 shows proving time aggregated in two ways. First, we compare case studies. Heather/Matt has the highest variance because files are jointly owned by the couple, adding an extra layer of indirection for many proofs. Susie has a higher median and variance than Dana or Jean because of her negative policies, which lead to more red herrings. Second, we compare proof generation times, aggregated across case studies, based on whether a proof was made by the primary user, by device agents as part of remote operations, or by other users. Most important for Penumbra is that proofs for primary users be fast, as users do not expect delays when accessing their own content; these proofs had a median time less than 0.52 ms in each case study. Also important is that device proofs are fast, as they are an extra layer of overhead on all remote operations. Device proofs had median times of 1.1-1.7 ms for each case study. Proofs for other users were slightly slower, but had medians of 2-9 ms in each case study.

We also measured the time it takes for the prover to conclude no proof can be made. Across all experiments, 1,375,259 instances of failed proofs had median and 90th-percentile times of 9 and 42 ms, respectively.

Finally, we consider the long tail of proving times. Across all 40 case study runs, the 90th-percentile proof time was 10 ms, the 99th was 45 ms, and the maximum was 1531 ms. Of 1,449,920 proofs, 3,238 (0.2%) took longer than 100 ms. These pathological cases may have several causes: high depth, bad luck in red herrings, and even Java garbage collection. Reducing the tail of proving times is an important goal for future work.

Effects of negative policy. Implementing negative policy for attributes without well-defined values (such as the allow *weird=false* example from Section 4.3) requires adding inverse policy tags to many files. A policy with negative attributes needs $n \times m$ extra attribute credentials, where n is the number of negative attributes in the policy and m is the number of affected files.

Users with default-share mentalities who tend to specify policy in terms of exceptions are most affected. Susie, our default-share case study, has five such negative attributes: *personal*, *very personal*, *mom-sensitive*, *redflag*, and *kids*. Two other case studies have one each: Jean restricts photos tagged *goofy*, while Heather and Matt restrict media files tagged *inappropriate* from their young daughter. Dana, an unusually strong example of the default-protect attitude, has none. We also reviewed detailed policy data from [28] and found that for photos, the number of negative tags ranged from 0 to 7, with median 3 and mode 1. For most study participants, negative tags fall into a few categories: synonyms for private, synonyms for weird or funny, and references to alcohol. A few also identified one or two people who prefer not to have photos of them made public. Two of 18 participants

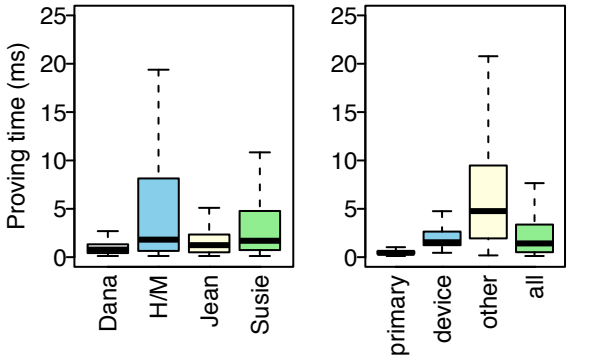


Figure 7: Proving times organized by (left) case study and (right) primary user, device, and other users.

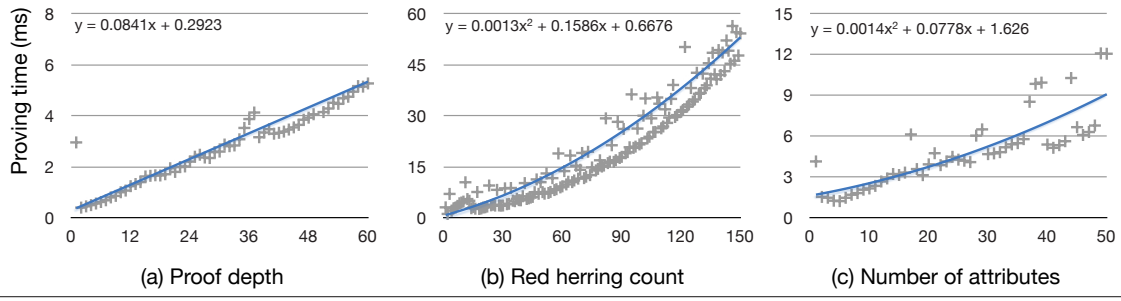


Figure 6: Three microbenchmarks showing how proving time scales with proving depth, red herrings, and attributes-per-policy. Shown with best-fit (a) line and (b,c) quadratic curve.

used a wider range of less general negative tags.

The value of m is determined in part by the complexity of the user's policy: the set of files to which the negative attributes must be attached is the set of files with the positive attributes in the same policy. For example, a policy on files with *type=photo & goofy=false* will have a larger m-value than a policy on files with *type=photo & party=true & goofy=false*.

Because attributes are indexed by file in the prover, the value of n has a much stronger affect on proving time than the value of m. Our negative-policy microbenchmark tests the prover's performance as the number of attributes per policy (and consequently per file) increases.

Figure 6c shows the results. Proving times grew approximately quadratically but with very low coefficients. For policies of up to 10 attributes (the range discussed above), proving time was less than 2.5 ms.

Adding users and devices. Penumbra was designed to support groups of users who share with each other regularly – household members, family, and close friends. Based on user studies, we estimate this is usually under 100 users. Our evaluation (Section 7) examined Penumbra's performance under these and somewhat more challenging circumstances. Adding more users and devices, however, raises some potential challenges.

When devices are added, readdir operations that must visit all devices will require more work; much of this work can be parallelized, so the latency of a readdir should grow sub-linearly in the number of devices. With more users and devices, more files are also expected, with correspondingly more total attributes. The latency of a readdir to an individual device is approximately linear in the number of attributes that are returned. Proving time should scale sub-linearly with increasing numbers of files, as attributes are indexed by file ID; increasing the number of attributes per file should scale linearly as the set of attributes for a given file is searched. Adding users can also be expected to add policy credentials. Users can be added to existing policy groups with sub-linear overhead, but more complex policy additions can have varying effects. If a new policy is mostly disjoint from old policies, it can quickly be skipped during proof search, scaling sub-linearly. However, policies that heavily overlap may lead to increases in red herrings and proof depths; interactions between these could cause proving time to increase quadratically (see Figure 6) or faster. Addressing this problem could require techniques such as pre-computing proofs or subproofs [10], as well as more aggressive indexing and parallelization within proof search to help rule out red herrings sooner.

In general, users' agents must maintain knowledge of available credentials for use in proving. Because they are cryptographically signed, credentials can be up to about 2 kB in size. Currently, these credentials are stored in memory, indexed and preprocessed in several ways, to streamline the proving process. As a result, memory requirements grow linearly, but with a large constant, as credentials are added. To support an order of magnitude more credentials would require revisiting the data structures within the users' agents and carefully considering tradeoffs among insertion time, deletion time, credential matching during proof search, and memory use.

8 Conclusion

Penumbra is a distributed file system with an access-control infrastructure for distributed personal data that combines semantic policy specification with logic-based enforcement. Using case studies grounded in data from user studies, we demonstrated that Penumbra can accommodate and enforce commonly desired policies, with reasonable efficiency. Our case studies can also be applied to other systems in this space.

9 Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. 0946825, CNS-0831407, and DGE-0903659, by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389

and W911NF-09-1-0273 from the Army Research Office, by gifts from Cisco Systems Inc. and Intel, and by Facebook and the ARCS Foundation. We thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Huawei, Intel, Microsoft Research, NEC Laboratories, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, Symantec, VMware, and Western Digital) for their interest, insights, feedback, and support. We thank Michael Stroucken and Zis Economou for help setting up testing environments.

References

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

[2] Average number of uploaded and linked photos of Facebook users as of January 2011, by gender. Statista, 2013.

[3] M. S. Ackerman. The intellectual challenge of CSCW: The gap between social requirements and technical feasibility. *Human-Computer Interaction*, 15(2):179–203, 2000.

[4] S. Ahern, D. Eckles, N. S. Good, S. King, M. Naaman, and R. Nair. Over-exposed? Privacy patterns and considerations in online and mobile photo sharing. In *Proc. ACM CHI*, 2007.

[5] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proc. ACM CCS*, 1999.

[6] Apple. Apple iCloud. https://www.icloud.com/, 2013.

[7] C.-M. Au Yeung, L. Kagal, N. Gibbins, and N. Shadbolt. Providing access control to online photo albums based on tags and linked data. In *Proc. AAAI-SSS:Social Semantic Web*, 2009.

[8] O. Ayalon and E. Toch. Retrospective privacy: Managing longitudinal privacy in online social networks. In *Proc. SOUPS*, 2013.

[9] L. Bauer, S. Garriss, and M. K. Reiter. Distributed proving in access-control systems. In *Proc. IEEE SP*, 2005.

[10] L. Bauer, S. Garriss, and M. K. Reiter. Efficient proving for practical distributed access-control systems. In *ESORICS*, 2007.

[11] L. Bauer, M. A. Schneider, and E. W. Felten. A general and flexible access-control system for the Web. In *Proc. USENIX Security*, 2002.

[12] A. Besmer and H. Richter Lipford. Moving beyond untagging: Photo privacy in a tagged world. In *Proc. ACM CHI*, 2010.

[13] A. J. Brush and K. Inkpen. Yours, mine and ours? Sharing and use of technology in domestic environments. In *Proc. UbiComp*. 2007.

[14] Facebook & your privacy: Who sees the data you share on the biggest social network? Consumer Reports Magazine, June 2012.

[15] D. Coursey. Google apologizes for Buzz privacy issues. *PCWorld*. Feb. 15, 2010.

[16] J. L. De Coi, E. Ioannou, A. Koesling, W. Nejdl, and D. Olmedilla. Access control for sharing semantic data across desktops. In *Proc. ISWC*, 2007.

[17] E. De Cristofaro, C. Soriente, G. Tsudik, and A. Williams. Hummingbird: Privacy at the time of Twitter. In *Proc. IEEE SP*, 2012.

[18] K. W. Edwards, M. W. Newman, and E. S. Poole. The infrastructure problem in HCI. In *Proc. ACM CHI*, 2010.

[19] C. Elliott and F. Pfenning. A semi-functional implementation of a higher-order logic programming language. In P. Lee, editor, *Topics in Advanced Language Implementation*. MIT Press, 1991.

[20] D. Garg and F. Pfenning. A proof-carrying file system. In *Proc. IEEE SP*, 2010.

[21] R. Geambasu, M. Balazinska, S. D. Gribble, and H. M. Levy. Homeviews: Peer-to-peer middleware for personal data sharing applications. In *Proc. ACM SIGMOD*, 2007.

[22] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole. Semantic file systems. In *Proc. ACM SOSP*, 1991.

[23] M. Hart, C. Castille, R. Johnson, and A. Stent. Usable privacy controls for blogs. In *Proc. IEEE CSE*, 2009.

[24] K. Hill. Teacher accidentally puts racy photo on students' iPad. School bizarrely suspends students. *Forbes*, October 2012.

[25] M. Johnson, S. Egelman, and S. M. Bellovin. Facebook and privacy: It's complicated. In *Proc. SOUPS*, 2012.

[26] M. Johnson, J. Karat, C.-M. Karat, and K. Grueneberg. Usable policy template authoring for iterative policy refinement. In *Proc. IEEE POLICY*, 2010.

[27] A. K. Karlson, A. J. B. Brush, and S. Schechter. Can I borrow your phone? Understanding concerns when sharing mobile phones. In *Proc. ACM CHI*, 2009.

[28] P. Klemperer, Y. Liang, M. L. Mazurek, M. Sleeper, B. Ur, L. Bauer, L. F. Cranor, N. Gupta, and M. K. Reiter. Tag, you can see it! Using tags for access control in photo sharing. In *Proc. ACM CHI*, 2012.

[29] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, 1992.

[30] C. Lesniewski-Laas, B. Ford, J. Strauss, R. Morris, and M. F. Kaashoek. Alpaca: Extensible authorization for distributed services. In *Proc. ACM CCS*, 2007.

[31] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *Proc. IEEE SP*, 2002.

[32] L. Little, E. Sillence, and P. Briggs. Ubiquitous systems and the family: Thoughts about the networked home. In *Proc. SOUPS*, 2009.

[33] A. Masoumzadeh and J. Joshi. Privacy settings in social networking systems: What you cannot control. In *Proc. ACM ASIACCS*, 2013.

[34] M. L. Mazurek, J. P. Arsenault, J. Bresee, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, R. Shay, K. Vaniea, L. Bauer, L. F. Cranor, G. R. Ganger, and M. K. Reiter. Access control for home data sharing: Attitudes, needs and practices. In *Proc. ACM CHI*, 2010.

[35] M. L. Mazurek, P. F. Klemperer, R. Shay, H. Takabi, L. Bauer, and L. F. Cranor. Exploring reactive access control. In *Proc. ACM CHI*, 2011.

[36] D. D. McCracken and R. J. Wolfe. *User-centered website development: A human-computer interaction approach*. Prentice Hall Englewood Cliffs, 2004.

[37] Microsoft. Windows SkyDrive. http://windows.microsoft.com/en-us/skydrive/, 2013.

[38] R. Needleman. How to fix Facebook's new privacy settings. *cnet*, December 2009.

[39] J. Nielsen and J. T. Hackos. *Usability engineering*, volume 125184069. Academic press Boston, 1993.

[40] J. S. Olson, J. Grudin, and E. Horvitz. A study of preferences for sharing and privacy. In *Proc. CHI EA*, 2005.

[41] D. Peek and J. Flinn. EnsemBlue: Integrating distributed storage and consumer electronics. In *Proc. OSDI*, 2006.

[42] A. Post, P. Kuznetsov, and P. Druschel. PodBase: Transparent storage management for personal devices. In *Proc. IPTPS*, 2008.

[43] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proc. NSDI*, 2009.

[44] M. N. Razavi and L. Iverson. A grounded theory of information sharing behavior in a personal learning space. In *Proc. ACM CSCW*, 2006.

[45] R. W. Reeder, L. Bauer, L. Cranor, M. K. Reiter, K. Bacon, K. How, and H. Strong. Expandable grids for visualizing and authoring computer security policies. In *Proc. ACM CHI*, 2008.

[46] O. Riva, Q. Yin, D. Juric, E. Ucan, and T. Roscoe. Policy expressivity in the Anzere personal cloud. In *Proc. ACM SOCC*, 2011.

[47] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: Semantic data management for the home. In *Proc. USENIX FAST*, 2009.

[48] S. Schroeder. Facebook privacy: 10 settings every user needs to know. *Mashable*, February 2011.

[49] M. Seltzer and N. Murphy. Hierarchical file systems are dead. In *Proc. USENIX HotOS*, 2009.

[50] D. K. Smetters and N. Good. How users use access control. In *Proc. SOUPS*, 2009.

[51] J. Staddon, P. Golle, M. Gagné, and P. Rasmussen. A content-driven access control system. In *Proc. IDTrust*, 2008.

[52] J. Strauss, J. M. Paluska, C. Lesniewski-Laas, B. Ford, R. Morris, and F. Kaashoek. Eyo: device-transparent personal storage. In *Proc. USENIX-ATC*, 2011.

[53] F. Stutzman, R. Gross, and A. Acquisti. Silent listeners: The evolution of privacy and disclosure on facebook. *Journal of Privacy and Confidentiality*, 4(2):2, 2013.

[54] K. Vaniea, L. Bauer, L. F. Cranor, and M. K. Reiter. Out of sight, out of mind: Effects of displaying access-control information near the item it controls. In *Proc. IEEE PST*, 2012.

[55] J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. *Proc. CSF*, 2008.

[56] B. Weitzenkorn. McAfee's rookie mistake gives away his location. *Scientific American*, December 2012.

[57] S. Whittaker, O. Bergman, and P. Clough. Easy on that trigger dad: a study of long term family photo retrieval. *Personal and Ubiquitous Computing*, 14(1):31–43, 2010.

[58] P. J. Wisniewski, H. Richter Lipford, and D. C. Wilson. Fighting for my space: Coping mechanisms for SNS boundary regulation. In *Proc. ACM CHI*, 2012.

[59] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. In *Proc. ACM SOSP*, 1993.

[60] T. Wobber, T. L. Rodeheffer, and D. B. Terry. Policy-based access control for weakly consistent replication. In *Proc. Eurosys*, 2010.

[61] S. Yardi and A. Bruckman. Income, race, and class: Exploring socioeconomic differences in family technology use. In *Proc. ACM CHI*, 2012.

[62] G. Zimmermann and G. Vanderheiden. Accessible design and testing in the application development process: Considerations for an integrated approach. *Universal Access in the Information Society*, 7(1-2):117–128, 2008.

On the Energy Overhead of Mobile Storage Systems

Jing Li[†] Anirudh Badam[*] Ranveer Chandra[*]
Steven Swanson[†] Bruce Worthington[§] Qi Zhang[§]

[†]UCSD [*]Microsoft Research [§]Microsoft

Abstract

Secure digital cards and embedded multimedia cards are pervasively used as secondary storage devices in portable electronics, such as smartphones and tablets. These devices cost under 70 cents per gigabyte. They deliver more than 4000 random IOPS and 70 MBps of sequential access bandwidth. Additionally, they operate at a peak power lower than 250 milliwatts. However, software storage stack above the device level on most existing mobile platforms is not optimized to exploit the low-energy characteristics of such devices. This paper examines the energy consumption of the storage stack on mobile platforms.

We conduct several experiments on mobile platforms to analyze the energy requirements of their respective storage stacks. Software storage stack consumes up to 200 times more energy when compared to storage hardware, and the security and privacy requirements of mobile apps are a major cause. A storage energy model for mobile platforms is proposed to help developers optimize the energy requirements of storage intensive applications. Finally, a few optimizations are proposed to reduce the energy consumption of storage systems on these platforms.

1 Introduction

NAND-Flash in the form of secure digital cards (SD cards) [36] and embedded multimedia cards (eMMC) [13] is the choice of storage hardware for almost all mobile phones and tablets. These storage devices consume less energy and provide significantly lower performance when compared to solid state disks (SSD). Such a trade-off is acceptable for battery-powered hand-held devices like phones and tablets, which run mostly one user-facing app at a time and therefore do not require SSD-level performance.

SD cards and eMMC devices deliver adequate performance while consuming low energy. For example, an eMMC 4.5 [35] device that we tested delivers 4000 random read, and 2000 random write 4K IOPS. Additionally, it delivers close to 70 MBps sequential read, and 40 MBps sequential write bandwidth. While the sequential bandwidth is comparable to that of a single-platter 5400 RPM magnetic disk, the random IOPS performance is an order of magnitude higher than a 15000 RPM magnetic disk. To deliver this performance, the eMMC device consumes less than 250 milliwatts (see Section 2) of peak power.

Storage software on mobile platforms, unfortunately, is not well equipped to exploit these low-energy characteristics of mobile-storage hardware. In this paper, we examine the energy cost of storage software on popular mobile platforms. The storage software consumes as much as 200 times more energy when compared to storage hardware for popular mobile platforms using Android and Windows RT. Instead of comparing performance across different platforms, this paper focuses on illustrating several fundamental hardware-independent, and platform-independent challenges with regards to the energy consumption of mobile storage systems.

We believe that most developers design their applications under the assumption that storage systems on mobile platforms are not energy-hungry. However, experimental results demonstrate the contrary. To help developers, we build a model for energy consumption of storage systems on mobile platforms. Developers can leverage such a model to optimize the energy consumption of storage-intensive mobile apps.

A detailed breakdown of the energy consumption of various storage software and hardware components was generated by analyzing data from fine-grained performance and energy profilers. This paper makes the following contributions:

1. The hardware and software energy consumption of storage systems on Android and Windows RT platforms is analyzed.

2. A model is presented that app developers can use to estimate the amount of energy consumed by storage systems and optimize their energy-efficiency accordingly.
3. Optimizations are proposed for reducing the energy consumption of mobile storage software.

The rest of this paper is organized as follows. Sections 2, 3, and 4 present an analysis of the energy consumption of storage software and hardware on Android and Windows RT systems. A model to estimate energy consumption of a given storage workload is presented in Section 5. Section 6 describes a proposal for optimizing the energy needed by mobile storage systems. Section 7 presents related work, and the conclusions from this paper are given in Section 8.

2 The Case for Storage Energy

Past studies have shown that storage is a performance bottleneck for many mobile apps [21]. This section examines the energy-overhead of storage for similar apps. In particular, background applications such as email, instant messaging, file synchronization, updates for the OS and applications, and certain operating system services like logging and bookkeeping, can be storage-intensive. This section devises estimates for the proportion of energy that these applications spend on each storage system component. Understanding the energy consumption of storage-intensive background applications can help improve the standby times of mobile devices.

Hardware power monitors are used to profile the energy consumption of real and synthetic workloads. Traces, logs and stackdumps were analyzed to understand where the energy is being spent.

2.1 Setup to Measure Energy

An Android phone and two Windows RT tablets were selected for the storage component energy consumption experiments. While these platforms provide some OS and hardware diversity for the purposes of analyses and initial conclusions, additional platforms would need to be tested in order to create truly robust power models.

2.1.1 Android Setup

The battery of a Samsung Galaxy Nexus S phone running Android version 4.2 was instrumented and connected to a Monsoon Power Monitor [26] (see

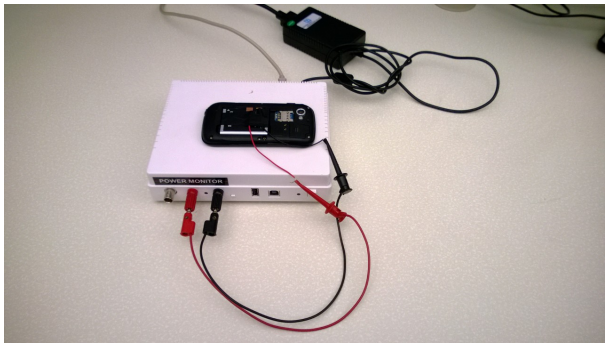


Figure 1: Android 4.2 power profiling setup: The battery leads on a Samsung Galaxy Nexus S phone were instrumented and connected to a Monsoon power monitor. The power draw of the phone was monitored using Monsoon software.

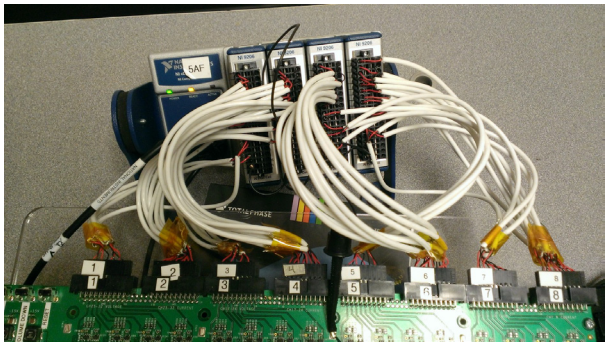


Figure 2: Windows RT 8.1 power profiling setup #1: Individual power rails were appropriately wired for monitoring by a National Instruments DAQ that captured power draws for the CPU, GPU, display, DRAM, eMMC, and other components.

Figure 3: Windows RT 8.1 power profiling setup #2: Pre-instrumented to gather fine-grained power numbers for a smaller set of power rails including the CPU, GPU, Screen, WiFi, eMMC, and DRAM.

Figure 1). In combination with Monsoon software, this meter can sample the current drawn from the battery 10's of times per second. Traces of application activity on the Android phone were captured using developer tools available for that platform [1, 2].

2.1.2 Windows RT Setup

Two Microsoft Surface RT systems were instrumented for power analysis. The first platform uses a National Instruments Digital Acquisition System (NI9206) [27] to monitor the current drawn by the CPU, GPU, display, DRAM, eMMC storage, and other components (see Figure 2). This DAQ captures 1000's of samples per second.

Figure 3 shows a second Surface RT setup, which uses a simpler DAQ chip that captures the current drawn from the CPU, memory, and other subsystems 10's of times per second. This hardware instrumentation is used in combination with the Windows Performance Toolkit [42] to concurrently profile software activity.

2.1.3 Software

Storage benchmarking tools for Android and Windows RT were built using the recommended APIs available for app-store application developers on these platforms [3, 43]. These microbenchmarks were varied using the parameters specified in Table 1. A "warm" cache is created by reading the entire contents of a file small enough to fit in DRAM at least once before the actual benchmark. A "cold" cache is created by rebooting the device before running the benchmark, and by accessing a large enough range of sectors such that few read "hits" in the DRAM are expected. The write-back experiments use a small file that is caches in DRAM in such a way that writes are lazily written to secondary storage. Such a setting enables us to estimate the energy required for writes to data that is cached. Each microbenchmark was run for one minute. The caches are always warmed from a separate process to ensure that the microbenchmarking process traverses the entire storage stack before experiencing a "hit" in the system cache.

To reduce noise, most of the applications from the systems were uninstalled, and unnecessary hardware components were disabled whenever possible (e.g., by putting the network devices into airplane mode and turning off the screen). For all the components, their idle-state power is subtracted from the power consumed during the experiment to accurately reflect only the energy used by the workload.

Parameter	Value Range
IO Size (KB)	0.5, 1, 2, 4, ..., or 1024
Read Cache Config	Warm or Cold
Write Policy	Write-through or Write-back
Access Pattern	Sequential or Random
IO Performed	Read or Write
Benchmark Language	Managed Language or Native `C`
Full-disk Encryption	Enabled or disabled

Table 1: Storage workload parameters varied between each 1-minute energy measurement.

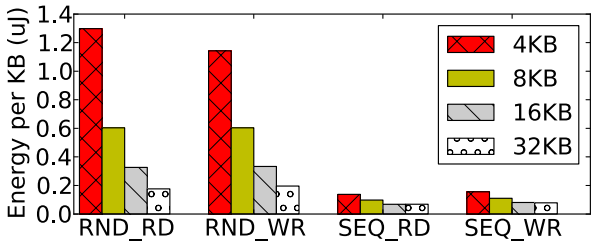


Figure 4: Storage energy per KB on Surface RT: Smaller IOs consume more energy per KB because of the per-IO cost at eMMC controller.

2.2 Experimental Results

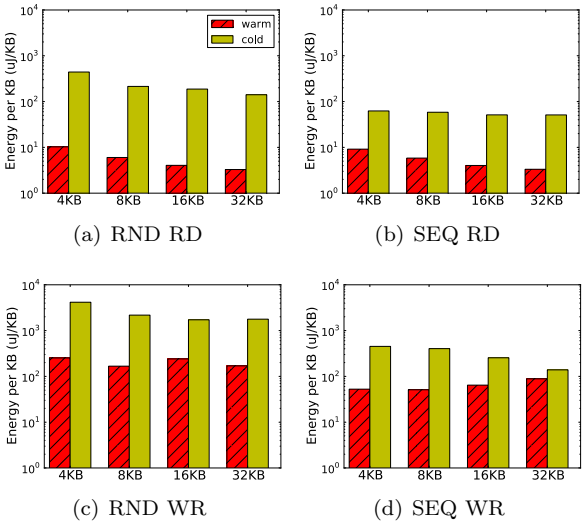
The energy overhead of the storage system was determined via microbenchmark and real application experiments. The microbenchmarks enable tightly controlled experiments, while the real application experiments provide realistic IO traces that can be replayed.

2.2.1 Microbenchmarks

Figure 4 shows the amount of energy per KB consumed by the eMMC storage for various block sizes and access patterns on the Microsoft Surface RT.

- The eMMC device requires 0.1–1.3 μJ/KB for its operations. Sequential operations are the most energy efficient from the point of view of the device.
- Random accesses of 32 KB have similar energy efficiency as sequential accesses. Smaller random accesses are more expensive – requiring more than 1 μJ/KB. This is due to the setup cost of servicing an IO at the eMMC controller level.

From a performance perspective, for a given block size, read performance is higher than write performance, and sequential IO has higher performance than random IO. We expect this to be due to the simplistic nature of eMMC controllers. Studies have shown other trends with more complex controllers [9]. For eMMC, however, the delta between read and write performance (and energy) will likely widen in the future, since eMMC devices have been increasing in read performance faster than they have been increasing in write performance.



(a) RND RD (b) SEQ RD

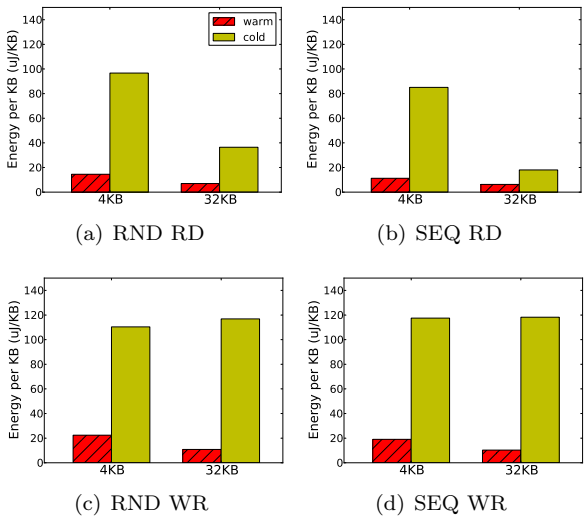
(c) RND WR (d) SEQ WR

Figure 5: System energy per KB on Android: The slower eMMC device on this platform results in more CPU and DRAM energy consumption, especially for writes. "Warm" file operations (from DRAM) are 10x more energy efficient.

Figure 5 shows that the energy per KB required by storage software on Android is two to four orders of magnitude higher than the energy consumption by the eMMC device (even though the eMMC controller in the Android platform is an older and slower generation device, the device power is in a range similar to that of the RT's eMMC device).

- Sequential reads are the most energy-efficient at the system level, requiring only one-third of the energy of random reads.
- Cold sequential reads require up to 45% more system energy than warm reads, as shown in Figure 5(b).
- Writes are one to two orders of magnitude less efficient than reads due to the additional CPU and DRAM time waiting for the writes to complete. Random writes are particularly expensive, requiring as much as 4200 μJ/KB.

The impact of low-end storage devices on performance has been well studied by Kim *et al.* [21]. Low performance, unfortunately, translates directly into high energy consumption for IO-intensive applications. We hypothesize that the idle energy consumption of CPU and DRAM (because of not entering deep idle power states soon enough) contribute to this high energy. However, we expect the energy wastage from idle power states to go down with the usage of newer and faster eMMC devices like the ones found in the tested Windows RT systems and other newer Android devices.



(a) RND RD (b) SEQ RD

(c) RND WR (d) SEQ WR

Figure 6: System energy per KB on Windows RT: The faster eMMC 4.5 card on this platform reduces the amount of idle CPU and DRAM time. "Warm" file operations (from DRAM) are 5x more energy efficient.

Figure 6 presents the energy per KB needed for the entire Windows RT platform. All "warm" IO requires less than 20 μJ/KB, whereas writes to the storage device require up to 120 μJ/KB. These energy costs are reflective of how higher performant eMMC devices can reduce energy wastage from non-sleep idle power states (tail power states). While some of this is the energy cost at the device, most of it is due to execution of the storage software, as discussed later in this section.

2.2.2 Application Benchmarks

Disk IO logs from several storage-intensive applications on Android and Windows RT were replayed to profile their energy requirements. During the replay, OS traces were captured for attributing power consumption to specific pieces of software, as well as

Email	Synchronize a mailbox with 500 emails totaling 50 MB.
File upload	Upload 100 photos totaling 80 MB to cloud storage.
File download	Download 100 photos totaling 8 0MB from cloud storage.
Music	Play local MP3 music files.
Instant messaging	Receive 100 instant messages.

Table 2: Storage-intensive background applications profiled to estimate storage software energy consumption.

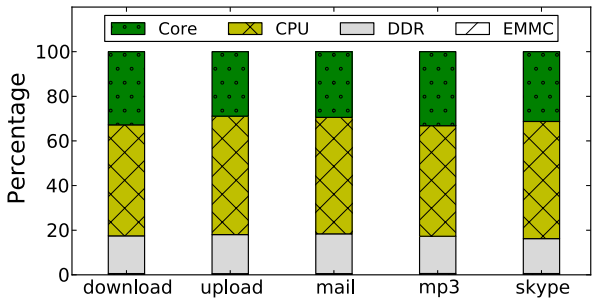


Figure 7: Breakdown of Windows RT energy consumption by hardware component. Storage software consumes more than 200x more energy than the eMMC device for background applications.

noting intervals where the CPU or DRAM were idle.

This paper focuses primarily on storage-intensive background applications that run while the screen is turned off, such as email, cloud storage uploads and downloads, local music streaming, application and OS updates, and instant messaging clients. However, many of the general observations hold true for screen-on apps as well, although display-related hardware and software tend to take up a large portion of the system energy consumption. Better understanding and optimization of the energy consumed by such applications would help increase platform standby time.

Table 2 presents the list of application scenarios profiled. Traces were taken when the device was using battery with the screen turned off.

During IO trace replay on Windows RT, power readings are captured for individual hardware components. Figure 7 plots the energy breakdown for eMMC, DRAM, CPU and Core. The "Core" power rail supplies the majority of the non-CPU compute components (GPU, encode/decode, crypto, etc.).

Library Name	% CPU Busy Time
Filesystem APIs	19.6
CLR APIs	25.8
Encryption APIs	42.1
Other APIs	12.5

Table 3: Breakdown of functionality with respect to CPU usage for a storage benchmark run on Windows RT. Overhead from managed language environment (CLR) and encryption is significant.

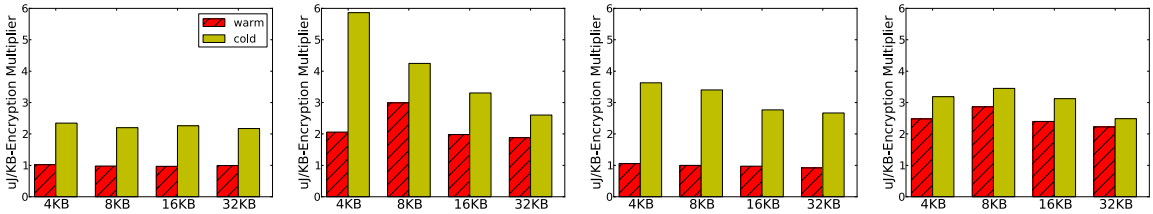
The storage software consumes between 5x and 200x more energy than the storage IO itself, depending on how the DRAM power is attributed. The fact that storage software is the primary energy consumer for storage-intensive applications is consistent with our hypothesis from the microbenchmark data. The IO traces of these applications also showed that a majority (92%) of the IO sizes were less than 64KB. We will, therefore, focus on smaller IO sizes in the rest of the paper.

Table 3 provides an overview of the stack traces collected on the Windows RT device using the Windows Performance Toolkit [42] for the mail IO workload. The majority of the CPU activity (when it was not in sleep) resulted from encryption APIs (∼42%) and Common Language Runtime (CLR) APIs (∼26%). The CLR is the virtual machine on which all the apps on Windows RT run. While there was a tail of other APIs, including filesystem APIs, contributing to CPU utilization, the largest group was associated with encryption.

The energy overhead of native filesystem APIs has been studied recently [8]. However, the overhead from disk encryption (security requirements) and the managed language environment (privacy and isolation requirements) are not well understood. Security, privacy, and isolation mechanisms are of a great importance for mobile applications. Such mechanisms not only protect sensitive user information (e.g., geographic location) from malicious applications, but they also ensure that private data cannot be retrieved from a stolen device. The following sections further examines the impact of disk encryption and managed language environments on storage systems for Windows RT and Android.

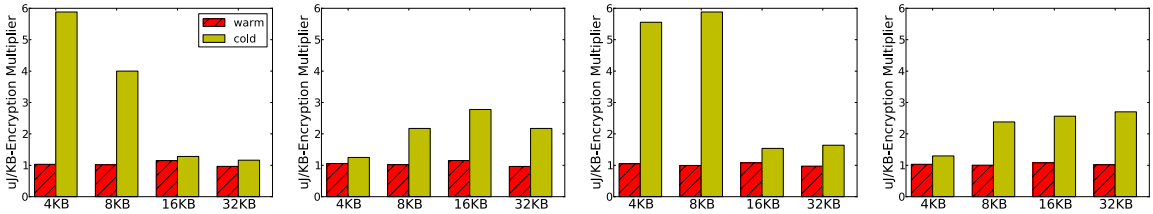
3 The Cost of Encryption

Full-disk encryption is used to protect user data from attackers with physical access to a device. Many cur-



(a) RND RD (b) RND WR (c) SEQ RD (d) SEQ WR

Figure 8: The impact of enabling encryption on the Android phone is 2.6–5.9x more energy per KB.



(a) RND RD (b) RND WR (c) SEQ RD (d) SEQ WR

Figure 9: The impact of enabling encryption on the Windows RT tablet is 1.1–5.8x more energy per KB.

rent portable devices have an option for turning on full-disk encryption to help users protect their privacy and secure their data. BitLocker [6] on Windows and similar features on Android allow users to encrypt their data. While enterprise-ready devices like Windows RT and Windows 8 tablets ship with BitLocker enabled, most Android devices ship with encryption turned off. However, most corporate Exchange and email services require full-disk encryption when they are accessed on mobile devices.

Encryption increases the energy required for all storage operations, but the cost has not been well quantified. This section presents analyses of various unencrypted and encrypted storage-intensive operations on Windows RT and Android.

Experimental Setup: Energy measurements were taken for microbenchmark workloads with variations of the first set of parameters shown in Table 1 as well as with encryption enabled and disabled while using the managed language APIs for Android, and Windows RT systems. The results are shown in Figures 8 and 9 for Android and Windows RT respectively. Each bar represents the multiplication factor by which energy consumption per KB increases when storage encryption is enabled.

"Warm" and "cold" variations are shown. As before, "warm" represents a best-case scenario where all requests are satisfied out of DRAM. "Cold" represents a worst-case scenario where all requests require storage hardware access. In all cases, except Android writes as shown in Figures 8(b) and 8(d), "warm" runs have lower energy requirements per KB.

The cost of encryption, however, still needs to be paid when cached blocks are flushed to the storage device. Section 5 presents a model to analyze the energy consumption for a given storage workload for cached and uncached IO.

Figure 8 presents the encryption energy multiplier for the Android platform:

- The energy overhead of enabling encryption ranges from 2.6x for random reads to 5.9x for random writes.
- Encryption costs per KB are almost always reduced as IO size increases, likely due to the amortization of fixed encryption start-up costs.
- Android appears to flush dirty data to the eMMC device aggressively. Even for small files that can fit entirely in memory and for experiments as short as 5 seconds, dirty data is flushed, thereby incurring at least part of the energy overhead from encryption. Therefore, Android's caching algorithms do not delay the encryption overhead as much as expected. They may also not provide as much opportunity for "over-writes" to reduce the total amount of data written, or for small sequential writes to be concatenated into more efficient large IOs.

Figure 9 presents the energy multiplier for enabling BitLocker on the Windows RT platform:

- The energy overhead of encryption ranges from 1.1x for reads to 5.8x for writes.
- The energy consumption correlation with request size is less obvious for the Windows platform. While increasing read size generally reduces energy costs because of the usage of crypto engines for larger sizes, as was the case for the Android platform, write sizes appear to have the opposite trend. All of the shown request sizes are fairly small when the CPU was used for encryption; we found that that this trend reverses as request sizes increased beyond 32 KB.
- DRAM caching does delay the energy cost of encryption for reads and writes, even for experiments as long as 60 seconds. This could provide opportunity to reduce energy because of over-writes, and also due to read prefetching at larger IO sizes and concatenation of smaller writes to form larger writes.

On Windows RT, encryption and decryption costs are highly influenced by hardware features and software algorithms used. Hardware features include the number of concurrent crypto engines, the types of encryption supported, the number of engine speeds (clock frequencies) available, the amount of local (dedicated) memory, the bandwidth to main memory, and so on. Software can choose to send all or part (or none) of the crypto work to the hardware crypto engines. For example, small crypto tasks are faster on the general purpose CPU. Using the hardware crypto engine can produce a sharp drop in energy consumption when the size of a disk IO reaches an algorithmic inflection point with regard to performance. See Section 6 for a hardware optimization we propose to bring down the energy cost of encryption for all IO sizes.

4 The Runtime Cost

Applications on mobile platforms are typically built using managed languages and run in secure containers. Mobile applications have access to sensitive user data such as geographic location, passwords, intellectual property, and financial information. Therefore, running them in isolation from the rest of the system using managed languages like Java or the Common Language Runtime (CLR) is advisable. While this eases development and makes the platform more secure, it affects both performance and energy consumption.

Any extra IO activity generated as a result of the use of managed code can significantly increase the average storage-related power, especially since mobile storage has such a low idle power envelope. This section explores the performance and energy impact of using managed code.

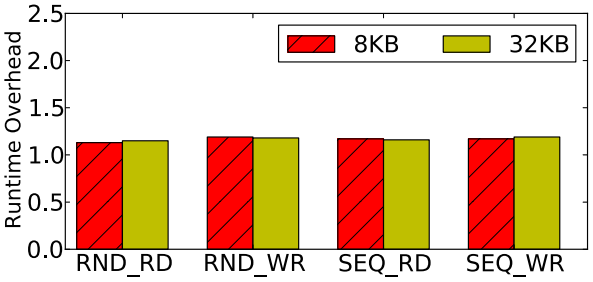


Figure 10: Impact of managed programming languages on Windows RT tablet: 13–18% more energy per KB for using the CLR.

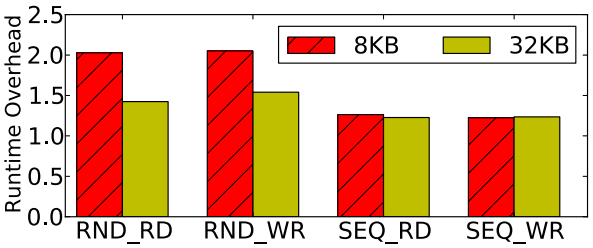
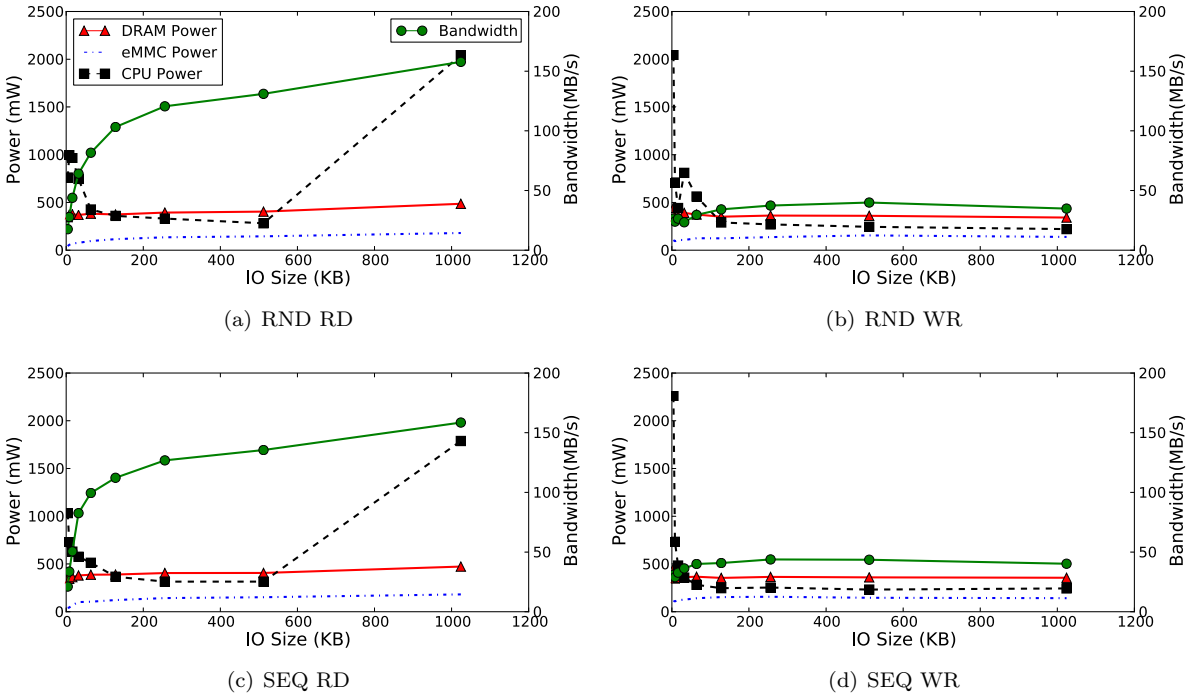


Figure 11: Impact of managed programming language on Android phone: 24–102% more energy per KB for using the Dalvik runtime.

Experimental Setup: The first set of parameters from Table 1 are again varied during a set of microbenchmarking runs using native and managed code APIs for Windows RT, and Android with encryption disabled. The pre-instrumented Windows RT tablet is specially configured (via Microsoft-internal functionality) to allow the development and running of applications natively. The native version of the benchmarking application uses the `OpenFile`, `ReadFile`, and `WriteFile` APIs on Windows. The Android version uses the Java Native Interface [20] to call the native `C fopen`, `fread`, `fseek`, and `fwrite` APIs.

The measured energy consumption for the Windows and Android platforms are shown in Figures 10, and 11, respectively. Each bar represents the multiplication factor by which energy consumption per KB increases when using managed rather than native code.

- On Windows RT, the energy overhead on storage systems from running applications in a managed environment is 12.6–18.3%.



(a) RND RD

(b) RND WR

(c) SEQ RD

(d) SEQ WR

Figure 12: Power draw by DRAM, eMMC, and CPU for different IO sizes on Windows RT with encryption disabled. CPU power draw generally decreases as the IO rate drops. However, large (e.g., 1 MB) IOs incur more CPU path (and power) because they trigger more working set trimming activity during each run.

- The overhead on Android is between 24.3–102.1%. We believe that the higher energy overhead for smaller IO sizes (some not shown) is likely due to a larger prefetching granularity used by the storage system. For larger IO sizes (some not shown), the overhead was always lower than 25%.

Security and privacy requirements of applications on mobile platforms clearly add an energy overhead as demonstrated in this section and the previous one. If developers of storage-intensive applications take these overheads into account, more energy-efficient applications could be built. See Section 6 for a hardware optimization that we propose for reducing the energy overhead due to the isolation requirements of mobile applications.

5 Energy Modeling for Storage

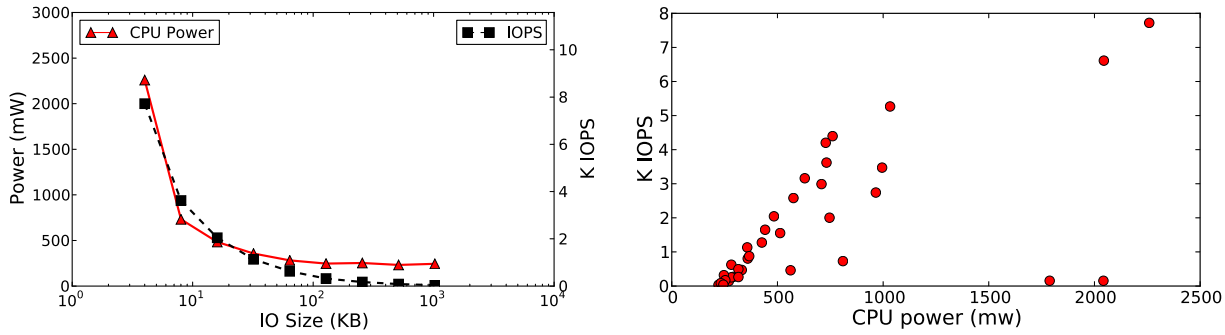
As shown in the previous sections, encryption and the use of managed code add a significant amount of overhead to the storage APIs – in terms of energy. Therefore, we believe that it is necessary to empower developers with tools to understand and optimize the energy consumed by their applications with regard to storage APIs.

This section first attempts to formalize the energy consumption characteristics of the storage subsystem. It then presents EMOS (Energy MOdeling for Storage), a simulation tool that an application or OS developer can use to estimate the amount of energy needed for their storage activity. Such a tool can be used standalone or as part of a complete energy modeling system such as WattsOn [25]. For each IO size, request type (read or write), cache behavior (hit or miss), and encryption setting (disabled or enabled), the model allows the developer to obtain an energy value.

5.1 Modeling Storage Energy

The energy cost of a given IO size and type can be broken down into its power and throughput components. If the total power of read and write operations are P_r and P_w, respectively, and the corresponding read and write throughputs are T_r and T_w KB/s, then the energy consumed by the storage device per KB for reads (E_r) and writes (E_w) is:

$$E_r = P_r/T_r, E_w = P_w/T_w$$



(a) CPU vs IOPS Correlation

(b) CPU vs IOPS Scatter plot

Figure 13: CPU power & IOps for different sizes of random and sequential reads on the Surface RT. Both metrics follow an exponential curve and show good linear correlation. The two outliers in the scatter plot towards the bottom right are caused by high read throughput triggering the CPU-intensive working set trimming process in Windows RT.

The hardware "energy" cost of accessing a storage page depends on whether it is a read or a write operation, file cache hit or miss, sequential or random, encrypted or not, and other considerations not covered by this analysis, such as request inter-arrival time, interleaving of different IO sizes and types, and the effects of storage hardware caches or device-level queuing.

In this model, P is comprised of CPU(P_{CPU}), memory (P_{DRAM}), and storage hardware(P_{EMMC}) power. Figure 12 shows the variation of each of these power components for uncached, unencrypted, random, and sequential, reads and writes via managed language microbenchmarking apps that we described in Section 2.

P_{DRAM} can be modeled as follows:

- For writes, the DRAM consumes 450 mW when the IO size is less than 8 KB. When the IO size is greater than or equal to 8 KB, this power is closer to 360 mW. This may be due to a change in memory bus speed for smaller IOs (with more IOps and higher CPU requirements driving up the memory-bus frequency).
- For reads, DRAM power increases linearly with request size from 350 mW for 4 KB reads to 475 mW for 1 MB reads. Write throughput rates are low enough that DRAM power variation for different write sizes is low. This is likely caused by more "active" power draw at the DRAM and the controller as utilization increases.

Storage unit power (P_{EMMC}) can be modeled as follows:

- For writes, the eMMC power variation due to sequentiality and request size is fairly low – from 105 mW for 4 KB IOs to 140 mW for 1 MB IOs.
- For random and sequential reads, the eMMC power varies from 40 mW for 4 KB IOs to 180 mW for 1 MB IOs, with most of the variation coming from IO sizes less than 4 KB. 4KB or less IOs are traditionally more difficult for these types of eMMC drives, because some of their internal architecture is optimized for transfers that are 8KB or larger (and aligned to corresponding logical address boundaries).

The graphs show that P_{CPU} follows an exponential curve with respect to the IO size. However, the CPU power actually tracks the storage API IOps curve, which is T/IO_size. Since IOps actually follows an exponential curve when plotted against IO size, a linear correlation exists between P_{CPU} and IOps (see Figure 13). The two scatter plot outliers that consume high CPU power at low IOps are the 1 MB sequential and random read operations. The bandwidth of these workloads (160 MB/s) was large enough and the experiments were long enough for the OS to start trimming working sets. If the other request size experiments were run for long enough, they would also incur some additional power cost when trimming finally kicks in.

With Encryption: If similar graphs were plotted for the experiments with encryption enabled, the following would be seen for the Surface RT:

- All component power values generally increase with IO size.
- P_{DRAM} is higher for reads than writes, staying fairly constant at 515 mW. For writes, the

Platform	Caching	IO Size	RND_RD	RND_WR	SEQ_RD	SEQ_WR
Windows RT	Hit	8KB	14.2	22.4	11.2	19.0
		32KB	11.4	18.2	8.6	18.2
	Miss	8KB	96.7	110.4	85.0	117.5
		32KB	36.4	116.8	18.0	118.2
Android	Hit	4KB	10.3	252.9	9.1	52.6
		8KB	6.0	167.2	5.8	51.0
		16KB	4.0	240.7	4.0	64.4
		32KB	3.3	169.7	3.3	88.5
	Miss	4KB	441.9	2402.7	62.5	451.8
		8KB	214.4	2176.7	58.5	403.5
		16KB	187.6	1720.9	51.3	254.9
		32KB	141.0	1776.0	51.1	138.8

Table 4: Energy (uJ) per KB for different IO requests. Such tables can be built for a specific platform and subsequently incorporated into power modeling software usable by developers for optimizing their storage API calls.

power increases linearly with IO size, varying from 370 mW for 4 KB IOs to 540 mW for 1 MB IOs. This variation is mostly because of the extra memory needed for encryption to complete.

- P_{EMMC} values for reads and writes are similar to their unencrypted counterparts. Given that encryption (and decryption) in current mobile devices is handled using on-SoC hardware, this is to be expected.

- P_{CPU} is fairly linear with IOPS for reads, but the power characteristics for writes are more complex. This may be due to the dynamic encryption algorithms discussed previously, where request size factors into the decision on whether to use crypto offload engines or general-purpose CPU cores to perform the encryption.

Specific measurements can change for newer hardware, however the general trends that we expect to hold are the following: P_{DRAM} would be significantly higher when encryption is enabled vs when it is disabled. This will be true as long as the hardware crypto engines do not have enough dedicated RAM. P_{EMMC} is expected to be the same whether encryption is enabled or disabled as long as the crypto engines are inside the SoC and not packaged along with the eMMC device. P_{CPU} is expected to be higher when encryption is enabled as long as the hardware crypto engines are unable to meet the throughput requirements of storage for all possible storage workloads. P_{CPU} is also expected to be correlated with the application level IOps because of software setup costs required on a per IO basis. The power trends for reads vs. writes will continue as long as eMMC controllers increase read performance at a faster pace than write performance.

5.2 The EMOS (Energy MOdeling for Storage) Simulator

The EMOS simulator takes as input a sequence of timestamped disk requests and the total size of the filesystem cache. It emulates the file caching mechanism of the operation system to identify hits and misses. Each IO is broken into small primitive operations, each of which has been empirically measured for its energy consumption.

Ideally, component power numbers (P_{CPU}, P_{DRAM}, and P_{EMMC}) would be generated for every platform. It is infeasible for a single company to take on this task, but the possibility exists for scaling out the data capture to a broader set of manufacturers. For the purposes of this paper, the EMOS simulator is tuned and tested on the Microsoft Surface RT, and Samsung Nexus S platforms.

For each platform, the average energy needed for completing a given IO type (read/write, size, cache hit/miss) is measured. The energy values are aggregated from DRAM, CPU, eMMC, and Core (idle energy values are subtracted). A table such as Table 4 can be populated to summarize the measured energy consumption required for each type of storage request. We show only a few request sizes in the

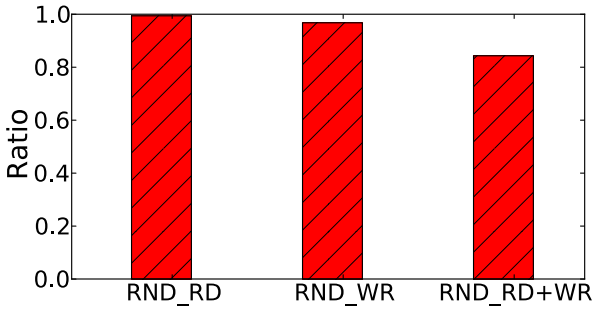


Figure 14: Experimental validation of EMOS on Android shows greater than 80% accuracy for predicting 4KB IO microbenchmark energy consumption.

table for the sake of brevity.

Simulation of cache behavior: Cache hits and misses have different storage request energy consumption. Since many factors affect the actual cache hit or miss behavior (e.g., replacement policy, cache size, prefetching algorithm, etc.), a subset of the possible cache characteristics was selected for EMOS. For example, only the LRU (Least Recently Used) cache replacement policy is simulated, but the cache size and prefetch policy are configurable.

EMOS was validated using the 4 KB random IO micro-benchmarks on the Android platform without any changes to the default cache size, or prefetch policy. The measured versus calculated energy consumption of the system were compared for workloads of 100% reads, 100% writes, and a 50%/50% mix. Figure 14 shows that while the model is accurate for pure read and write workloads, it is only 80% accurate for a mixed workload. We attribute this to the IO scheduler and the file cache software behaving differently when there is a mix of reads and writes, as well as changes in eMMC controller behavior for mixed workloads. Future investigations are planned to fully account for these behaviors.

6 Discussion: Reducing Mobile Storage Energy

We suggest ways to reduce the energy consumption of the storage stack through hardware and software modifications.

6.1 Partially-Encrypted File systems

While full-disk encryption thwarts a wide range of physical security attacks, it may be an overkill for some scenarios. It puts an unnecessary burden on accessing data that does not require encryption. For example, most OS files, application binaries, some caches, and possibly even media purchased online may not need to be encrypted. A naive solution would be to partition the disk into encrypted and unencrypted file systems / partitions. However, if free space cannot be dynamically shifted between the partitions, this solution may result in wasted disk space. More importantly, some entity has to make decisions about which files to store in which file systems, and the user would need to explicitly make some of these decisions in order to achieve optimal and appropriate partitioning. For example, a user may or may not wish his or her personal media files to be visible if a mobile device is stolen.

Partially-encrypted filesystems that allow some data to be encrypted while other data is unencrypted represent a better solution for mobile storage systems. This removes the concern over lost disk space, but some or all of the difficulties associated with the encrypt-or-not decision remain. Nevertheless, opens the option for individual applications to make some decisions about the privacy and security of files they own, perhaps splitting some files in two in order to encrypt only a portion of the data contained within. This increases development overhead, but it does provide applications with a knob to tune their energy requirements.

GNU Privacy Guard [19] for Linux and Encrypting File Systems [15] on Windows provide such services. However, care must be taken to ensure that unencrypted copies of private data not be left in the filesystem at any point unless the user is cognizant (and accepting) of this vulnerability. Additional security and privacy systems are needed to fully secure partially-encrypted file systems. Once the data from an encrypted file has been decrypted for usage, it must be actively tracked using taint analysis. Information flow control tools [14, 18, 46] are required to ensure that unencrypted copies of data are not left behind on persistent storage for attackers to exploit.

6.2 Storage Hardware Virtualization

Low-cost storage targeted to mobile platforms relies on storage software features. Isolation between applications is provided using managed languages, per-application users and groups, and virtual machines on Android and Windows RT for applications developed in Java and .NET, respectively. Storage software overhead can be reduced by moving much of this complexity into the storage hardware [8].

Mobile storage can be built in a manner such that each application is provided with the illusion of a

private filesystem. In fact, Windows RT already provides such isolation using only software [28]. Moving such isolation mechanisms into hardware can enable managed languages to directly use native APIs for applications to obtain native software like energy-usage with isolation guarantees.

6.3 SoC Offload Engines for Storage

Various components inside mobile platforms have moved their latency- and energy-intensive tasks to hardware. Audio, video, radio, and location sensors have dedicated SoC engines for frequent, narrowly-focused tasks, such as decompression, echo cancellation, and digital signal processing. This type of optimization may also be appropriate for storage. For example, the SoC can fully support encryption and improve hardware virtualization. Some SoC's already support encryption in hardware, but they do not meet the throughput expectations of applications. Crypto engines inside SoCs must be designed to match the throughput of the eMMC device at various block sizes to reduce the dependence of the OS on energy-hungry general-purpose CPU for encryption. Dedicated hardware engines for file system activity could provide metadata or data access functionality while ensuring privacy, and security.

7 Related Work

To our knowledge, a comprehensive study of storage systems on mobile platforms from the perspective of energy has not been presented to date. Kim et al [21] present a comprehensive analysis of the performance of secondary storage devices, such as SD cards often used on mobile platforms. Past research studies have presented energy analysis of other mobile subsystems, such as networking [4, 17], location sensing [41], the CPU complex [24], graphics [40], and other system components [5]. Carroll *et al.* [7] present the storage energy consumption of SD cards using native IO. Shye *et al.* [38] implement a logger to help analyze and optimize energy consumption by collecting traces of software activities.

Energy estimation and optimization tools [12, 47, 16, 25, 31, 30, 34, 33, 45] have been devised to estimate how much energy an application consumes during its execution. This paper uses similar techniques to analyze energy requirements from the perspective of the storage stack as opposed to a broader OS perspective or a narrower application perspective.

Energy consumption of storage software has been analyzed in the past for distributed systems [23], servers [32, 37, 39], PCs [29] and embedded systems [10], as opposed to the mobile platforms analyzed in this paper. Mobile storage systems are sufficiently different from these systems because of their security, privacy, and isolation requirements. This paper examines the energy overhead of these requirements.

Storage systems using new memory technologies like phase-change memory (PCM) focus on analyzing and eliminating the overhead from software [8, 11, 22, 44]. However, existing storage work for new memory technologies focuses only on native IO performance. This paper also includes analysis of managed language environments.

8 Conclusions

Battery life is a key concern for mobile devices such as phones and tablets. Although significant research has gone into improving the energy efficiency of these devices, the impact of storage (and associated APIs) on battery life has not received much attention. In part this is due to the low idle power draw of storage devices such as eMMC storage.

This paper takes a principled look at the energy consumed by storage hardware and software on mobile devices. Measurements across a set of storage-intensive microbenchmarks show that storage software may consume as much as 200x more energy than storage hardware on an Android phone and a Windows RT tablet. The two biggest energy consumers are encryption and managed language environments. Energy consumed by storage APIs increases by up to 6.0x when encryption is enabled for security. Managed language storage APIs that provide privacy, and isolation consume 25% more energy compared to their native counterparts.

We build an energy model to help developers understand the energy costs of security and privacy requirements of mobile apps. The EMOS model can predict the energy required for a mixed read/write micro-benchmark with 80% accuracy. The paper also supplies some observations on how mobile storage energy efficiency can be improved.

9 Acknowledgments

We would like to thank our shepherd, Brian Noble, as well as the anonymous FAST reviewers. We would like to thank Taofiq Ezaz, and Mohammad Jalali for helping us with the Windows RT experimental setup. We would also like to thank Lee Prewitt, and Stefan Saroiu for their valuable feedback.

References

[1] Android Application Tracing. http://developer.android.com/tools/debugging/debugging-tracing.html.

[2] Android Full System Tracing. http://developer.android.com/tools/debugging/systrace.html.

[3] Android Storage API. http://developer.android.com/guide/topics/data/data-storage.html.

[4] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *Proc. ACM IMC*, Chicago, IL, Nov. 2009.

[5] J. Bickford, H. A. Lagar-Cavilla, A. Varshavsky, V. Ganapathy, and L. Iftode. Security versus Energy Tradeoffs in Host-Based Mobile Malware Detection, June 2011.

[6] BitLocker Drive Encrytion. http://windows.microsoft.com/en-us/windows7/products/features/bitlocker.

[7] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *Proc. USENIX ATC*, Boston, MA, June 2010.

[8] A. M. Caulfield, T. I. Mollov, L. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proc. ACM ASPLOS*, London, United Kingdom, Mar. 2012.

[9] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proc. ACM SIGMETRICS*, Seattle, WA, June 2009.

[10] S. Choudhuri and R. N. Mahapatra. Energy Characterization of Filesystems for Diskless Embedded Systems. In *Proc. 41st DAC*, San Diego, CA, 2004.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proc. 22nd ACM SOSP*, Big Sky, MT, Oct. 2009.

[12] M. Dong and L. Zhong. Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems. In *Proc. 9th ACM MobiSys*, Washington, DC, June 2011.

[13] eMMC 4.51, JEDEC Standard. http://www.jedec.org/standards-documents/results/jesd84-b45.

[14] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy MOnitoring on Smartphones. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, Oct. 2010.

[15] Encrypting File System for Windows. http://technet.microsoft.com/en-us/library/cc700811.aspx.

[16] J. Flinn and M. Satyanarayanan. Energy-Aware Adaptation of Mobile Applications, Dec. 1999.

[17] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking Energy in Networked Embedded Systems. In *Proc. 8th USENIX OSDI*, San Diego, CA, Dec. 2008.

[18] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An Auditing File SYstem for Theft-Prone Devices. In *Proc. 6th ACM EUROSYS*, Salzburg, Austria, Apr. 2011.

[19] GNU Privacy Guard: Encrypt files on Linux. http://www.gnupg.org/.

[20] Java Native Interface. http://developer.android.com/training/articles/perf-jni.html.

[21] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage on Smartphones. 8(4):14:1–14:25, 2012.

[22] E. Lee, H. Bahn, and S. H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proc. 11th USENIX FAST*, San Jose, CA, Feb. 2013.

[23] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. *ACM SIGOPS OSR*, 44:61–65, 2010.

[24] A. P. Miettinen and J. K. Nurminen. Energy Efficiency of Mobile Clients in Cloud Computing. In *Proc. 2nd USENIX HotCloud*, Boston, MA, June 2010.

[25] R. Mittal, A. Kansal, and R. Chandra. Empowering Developers to Estimate App Energy Consumption. In *Proc. 18th ACM MobiCom*, Istanbul, Turkey, Aug. 2012.

[26] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.

[27] National Instruments 9206 DAQ Toolkit. http://sine.ni.com/nips/cds/view/p/lang/en/nid/209870.

[28] .NET Isolated Storage API. http://msdn.microsoft.com/en-us/

library/system.io.isolatedstorage.isolatedstoragefile.aspx.

[29] E. B. Nightingale and J. Flinn. Energy-Efficiency and Storage Flexibility in the Blue File System. In *Proc. 5th USENIX OSDI*, San Francisco, CA, Dec. 2004.

[30] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine Grained Energy Accounting on Smartphones. In *Proc. 7th ACM EUROSYS*, Bern, Switzerland, Apr. 2012.

[31] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-Grained Power Modeling for Smartphones using System Call Tracing. In *Proc. 6th ACM EUROSYS*, Salzburg, Austria, Apr. 2011.

[32] E. Pinheiro and R. Bianchini. Energy Conservation Techniques for Disk Array-Based Servers. In *Proc. 18th ACM ICS*, Saint-Malo, France, June 2004.

[33] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatschek. Profiling Resource Usage for Mobile Applications: a Cross-layer Approach. In *Proc. 9th ACM MobiSys*, Washington, DC, June 2011.

[34] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazieres, and N. Zeldovich. Energy Management in Mobile Devices with Cinder Operating System. In *Proc. 6th ACM EUROSYS*, Salzburg, Austria, Apr. 2011.

[35] Samsung eMMC 4.5 Prototype. http://www.samsung.com/us/business/oem-solutions/pdfs/eMMC_Product%20Overview.pdf.

[36] Secure Digital Card Specification. https://www.sdcard.org/downloads/pls/simplified_specs/.

[37] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating Performance and Energy in File System Server Workloads. In *Proc. USENIX ATC*, Boston, MA, June 2010.

[38] A. Shye, B. Scholbrock, and G. Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proc. 42nd IEEE MICRO*, New York, NY, Dec. 2009.

[39] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing Tape with Energy Efficient, Reliable, Disk-Based Archival Storage. In *Proc. 6th USENIX FAST*, San Jose, CA, 2008.

[40] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Signh. Who Killed My Battery: Analyzing Mobile Browser Energy Consumption. In *Proc. WWW*, Lyon, France, Apr. 2012.

[41] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh-Koniecpol. A Framework for Energy Efficient Mobile Sensing for Automatic Human State Recognition. In *Proc. 7th ACM Mobisys*, Krakow, Poland, June 2009.

[42] Windows Performance Toolkit. http://msdn.microsoft.com/en-us/performance/cc825801.aspx.

[43] Windows RT Storage API. http://msdn.microsoft.com/en-us/library/windows/apps/hh758325.aspx.

[44] X. Wu and A. L. N. Reddy. SCMFS: A File System for Storage Class Memory. In *Proc. IEEE/ACM SC*, Seattle, WA, Nov. 2011.

[45] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. AppScope: Application Energy Metering Framework for Android Smartphones using Kernel Activity Monitoring. In *Proc. USENIX ATC*, Boston, MA, June 2012.

[46] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres. Making Information Flow Explicit in HiStar. In *Proc. 7th USENIX OSDI*, Seattle, WA, Dec. 2006.

[47] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. 8th IEEE/ACM/IFIP CODES+ISSS*, Taipei, Taiwan, 2010.

ViewBox: Integrating Local File Systems with Cloud Storage Services

Yupu Zhang[†], Charlotte Dragga[†*], Andrea C. Arpaci-Dusseau[†], Remzi H. Arpaci-Dusseau[†]
[†] *University of Wisconsin-Madison,* [*] *NetApp, Inc.*

Abstract

Cloud-based file synchronization services have become enormously popular in recent years, both for their ability to synchronize files across multiple clients and for the automatic cloud backups they provide. However, despite the excellent reliability that the cloud back-end provides, the loose coupling of these services and the local file system makes synchronized data more vulnerable than users might believe. Local corruption may be propagated to the cloud, polluting all copies on other devices, and a crash or untimely shutdown may lead to inconsistency between a local file and its cloud copy. Even without these failures, these services cannot provide causal consistency.

To address these problems, we present ViewBox, an integrated synchronization service and local file system that provides freedom from data corruption and inconsistency. ViewBox detects these problems using ext4-cksum, a modified version of ext4, and recovers from them using a user-level daemon, cloud helper, to fetch correct data from the cloud. To provide a stable basis for recovery, ViewBox employs the view manager on top of ext4-cksum. The view manager creates and exposes views, consistent in-memory snapshots of the file system, which the synchronization client then uploads. Our experiments show that ViewBox detects and recovers from both corruption and inconsistency, while incurring minimal overhead.

1 Introduction

Cloud-based file synchronization services, such as Dropbox [11], SkyDrive [28], and Google Drive [13], provide a convenient means both to synchronize data across a user's devices and to back up data in the cloud. While automatic synchronization of files is a key feature of these services, the reliable cloud storage they offer is fundamental to their success. Generally, the cloud backend will checksum and replicate its data to provide integrity [3] and will retain old versions of files to offer recovery from mistakes or inadvertent deletion [11]. The robustness of these data protection features, along with the inherent replication that synchronization provides, can provide the user with a strong sense of data safety.

Unfortunately, this is merely a sense, not a reality; the loose coupling of these services and the local file system endangers data even as these services strive to protect it. Because the client has no means of determining whether file changes are intentional or the result of corruption, it may send both to the cloud, ultimately spreading corrupt data to all of a user's devices. Crashes compound this problem; the client may upload inconsistent data to the cloud, download potentially inconsistent files from the cloud, or fail to synchronize changed files. Finally, even in the absence of failure, the client cannot normally preserve causal dependencies between files, since it lacks stable point-in-time images of files as it uploads them. This can lead to an inconsistent cloud image, which may in turn lead to unexpected application behavior.

In this paper, we present ViewBox, a system that integrates the local file system with cloud-based synchronization services to solve the problems above. Instead of synchronizing individual files, ViewBox synchronizes views, in-memory snapshots of the local synchronized folder that provide data integrity, crash consistency, and causal consistency. Because the synchronization client only uploads views in their entirety, ViewBox guarantees the correctness and consistency of the cloud image, which it then uses to correctly recover from local failures. Furthermore, by making the server aware of views, ViewBox can synchronize views across clients and properly handle conflicts without losing data.

ViewBox contains three primary components. Ext4-cksum, a variant of ext4 that detects corrupt and inconsistent data through data checksumming, provides ViewBox's foundation. Atop ext4-cksum, we place the view manager, a file-system extension that creates and exposes views to the synchronization client. The view manager provides consistency through *cloud journaling* by creating views at file-system epochs and uploading views to the cloud. To reduce the overhead of maintaining views, the view manager employs *incremental snapshotting* by keeping only deltas (changed data) in memory since the last view. Finally, ViewBox handles recovery of damaged data through a user-space daemon, cloud helper, that interacts with the server-backend independently of the client.

We build ViewBox with two file synchronization services: Dropbox, a highly popular synchronization service, and Seafile, an open source synchronization service based on GIT. Through reliability experiments, we demonstrate that ViewBox detects and recovers from local data corruption, thus preventing the corruption's propagation. We also show that upon a crash, ViewBox successfully rolls back the local file system state to a previously uploaded view, restoring it to a causally consistent image. By com-

paring ViewBox to Dropbox or Seafile running atop ext4, we find that ViewBox incurs less than 5% overhead across a set of workloads. In some cases, ViewBox even improves the synchronization time by 30%.

The rest of the paper is organized as follows. We first show in Section 2 that the aforementioned problems exist through experiments and identify the root causes of those problems in the synchronization service and the local file system. Then, we present the overall architecture of ViewBox in Section 3, describe the techniques used in our prototype system in Section 4, and evaluate ViewBox in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Motivation

As discussed previously, the loosely-coupled design of cloud-based file synchronization services and file systems creates an insurmountable semantic gap that not only limits the capabilities of both systems, but leads to incorrect behavior in certain circumstances. In this section, we demonstrate the consequences of this gap, first exploring several case studies wherein synchronization services propagate file system errors and spread inconsistency. We then analyze how the limitations of file synchronization services and file systems directly cause these problems.

2.1 Synchronization Failures

We now present three case studies to show different failures caused by the semantic gap between local file systems and synchronization services. The first two of these failures, the propagation of corruption and inconsistency, result from the client's inability to distinguish between legitimate changes and failures of the file system. While these problems can be warded off by using more advanced file systems, the third, causal inconsistency, is a fundamental result of current file-system semantics.

2.1.1 Data Corruption

Data corruption is not uncommon and can result from a variety of causes, ranging from disk faults to operating system bugs [5, 8, 12, 22]. Corruption can be disastrous, and one might hope that the automatic backups that synchronization services provide would offer some protection from it. These backups, however, make them likely to propagate this corruption; as clients cannot detect corruption, they simply spread it to all of a user's copies, potentially leading to irrevocable data loss.

To investigate what might cause disk corruption to propagate to the cloud, we first inject a disk corruption to a block in a file synchronized with the cloud (by flipping bits through the device file of the underlying disk). We then manipulate the file in several different ways, and observe which modifications cause the corruption to be uploaded. We repeat this experiment for Dropbox, ownCloud, and Seafile atop ext4 (both ordered and data journaling modes) and ZFS [2] in Linux (kernel 3.6.11) and Dropbox, ownCloud, Google Drive, SugarSync, and Syncplicity atop HFS+ in Mac OS X (10.5 Lion).

FS	Service	Data write	Metadata mtime	ctime	atime
ext4 (Linux)	Dropbox	*LG*	*LG*	*LG*	*L*
	ownCloud	*LG*	*LG*	*L*	*L*
	Seafile	*LG*	*LG*	*LG*	*LG*
ZFS (Linux)	Dropbox	*L*	*L*	*L*	*L*
	ownCloud	*L*	*L*	*L*	*L*
	Seafile	*L*	*L*	*L*	*L*
HFS+ (Mac OS X)	Dropbox	*LG*	*LG*	*L*	*L*
	ownCloud	*LG*	*LG*	*L*	*L*
	GoogleDrive	*LG*	*LG*	*L*	*L*
	SugarSync	*LG*	*L*	*L*	*L*
	Syncplicity	*LG*	*LG*	*L*	*L*

Table 1: **Data Corruption Results.** *"L": corruption remains local. "G": corruption is propagated (global).*

We execute both data operations and metadata-only operations on the corrupt file. Data operations consist of both appends and in-place updates at varying distances from the corrupt block, updating both the modification and access times; these operations never overwrite the corruption. Metadata operations change only the timestamps of the file. We use *touch -a* to set the access time, *touch -m* to set the modification time, and *chown* and *chmod* to set the attribute-change time.

Table 1 displays our results for each combination of file system and service. Since ZFS is able to detect local corruption, none of the synchronization clients propagate corruption. However, on ext4 and HFS+, all clients propagate corruption to the cloud whenever they detect a change to file data and most do so when the modification time is changed, even if the file is otherwise unmodified. In both cases, clients interpret the corrupted block as a legitimate change and upload it. Seafile uploads the corruption whenever any of the timestamps change. SugarSync is the only service that does not propagate corruption when the modification time changes, doing so only once it explicitly observes a write to the file or it restarts.

2.1.2 Crash Inconsistency

The inability of synchronization services to identify legitimate changes also leads them to propagate inconsistent data after crash recovery. To demonstrate this behavior, we initialize a synchronized file on disk and in the cloud at version v_0. We then write a new version, v_1, and inject a crash which may result in an inconsistent version v_1' on disk, with mixed data from v_0 and v_1, but the metadata remains v_0. We observe the client's behavior as the system recovers. We perform this experiment with Dropbox, ownCloud, and Seafile on ZFS and ext4.

Table 2 shows our results. Running the synchroniza-

FS	Service	Upload local ver.	Download cloud ver.	OOS
ext4 (ordered)	Dropbox	√	×	√
	ownCloud	√	√	√
	Seafile	N/A	N/A	N/A
ext4 (data)	Dropbox	√	×	×
	ownCloud	√	√	×
	Seafile	√	×	×
ZFS	Dropbox	√	×	×
	ownCloud	√	√	×
	Seafile	√	×	×

Table 2: **Crash Consistency Results.** *There are three outcomes: uploading the local (possibly inconsistent) version to cloud, downloading the cloud version, and OOS (out-of-sync), in which the local version and the cloud version differ but are not synchronized. "×" means the outcome does not occur and "√" means the outcome occurs. Because in some cases the Seafile client fails to run after the crash, its results are labeled "N/A".*

tion service on top of ext4 with ordered journaling produces erratic and inconsistent behavior for both Dropbox and ownCloud. Dropbox may either upload the local, inconsistent version of the file or simply fail to synchronize it, depending on whether it had noticed and recorded the update in its internal structures before the crash. In addition to these outcomes, ownCloud may also download the version of the file stored in the cloud if it successfully synchronized the file prior to the crash. Seafile arguably exhibits the best behavior. After recovering from the crash, the client refuses to run, as it detects that its internal metadata is corrupted. Manually clearing the client's metadata and resynchronizing the folder allows the client to run again; at this point, it detects a conflict between the local file and the cloud version.

All three services behave correctly on ZFS and ext4 with data journaling. Since the local file system provides strong crash consistency, after crash recovery, the local version of the file is always consistent (either v_0 or v_1). Regardless of the version of the local file, both Dropbox and Seafile always upload the local version to the cloud when it differs from the cloud version. OwnCloud, however, will download the cloud version if the local version is v_0 and the cloud version is v_1. This behavior is correct for crash consistency, but it may violate causal consistency, as we will discuss.

2.1.3 Causal Inconsistency

The previous problems occur primarily because the file system fails to ensure a key property—either data integrity or consistency—and does not expose this failure to the file synchronization client. In contrast, causal inconsistency derives not from a specific failing on the file system's part, but from a direct consequence of traditional file system semantics. Because the client is unable to obtain a unified view of the file system at a single point in time, the client has to upload files as they change in piecemeal fashion, and the order in which it uploads files may not correspond to the order in which they were changed. Thus, file synchronization services can only guarantee eventual consistency: given time, the image stored in the cloud will match the disk image. However, if the client is interrupted—for instance, by a crash, or even a deliberate powerdown—the image stored remotely may not capture the causal ordering between writes in the file system enforced by primitives like POSIX's `sync` and `fsync`, resulting in a state that could not occur during normal operation.

To investigate this problem, we run a simple experiment in which a series of files are written to a synchronization folder in a specified order (enforced by fsync). During multiple runs, we vary the size of each file, as well as the time between file writes, and check if these files are uploaded to the cloud in the correct order. We perform this experiment with Dropbox, ownCloud, and Seafile on ext4 and ZFS, and find that for all setups, there are always cases in which the cloud state does not preserve the causal ordering of file writes.

While causal inconsistency is unlikely to directly cause data loss, it may lead to unexpected application behavior or failure. For instance, suppose the user employs a file synchronization service to store the library of a photo-editing suite that stores photos as both full images and thumbnails, using separate files for each. When the user edits a photo, and thus, the corresponding thumbnail as well, it is entirely possible that the synchronization service will upload the smaller thumbnail file first. If a fatal crash, such as a hard-drive failure, then occurs before the client can finish uploading the photo, the service will still retain the thumbnail in its cloud storage, along with the original version of the photo, and will propagate this thumbnail to the other devices linked to the account. The user, accessing one of these devices and browsing through their thumbnail gallery to determine whether their data was preserved, is likely to see the new thumbnail and assume that the file was safely backed up before the crash. The resultant mismatch will likely lead to confusion when the user fully reopens the file later.

2.2 Where Synchronization Services Fail

Our experiments demonstrate genuine problems with file synchronization services; in many cases, they not only fail to prevent corruption and inconsistency, but actively spread them. To better explain these failures, we present a brief case-study of Dropbox's local client and its interactions with the file system. While Dropbox is merely one service among many, it is well-respected and established, with a broad user-base; thus, any of its flaws are likely to be endemic to synchronization services as a whole and not merely isolated bugs.

Like many synchronization services, Dropbox actively

monitors its synchronization folder for changes using a file-system notification service, such as Linux's inotify or Mac OS X's Events API. While these services inform Dropbox of both namespace changes and changes to file content, they provide this information at a fairly coarse granularity—per file, for inotify, and per directory for the Events API, for instance. In the event that these services fail, or that Dropbox itself fails or is closed for a time, Dropbox detects changes in local files by examining their statistics, including size and modification timestamps

Once Dropbox has detected that a file has changed, it reads the file, using a combination of rsync and file chunking to determine which portions of the file have changed and transmits them accordingly [10]. If Dropbox detects that the file has changed while being read, it backs off until the file's state stabilizes, ensuring that it does not upload a partial combination of several separate writes. If it detects that multiple files have changed in close temporal proximity, it uploads the files from smallest to largest.

Throughout the entirety of the scanning and upload process, Dropbox records information about its progress and the current state of its monitored files in a local SQLite database. In the event that Dropbox is interrupted by a crash or deliberate shut-down, it can then use this private metadata to resume where it left off.

Given this behavior, the causes of Dropbox's inability to handle corruption and inconsistency become apparent. As file-system notification services provide no information on what file contents have changed, Dropbox must read files in their entirety and assume that any changes that it detects result from legitimate user action; it has no means of distinguishing unintentional changes, like corruption and inconsistent crash recovery. Inconsistent crash recovery is further complicated by Dropbox's internal metadata tracking. If the system crashes during an upload and restores the file to an inconsistent state, Dropbox will recognize that it needs to resume uploading the file, but it cannot detect that the contents are no longer consistent. Conversely, if Dropbox had finished uploading and updated its internal timestamps, but the crash recovery reverted the file's metadata to an older version, Dropbox must upload the file, since the differing timestamp could potentially indicate a legitimate change.

2.3 Where Local File Systems Fail

Responsibility for preventing corruption and inconsistency hardly rests with synchronization services alone; much of the blame can be placed on local file systems, as well. File systems frequently fail to take the preventative measures necessary to avoid these failures and, in addition, fail to expose adequate interfaces to allow synchronization services to deal with them. As summarized in Table 3, neither a traditional file system, ext4, nor a modern file system, ZFS, is able to avoid all failures.

FS	Corruption	Crash	Causal
ext4 (ordered)	×	×	×
ext4 (data)	×	√	×
ZFS	√	√	×

Table 3: **Summary of File System Capabilities.** *This table shows the synchronization failures each file system is able to handle correctly. There are three types of failures: Corruption (data corruption), Crash (crash inconsistency), and Causal (causal inconsistency). "√" means the failure does not occur and "×" means the failure may occur.*

File systems primarily prevent corruption via checksums. When writing a data or metadata item to disk, the file system stores a checksum over the item as well. Then, when it reads that item back in, it reads the checksum and uses that to validate the item's contents. While this technique correctly detects corruption, file system support for it is limited. ZFS [6] and btrfs [23] are some of the few widely available file systems that employ checksums over the whole file system; ext4 uses checksums, but only over metadata [9]. Even with checksums, however, the file system can only detect corruption, requiring other mechanisms to repair it.

Recovering from crashes without exposing inconsistency to the user is a problem that has dogged file systems since their earliest days and has been addressed with a variety of solutions. The most common of these is journaling, which provides consistency by grouping updates into transactions, which are first written to a log and then later checkpointed to their fixed location. While journaling is quite popular, seeing use in ext3 [26], ext4 [20], XFS [25], HFS+ [4], and NTFS [21], among others, writing all data to the log is often expensive, as doing so doubles all write traffic in the system. Thus, normally, these file systems only log metadata, which can lead to inconsistencies in file data upon recovery, even if the file system carefully orders its data and metadata writes (as in ext4's ordered mode, for instance). These inconsistencies, in turn, cause the erratic behavior observed in Section 2.1.2.

Crash inconsistency can be avoided entirely using copy-on-write, but, as with file-system checksums, this is an infrequently used solution. Copy-on-write never overwrites data or metadata in place; thus, if a crash occurs mid-update, the original state will still exist on disk, providing a consistent point for recovery. Implementing copy-on-write involves substantial complexity, however, and only recent file systems, like ZFS and btrfs, support it for personal use.

Finally, avoiding causal inconsistency requires access to stable views of the file system at specific points in time. File-system snapshots, such as those provided by ZFS or Linux's LVM [1], are currently the only means of obtaining such views. However, snapshot support is relatively uncommon, and when implemented, tends not to be de-

signed for the fine granularity at which synchronization services capture changes.

2.4 Summary

As our observations have shown, the sense of safety provided by synchronization services is largely illusory. The limited interface between clients and the file system, as well as the failure of many file systems to implement key features, can lead to corruption and flawed crash recovery polluting all available copies, and causal inconsistency may cause bizarre or unexpected behavior. Thus, naively assuming that these services will provide complete data protection can lead instead to data loss, especially on some of the most commonly-used file systems.

Even for file systems capable of detecting errors and preventing their propagation, such as ZFS and btrfs, the separation of synchronization services and the file system incurs an opportunity cost. Despite the presence of correct copies of data in the cloud, the file system has no means to employ them to facilitate recovery. Tighter integration between the service and the file system can remedy this, allowing the file system to automatically repair damaged files. However, this makes avoiding causal inconsistency even more important, as naive techniques, such as simply restoring the most recent version of each damaged file, are likely to directly cause it.

3 Design

To remedy the problems outlined in the previous section, we propose ViewBox, an integrated solution in which the local file system and the synchronization service cooperate to detect and recover from these issues. Instead of a clean-slate design, we structure ViewBox around ext4 (ordered journaling mode), Dropbox, and Seafile, in the hope of solving these problems with as few changes to existing systems as possible.

Ext4 provides a stable, open-source, and widely-used solution on which to base our framework. While both btrfs and ZFS already provide some of the functionality we desire, they lack the broad deployment of ext4. Additionally, as it is a journaling file system, ext4 also bears some resemblance to NTFS and HFS+, the Windows and Mac OS X file systems; thus, many of our solutions may be applicable in these domains as well.

Similarly, we employ Dropbox because of its reputation as one of the most popular, as well as one of the most robust and reliable, synchronization services. Unlike ext4, it is entirely closed source, making it impossible to modify directly. Despite this limitation, we are still able to make significant improvements to the consistency and integrity guarantees that both Dropbox and ext4 provide. However, certain functionalities are unattainable without modifying the synchronization service. Therefore, we take advantage of an open source synchronization service, Seafile, to show the capabilities that a fully integrated file system and synchronization service can provide. Although we only implement ViewBox with Dropbox and Seafile, we believe that the techniques we introduce apply more generally to other synchronization services.

In this section, we first outline the fundamental goals driving ViewBox. We then provide a high-level overview of the architecture with which we hope to achieve these goals. Our architecture performs three primary functions: detection, synchronization, and recovery; we discuss each of these in turn.

3.1 Goals

In designing ViewBox, we focus on four primary goals, based on both resolving the problems we have identified and on maintaining the features that make users appreciate file-synchronization services in the first place.

Integrity: Most importantly, ViewBox must be able to detect local corruption and prevent its propagation to the rest of the system. Users frequently depend on the synchronization service to back up and preserve their data; thus, the file system should never pass faulty data along to the cloud.

Consistency: When there is a single client, ViewBox should maintain causal consistency between the client's local file system and the cloud and prevent the synchronization service from uploading inconsistent data. Furthermore, if the synchronization service provides the necessary functionality, ViewBox must provide multi-client consistency: file-system states on multiple clients should be synchronized properly with well-defined conflict resolution.

Recoverability: While the previous properties focus on containing faults, containment is most useful if the user can subsequently repair the faults. ViewBox should be able to use the previous versions of the files on the cloud to recover automatically. At the same time, it should maintain causal consistency when necessary, ideally restoring the file system to an image that previously existed.

Performance: Improvements in data protection cannot come at the expense of performance. ViewBox must perform competitively with current solutions even when running on the low-end systems employed by many of the users of file synchronization services. Thus, naive solutions, like synchronous replication [17], are not acceptable.

3.2 Fault Detection

The ability to detect faults is essential to prevent them from propagating and, ultimately, to recover from them as well. In particular, we focus on detecting corruption and data inconsistency. While ext4 provides some ability to detect corruption through its metadata checksums, these

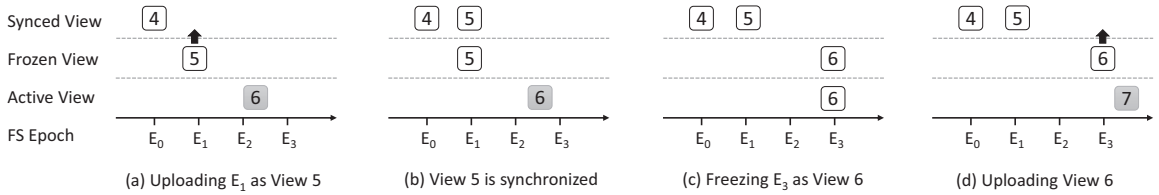


Figure 1: **Synchronizing Frozen Views.** *This figure shows how view-based synchronization works, focusing on how to upload frozen views to the cloud. The x-axis represents a series of file-system epochs. Squares represent various views in the system, with a view number as ID. A shaded active view means that the view is not at an epoch boundary and cannot be frozen.*

do not protect the data itself. Thus, to correctly detect all corruption, we add checksums to ext4's data as well, storing them separately so that we may detect misplaced writes [6, 18], as well as bit flips. Once it detects corruption, ViewBox then prevents the file from being uploaded until it can employ its recovery mechanisms.

In addition to allowing detection of corruption resulting from bit-flips or bad disk behavior, checksums also allow the file system to detect the inconsistent crash recovery that could result from ext4's journal. Because checksums are updated independently of their corresponding blocks, an inconsistently recovered data block will not match its checksum. As inconsistent recovery is semantically identical to data corruption for our purposes—both comprise unintended changes to the file system—checksums prevent the spread of inconsistent data, as well. However, they only partially address our goal of correctly restoring data, which requires stronger functionality.

3.3 View-based Synchronization

Ensuring that recovery proceeds correctly requires us to eliminate causal inconsistency from the synchronization service. Doing so is not a simple task, however. It requires the client to have an isolated view of all data that has changed since the last synchronization; otherwise, user activity could cause the remote image to span several file system images but reflect none of them.

While file-system snapshots provide consistent, static images [16], they are too heavyweight for our purposes. Because the synchronization service stores all file data remotely, there is no reason to persist a snapshot on disk. Instead, we propose a system of in-memory, ephemeral snapshots, or *views*.

3.3.1 View Basics

Views represent the state of the file system at specific points in time, or epochs, associated with quiescent points in the file system. We distinguish between three types of views: active views, frozen views, and synchronized views. The active view represents the current state of the local file system as the user modifies it. Periodically, the file system takes a snapshot of the active view; this becomes the current frozen view. Once a frozen view is uploaded to the cloud, it then becomes a synchronized view, and can be used for restoration. At any time, there is only one active view and one frozen view in the local system, while there are multiple synchronized views on the cloud.

To provide an example of how views work in practice, Figure 1 depicts the state of a typical ViewBox system. In the initial state, (a), the system has one synchronized view in the cloud, representing the file system state at epoch 0, and is in the process of uploading the current frozen view, which contains the state at epoch 1. While this occurs, the user can make changes to the active view, which is currently in the middle of epoch 2 and epoch 3.

Once ViewBox has completely uploaded the frozen view to the cloud, it becomes a synchronized view, as shown in (b). ViewBox refrains from creating a new frozen view until the active view arrives at an epoch boundary, such as a journal commit, as shown in (c). At this point, it discards the previous frozen view and creates a new one from the active view, at epoch 3. Finally, as seen in (d), ViewBox begins uploading the new frozen view, beginning the cycle anew.

Because frozen views are created at file-system epochs and the state of frozen views is always static, synchronizing frozen views to the cloud provides both crash consistency and causal consistency, given that there is only one client actively synchronizing with the cloud. We call this *single-client consistency*.

3.3.2 Multi-client Consistency

When multiple clients are synchronized with the cloud, the server must propagate the latest synchronized view from one client to other clients, to make all clients' state synchronized. Critically, the server must propagate views in their entirety; partially uploaded views are inherently inconsistent and thus should not be visible. However, because synchronized views necessarily lag behind the active views in each file system, the current active file system may have dependencies that would be invalidated by a remote synchronized view. Thus, remote changes must be applied to the active view in a way that preserves local causal consistency.

To achieve this, ViewBox handles remote changes in two phases. In the first phase, ViewBox applies remote changes to the frozen view. If a changed file does not exist in the frozen view, ViewBox adds it directly; otherwise, it adds the file under a new name that indicates a conflict (e.g., "foo.txt" becomes "remote.foo.txt"). In the second

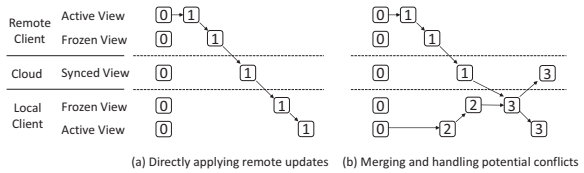


Figure 2: **Handling Remote Updates.** *This figure demonstrates two different scenarios where remote updates are handled. While case (a) has no conflicts, case (b) may, because it contains concurrent updates.*

phase, ViewBox merges the newly created frozen view with the active view. ViewBox propagates all changes from the new frozen view to the active view, using the same conflict handling procedure. At the same time, it uploads the newly merged frozen view. Once the second phase completes, the active view is fully updated; only after this occurs can it be frozen and uploaded.

To correctly handle conflicts and ensure no data is lost, we follow the same policy as GIT [14]. This can be summarized by the following three guidelines:

- Preserve any local or remote change; a change could be the addition, modification, or deletion of a file.
- When there is a conflict between a local change and a remote change, always keep the local copy untouched, but rename and save the remote copy.
- Synchronize and propagate both the local copy and the renamed remote copy.

Figure 2 illustrates how ViewBox handles remote changes. In case (a), both the remote and local clients are synchronized with the cloud, at view 0. The remote client makes changes to the active view, and subsequently freezes and uploads it to the cloud as view 1. The local client is then informed of view 1, and downloads it. Since there are no local updates, the client directly applies the changes in view 1 to its frozen view and propagates those changes to the active view.

In case (b), both the local client and the remote client perform updates concurrently, so conflicts may exist. Assuming the remote client synchronizes view 1 to the cloud first, the local client will refrain from uploading its frozen view, view 2, and download view 1 first. It then merges the two views, resolving conflicts as described above, to create a new frozen view, view 3. Finally, the local client uploads view 3 while simultaneously propagating the changes in view 3 to the active view.

In the presence of simultaneous updates, as seen in case (b), this synchronization procedure results in a cloud state that reflects a combination of the disk states of all clients, rather than the state of any one client. Eventually, the different client and cloud states will converge, providing *multi-client consistency*. This model is weaker than our single-client model; thus, ViewBox may not be able to provide causal consistency for each individual client under all circumstances.

Unlike single-client consistency, multi-client consistency requires the cloud server to be aware of views, not just the client. Thus, ViewBox can only provide multi-client consistency for open source services, like Seafile; providing it for closed-source services, like Dropbox, will require explicit cooperation from the service provider.

3.4 Cloud-aided Recovery

With the ability to detect faults and to upload consistent views of the file system state, ViewBox is now capable of performing correct recovery. There are effectively two types of recovery to handle: recovery of corrupt files, and recovery of inconsistent files at the time of a crash.

In the event of corruption, if the file is clean in both the active view and the frozen view, we can simply recover the corrupt block by fetching the copy from the cloud. If the file is dirty, the file may not have been synchronized to the cloud, making direct recovery impossible, as the block fetched from cloud will not match the checksum. If recovering a single block is not possible, the entire file must be rolled back to a previous synchronized version, which may lead to causal inconsistency.

Recovering causally-consistent images of files that were present in the active view at the time of a crash faces the same difficulties as restoring corrupt files in the active view. Restoring each individual file to its most recent synchronized version is not correct, as other files may have been written after the now-corrupted file and, thus, depend on it; to ensure these dependencies are not broken, these files also need to be reverted. Thus, naive restoration can lead to causal inconsistency, even with views.

Instead, we present users with the choice of individually rolling back damaged files, potentially risking causal inconsistency, or reverting to the most recent synchronized view, ensuring correctness but risking data loss. As we anticipate that the detrimental effects of causal inconsistency will be relatively rare, the former option will be usable in many cases to recover, with the latter available in the event of bizarre or unexpected application behavior.

4 Implementation

Now that we have provided a broad overview of ViewBox's architecture, we delve more deeply into the specifics of our implementation. As with Section 3, we divide our discussion based on the three primary components of our architecture: detection, as implemented with our new *ext4-cksum* file system; view-based synchronization using our *view manager*, a file-system agnostic extension to ext4-cksum; and recovery, using a user-space recovery daemon called *cloud helper*.

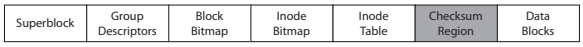


Figure 3: **Ext4-cksum Disk Layout.** *This graph shows the layout of a block group in ext4-cksum. The shaded checksum region contains data checksums for blocks in the block group.*

4.1 Ext4-cksum

Like most file systems that update data in place, ext4 provides minimal facilities for detecting corruption and ensuring data consistency. While it offers experimental metadata checksums, these do not protect data; similarly, its default ordered journaling mode only protects the consistency of metadata, while providing minimal guarantees about data. Thus, it requires changes to meet our requirements for integrity and consistency. We now present ext4-cksum, a variant of ext4 that supports data checksums to protect against data corruption and to detect data inconsistency after a crash without the high cost of data journaling.

4.1.1 Checksum Region

Ext4-cksum stores data checksums in a fixed-sized *checksum region* immediately after the inode table in each block group, as shown in Figure 3. All checksums of data blocks in a block group are preallocated in the checksum region. This region acts similarly to a bitmap, except that it stores checksums instead of bits, with each checksum mapping directly to a data block in the group. Since the region starts at a fixed location in a block group, the location of the corresponding checksum can be easily calculated, given the physical (disk) block number of a data block.

The size of the region depends solely on the total number of blocks in a block group and the length of a checksum, both of which are determined and fixed during file system creation. Currently, ext4-cksum uses the built-in crc32c checksum, which is 32 bits. Therefore, it reserves a 32-bit checksum for every 4KB block, imposing a space overhead of 1/1024; for a regular 128MB block group, the size of the checksum region is 128KB.

4.1.2 Checksum Handling for Reads and Writes

When a data block is read from disk, the corresponding checksum must be verified. Before the file system issues a read of a data block from disk, it gets the corresponding checksum by reading the checksum block. After the file system reads the data block into memory, it verifies the block against the checksum. If the initial verification fails, ext4-cksum will retry. If the retry also fails, ext4-cksum will report an error to the application. Note that in this case, if ext4-cksum is running with the cloud helper daemon, ext4-cksum will try to get the remote copy from cloud and use that for recovery. The read part of a read-modify-write is handled in the same way.

A read of a data block from disk always incurs an additional read for the checksum, but not every checksum read will cause high latency. First, the checksum read can be served from the page cache, because the checksum blocks are considered metadata blocks by ext4-cksum and are kept in the page cache like other metadata structures. Second, even if the checksum read does incur a disk I/O, because the checksum is always in the same block group as the data block, the seek latency will be minimal. Third, to avoid checksum reads as much as possible, ext4-cksum employs a simple prefetching policy: always read 8 checksum blocks (within a block group) at a time. Advanced prefetching heuristics, such as those used for data prefetching, are applicable here.

Ext4-cksum does not update the checksum for a dirty data block until the data block is written back to disk. Before issuing the disk write for the data block, ext4-cksum reads in the checksum block and updates the corresponding checksum. This applies to all data write-backs, caused by a background flush, fsync, or a journal commit.

Since ext4-cksum treats checksum blocks as metadata blocks, with journaling enabled, ext4-cksum logs all dirty checksum blocks in the journal. In ordered journaling mode, this also allows the checksum to detect inconsistent data caused by a crash. In ordered mode, dirty data blocks are flushed to disk before metadata blocks are logged in the journal. If a crash occurs before the transaction commits, data blocks that have been flushed to disk may become inconsistent, because the metadata that points to them still remains unchanged after recovery. As the checksum blocks are metadata, they will not have been updated, causing a mismatch with the inconsistent data block. Therefore, if such a block is later read from disk, ext4-cksum will detect the checksum mismatch.

To ensure consistency between a dirty data block and its checksum, data write-backs triggered by a background flush and fsync can no longer simultaneously occur with a journal commit. In ext4 with ordered journaling, before a transaction has committed, data write-backs may start and overwrite a data block that was just written by the committing transaction. This behavior, if allowed in ext4-cksum, would cause a mismatch between the already logged checksum block and the newly written data block on disk, thus making the committing transaction inconsistent. To avoid this scenario, ext4-cksum ensures that data write-backs due to a background flush and fsync always occur before or after a journal commit.

4.2 View Manager

To provide consistency, ViewBox requires file synchronization services to upload frozen views of the local file system, which it implements through an in-memory file-system extension, the view manager. In this section, we detail the implementation of the view manager, beginning with an overview. Next, we introduce two techniques, cloud journaling and incremental snapshotting, which are key to the consistency and performance provided by the view manager. Then, we provide an example that de-

scribes the synchronization process that uploads a frozen view to the cloud. Finally, we briefly discuss how to integrate the synchronization client with the view manager to handle remote changes and conflicts.

4.2.1 View Manager Overview

The view manager is a light-weight kernel module that creates views on top of a local file system. Since it only needs to maintain two local views at any time (one frozen view and one active view), the view manager does not modify the disk layout or data structures of the underlying file system. Instead, it relies on a modified tmpfs to present the frozen view in memory and support all the basic file system operations to files and directories in it. Therefore, a synchronization client now monitors the exposed frozen view (rather than the actual folder in the local file system) and uploads changes from the frozen view to the cloud. All regular file system operations from other applications are still directly handled by ext4-cksum. The view manager uses the active view to track the on-going changes and then reflects them to the frozen view. Note that the current implementation of the view manager is tailored to our ext4-cksum and it is not stackable [29]. We believe that a stackable implementation would make our view manager compatible with more file systems.

4.2.2 Consistency through Cloud Journaling

As we discussed in Section 3.3.1, to preserve consistency, frozen views must be created at file-system epochs. Therefore, the view manager freezes the current active view at the beginning of a journal commit in ext4-cksum, which serves as a boundary between two file-system epochs. At the beginning of a commit, the current running transaction becomes the committing transaction. When a new running transaction is created, all operations belonging to the old running transaction will have completed, and operations belonging to the new running transaction will not have started yet. The view manager freezes the active view at this point, ensuring that no in-flight operation spans multiple views. All changes since the last frozen view are preserved in the new frozen view, which is then uploaded to the cloud, becoming the latest synchronized view.

To ext4-cksum, the cloud acts as an external journaling device. Every synchronized view on the cloud matches a consistent state of the local file system at a specific point in time. Although ext4-cksum still runs in ordered journaling mode, when a crash occurs, the file system now has the chance to roll back to a consistent state stored on cloud. We call this approach cloud journaling.

4.2.3 Low-overhead via Incremental Snapshotting

During cloud journaling, the view manager achieves better performance and lower overhead through a technique called incremental snapshotting. The view manager always keeps the frozen view in memory and the frozen view only contains the data that changed from the previous view. The active view is thus responsible for tracking all the files and directories that have changed since it last was frozen. When the view manager creates a new frozen view, it marks all changed files copy-on-write, which preserves the data at that point. The new frozen view is then constructed by applying the changes associated with the active view to the previous frozen view.

The view manager uses several in-memory and on-cloud structures to support this incremental snapshotting approach. First, the view manager maintains an *inode mapping table* to connect files and directories in the frozen view to their corresponding ones in the active view. The view manager represents the namespace of a frozen view by creating *frozen inodes* for files and directories in tmpfs (their counterparts in the active view are thus called *active inodes*), but no data is usually stored under frozen inodes (unless the data is copied over from the active view due to copy-on-write). When a file in the frozen view is read, the view manager finds the active inode and fetches data blocks from it. The inode mapping table thus serves as a translator between a frozen inode and its active inode.

Second, the view manager tracks namespace changes in the active view by using an *operation log*, which records all successful namespace operations (e.g., create, mkdir, unlink, rmdir, and rename) in the active view. When the active view is frozen, the log is replayed onto the previous frozen view to bring it up-to-date, reflecting the new state.

Third, the view manager uses a *dirty table* to track what files and directories are modified in the active view. Once the active view becomes frozen, all these files are marked copy-on-write. Then, by generating inotify events based on the operation log and the dirty table, the view manager is able to make the synchronization client check and upload these local changes to the cloud.

Finally, the view manager keeps *view metadata* on the server for every synchronized view, which is used to identify what files and directories are contained in a synchronized view. For services such as Seafile, which internally keeps the modification history of a folder as a series of snapshots [24], the view manager is able to use its snapshot ID (called commit ID by Seafile) as the view metadata. For services like Dropbox, which only provides file-level versioning, the view manager creates a view metadata file for every synchronized view, consisting of a list of pathnames and revision numbers of files in that view. The information is obtained by querying the Dropbox server. The view manager stores these metadata files in a hidden folder on the cloud, so the correctness of these files is not affected by disk corruption or crashes.

4.2.4 Uploading Views to the Cloud

Now, we walk through an example in Figure 4 to explain how the view manager uploads views to the cloud. In the

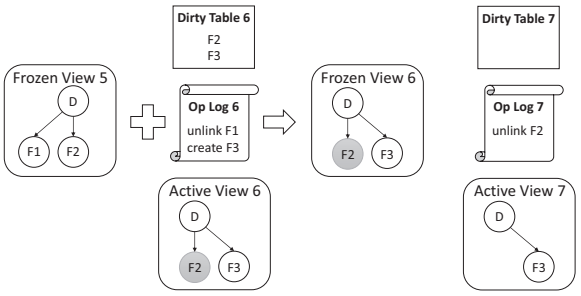


Figure 4: **Incremental Snapshotting.** *This figure illustrates how the view manager creates active and frozen views.*

example, the synchronization service is Dropbox.

Initially, the synchronization folder (D) contains two files (F1 and F2). While frozen view 5 is being synchronized, in active view 6, F1 is deleted, F2 is modified, and F3 is created. The view manager records the two namespace operations (unlink and create) in the operation log, and adds F2 and F3 to the dirty table. When frozen view 5 is completely uploaded to the cloud, the view manager creates a view metadata file and uploads it to the server.

Next, the view manager waits for the next journal commit and freezes active view 6. The view manager first marks F2 and F3 in the dirty table copy-on-write, preserving new updates in the frozen view. Then, it creates active view 7 with a new operation log and a new dirty table, allowing the file system to operate without any interruption. After that, the view manager replays the operation log onto frozen view 5 such that the namespace reflects the state of frozen view 6.

Finally, the view manager generates inotify events based on the dirty table and the operation log, thus causing the Dropbox client to synchronize the changes to the cloud. Since F3 is not changed in active view 7, the client reading its data from the frozen view would cause the view manager to consult the inode mapping table (not shown in the figure) and fetch requested data directly from the active view. Note that F2 is deleted in active view 7. If the deletion occurs before the Dropbox client is able to upload F2, all data blocks of F2 are copied over and attached to the copy of F2 in the frozen view. If Dropbox reads the file before deletion occurs, the view manager fetches those blocks from active view 7 directly, without making extra copies. After frozen view 6 is synchronized to the cloud, the view manager repeats the steps above, constantly uploading views from the local system.

4.2.5 Handling Remote Changes

All the techniques we have introduced so far focus on how to provide single-client consistency and do not require modifications to the synchronization client or the server. They work well with proprietary synchronization services such as Dropbox. However, when there are multiple clients running ViewBox and performing updates at the same time, the synchronization service itself must be view-aware. To handle remote updates correctly, we modify the Seafile client to perform the two-phase synchronization described in Section 3.3.2. We choose Seafile to implement multi-client consistency, because both its client and server are open-source. More importantly, its data model and synchronization algorithm is similar to GIT, which fits our view-based synchronization well.

4.3 Cloud Helper

When corruption or a crash occurs, ViewBox performs recovery using backup data on the cloud. Recovery is performed through a user-level daemon, cloud helper. The daemon is implemented in Python, which acts as a bridge between the local file system and the cloud. It interacts with the local file system using ioctl calls and communicates with the cloud through the service's web API.

For data corruption, when ext4-cksum detects a checksum mismatch, it sends a block recovery request to the cloud helper. The request includes the pathname of the corrupted file, the offset of the block inside the file, and the block size. The cloud helper then fetches the requested block from the server and returns the block to ext4-cksum. Ext4-cksum reverifies the integrity of the block against the data checksum in the file system and returns the block to the application. If the verification still fails, it is possibly because the block has not been synchronized or because the block is fetched from a different file in the synchronized view on the server with the same pathname as the corrupted file.

When a crash occurs, the cloud helper performs a scan of the ext4-cksum file system to find potentially inconsistent files. If the user chooses to only roll back those inconsistent files, the cloud helper will download them from the latest synchronized view. If the user chooses to roll back the whole file system, the cloud helper will identify the latest synchronized view on the server, and download files and construct directories in the view. The former approach is able to keep most of the latest data but may cause causal inconsistency. The latter guarantees causal consistency, but at the cost of losing updates that took place during the frozen view and the active view when the crash occurred.

5 Evaluation

We now present the evaluation results of our ViewBox prototype. We first show that our system is able to recover from data corruption and crashes correctly and provide causal consistency. Then, we evaluate the underlying ext4-cksum and view manager components separately, without synchronization services. Finally we study the overall synchronization performance of ViewBox with Dropbox and Seafile.

We implemented ViewBox in the Linux 3.6.11 kernel, with Dropbox client 1.6.0, and Seafile client and server

Service ViewBox w/	Data write	Metadata mtime	ctime	atime
Dropbox	*DR*	*DR*	*DR*	*DR*
Seafile	*DR*	*DR*	*DR*	*DR*

Table 4: **Data Corruption Results of ViewBox.** *In all cases, the local corruption is detected (D) and recovered (R) using data on the cloud.*

Service ViewBox w/	Upload local ver.	Download cloud ver.	Out-of-sync (no sync)
Dropbox	×	√	×
Seafile	×	√	×

Table 5: **Crash Consistency Results of ViewBox.** *The local version is inconsistent and rolled back to the previous version on the cloud.*

Workload	ext4 (MB/s)	ext4-cksum (MB/s)	Slowdown
Seq. write	103.69	99.07	4.46%
Seq. read	112.91	108.58	3.83%
Rand. write	0.70	0.69	1.42%
Rand. read	5.82	5.74	1.37%

Table 6: **Microbenchmarks on ext4-cksum.** *This figure compares the throughput of several micro benchmarks on ext4 and ext4-cksum. Sequential write/read are writing/reading a 1GB file in 4KB requests. Random write/read are writing/reading 128MB of a 1GB file in 4KB requests. For sequential read workload, ext4-cksum prefetches 8 checksum blocks for every disk read of a checksum block.*

Workload	ext4 (MB/s)	ext4-cksum (MB/s)	Slowdown
Fileserver	79.58	66.28	16.71%
Varmail	2.90	3.96	-36.55%
Webserver	150.28	150.12	0.11%

Table 7: **Macrobenchmarks on ext4-cksum.** *This table shows the throughput of three workloads on ext4 and ext4-cksum. Fileserver is configured with 50 threads performing creates, deletes, appends, and whole-file reads and writes. Varmail emulates a multi-threaded mail server in which each thread performs a set of create-append-sync, read-append-sync, read, and delete operations. Webserver is a multi-threaded read-intensive workload.*

1.8.0. All experiments are performed on machines with a 3.3GHz Intel Quad Core CPU, 16GB memory, and a 1TB Hitachi Deskstar hard drive. For all experiments, we reserve 512MB of memory for the view manager.

5.1 Cloud Helper

We first perform the same set of fault injection experiments as in Section 2. The corruption and crash test results are shown in Table 4 and Table 5. Because the local state is initially synchronized with the cloud, the cloud helper is able to fetch the redundant copy from cloud and recover from corruption and crashes. We also confirm that ViewBox is able to preserve causal consistency.

5.2 Ext4-cksum

We now evaluate the performance of standalone ext4-cksum, focusing on the overhead caused by data checksumming. Table 6 shows the throughput of several microbenchmarks on ext4 and ext4-cksum. From the table, one can see that the performance overhead is quite minimal. Note that checksum prefeteching is important for sequential reads; if it is disabled, the slowdown of the workload increases to 15%.

We perform a series of macrobenchmarks using Filebench on both ext4 and ext4-cksum with checksum prefetching enabled. The results are shown in Table 7. For the fileserver workload, the overhead of ext4-cksum is quite high, because there are 50 threads reading and writing concurrently and the negative effect of the extra seek for checksum blocks accumulates. The webserver workload, on the other hand, experiences little overhead, because it is dominated by warm reads.

It is surprising to notice that ext4-cksum greatly outperforms ext4 in varmail. This is actually a side effect of the ordering of data write-backs and journal commit, as discussed in Section 4.1.2. Note that because ext4 and ext4-cksum are not mounted with "journal_async_commit", the commit record is written to disk with a cache flush and the FUA (force unit access) flag, which ensures that when the commit record reaches disk, all previous dirty data (including metadata logged in the journal) have already been forced to disk. When running varmail in ext4, data blocks written by fsyncs from other threads during the journal commit are also flushed to disk at the same time, which causes high latency. In contrast, since ext4-cksum does not allow data write-back from fsync to run simultaneously with the journal commit, the amount of data flushed is much smaller, which improves the overall throughput of the workload.

5.3 View Manager

We now study the performance of various file system operations in an active view when a frozen view exists. The view manager runs on top of ext4-cksum.

We first evaluate the performance of various operations that do not cause copy-on-write (COW) in an active view. These operations are create, unlink, mkdir, rmdir, rename, utime, chmod, chown, truncate and stat. We run a workload that involves creating 1000 8KB files across 100 directories and exercising these operations on those files and directories. We prevent the active view from being frozen so that all these operations do not incur a COW. We see a

Operation	Normalized Response Time Before COW	After COW
unlink (cold)	484.49	1.07
unlink (warm)	6.43	0.97
truncate (cold)	561.18	1.02
truncate (warm)	5.98	0.93
rename (cold)	469.02	1.10
rename (warm)	6.84	1.02
overwrite (cold)	1.56	1.10
overwrite (warm)	1.07	0.97

Table 8: **Copy-on-write Operations in the View Manager.** *This table shows the normalized response time (against ext4) of various operations on a frozen file (10MB) that trigger copy-on-write of data blocks. "Before COW"/"After COW" indicates the operation is performed before/after affected data blocks are COWed.*

small overhead (mostly less than 5% except utime, which is around 10%) across all operations, as compared to their performance in the original ext4., This overhead is mainly caused by operation logging and other bookkeeping performed by the view manager.

Next, we show the normalized response time of operations that do trigger copy-on-write in Table 8. These operations are performed on a 10MB file after the file is created and marked COW in the frozen view. All operations cause all 10MB of file data to be copied from the active view to the frozen view. The copying overhead is listed under the "Before COW" column, which indicates that these operations occur before the affected data blocks are COWed. When the cache is warm, which is the common case, the data copying does not involve any disk I/O but still incurs up to 7x overhead. To evaluate the worst case performance (when the cache is cold), we deliberately force the system to drop all caches before we perform these operations. As one can see from the table, all data blocks are read from disk, thus causing much higher overhead. Note that cold cache cases are rare and may only occur during memory pressure. We further measure the performance of the same set of operations on a file that has already been fully COWed. As shown under the "After COW" column, the overhead is negligible, because no data copying is performed.

5.4 ViewBox with Dropbox and Seafile

We assess the overall performance of ViewBox using three workloads: openssh (building openssh from its source code), iphoto_edit (editing photos in iPhoto), and iphoto_view (browsing photos in iPhoto). The latter two workloads are from the iBench trace suite [15] and are replayed using Magritte [27]. We believe that these workloads are representative of ones people run with synchronization services.

The results of running all three workloads on ViewBox with Dropbox and Seafile are shown in Table 9. In all cases, the runtime of the workload in ViewBox is at most 5% slower and sometimes faster than that of the unmodified ext4 setup, which shows that view-based synchronization does not have a negative impact on the foreground workload. We also find that the memory overhead of ViewBox (the amount of memory consumed by the view manager to store frozen views) is minimal, at most 20MB across all three workloads.

We expect the synchronization time of ViewBox to be longer because ViewBox does not start synchronizing the current state of the file system until it is frozen, which may cause delays. The results of openssh confirm our expectations. However, for iphoto_view and iphoto_edit, the synchronization time on ViewBox with Dropbox is much greater than that on ext4. This is due to Dropbox's lack of proper interface support for views, as described in Section 4.2.3. Because both workloads use a file system image with around 1200 directories, to create the view metadata for each view, ViewBox has to query the Dropbox server numerous times, creating substantial overhead. In contrast, ViewBox can avoid this overhead with Seafile because it has direct access to Seafile's internal metadata. Thus, the synchronization time of iphoto_view in ViewBox with Seafile is near that in ext4.

Note that the iphoto_edit workload actually has a much shorter synchronization time on ViewBox with Seafile than on ext4. Because the photo editing workload involves many writes, Seafile delays uploading when it detects files being constantly modified. After the workload finishes, many files have yet to be uploaded. Since frozen views prevent interference, ViewBox can finish synchronizing about 30% faster.

6 Related Work

ViewBox is built upon various techniques, which are related to many existing systems and research work.

Using checksums to preserve data integrity and consistency is not new; as mentioned in Section 2.3, a number of existing file systems, including ZFS, btrfs, WAFL, and ext4, use them in various capacities. In addition, a variety of research work, such as IRON ext3 [22] and Z^2FS [31], explores the use of checksums for purposes beyond simply detecting corruption. IRON ext3 introduces transactional checksums, which allow the journal to issue all writes, including the commit block, concurrently; the checksum detects any failures that may occur. Z^2FS uses page cache checksums to protect the system from corruption in memory, as well as on-disk. All of these systems rely on locally stored redundant copies for automatic recovery, which may or may not be available. In contrast, ext4-cksum is the first work of which we are aware that employs the cloud for recovery. To our knowledge, it is also the first work to add data checksumming to ext4.

Similarly, a number of works have explored means

Workload	ext4 + Dropbox		ViewBox with Dropbox		ext4 + Seafile		ViewBox with Seafile	
	Runtime	Sync Time	Runtime	Sync Time	Runtime	Sync Time	Runtime	Sync Time
openssh	36.4	49.0	36.0	64.0	36.0	44.8	36.0	56.8
iphoto_edit	577.4	2115.4	563.0	2667.3	566.6	857.6	554.0	598.8
iphoto_view	149.2	170.8	153.4	591.0	150.0	166.6	156.4	175.4

Table 9: **ViewBox Performance.** *This table compares the runtime and sync time (in seconds) of various workloads running on top of unmodified ext4 and ViewBox using both Dropbox and Seafile. Runtime is the time it takes to finish the workload and sync time is the time it takes to finish synchronizing.*

of providing greater crash consistency than ordered and metadata journaling provide. Data journaling mode in ext3 and ext4 provides full crash consistency, but its high overhead makes it unappealing. OptFS [7] is able to achieve data consistency and deliver high performance through an optimistic protocol, but it does so at the cost of durability while still relying on data journaling to handle overwrite cases. In contrast, ViewBox avoids overhead by allowing the local file system to work in ordered mode, while providing consistency through the views it synchronizes to the cloud; it then can restore the latest view after a crash to provide full consistency. Like OptFS, this sacrifices durability, since the most recent view on the cloud will always lag behind the active file system. However, this approach is optional, and, in the normal case, ordered mode recovery can still be used.

Due to the popularity of Dropbox and other synchronization services, there are many recent works studying their problems. Our previous work [30] examines the problem of data corruption and crash inconsistency in Dropbox and proposes techniques to solve both problems. We build ViewBox on these findings and go beyond the original proposal by introducing view-based synchronization, implementing a prototype system, and evaluating our system with various workloads. Li et al. [19] notice that frequent and short updates to files in the Dropbox folder generate excessive amounts of maintenance traffic. They propose a mechanism called update-batched delayed synchronization (UDS), which acts as middleware between the synchronized Dropbox folder and an actual folder on the file system. UDS batches updates from the actual folder and applies them to the Dropbox folder at once, thus reducing the overhead of maintenance traffic. The way ViewBox uploads views is similar to UDS in that views also batch updates, but it differs in that ViewBox is able to batch all updates that reflect a consistent disk image while UDS provides no such guarantee.

7 Conclusion

Despite their near-ubiquity, file synchronization services ultimately fail at one of their primary goals: protecting user data. Not only do they fail to prevent corruption and inconsistency, they actively spread it in certain cases. The fault lies equally with local file systems, however, as they often fail to provide the necessary capabilities that would allow synchronization services to catch these errors. To remedy this, we propose ViewBox, an integrated system that allows the local file system and the synchronization client to work together to prevent and repair errors.

Rather than synchronize individual files, as current file synchronization services do, ViewBox centers around views, in-memory file-system snapshots which have their integrity guaranteed through on-disk checksums. Since views provide consistent images of the file system, they provide a stable platform for recovery that minimizes the risk of restoring a causally inconsistent state. As they remain in-memory, they incur minimal overhead.

We implement ViewBox to support both Dropbox and Seafile clients, and find that it prevents the failures that we observe with unmodified local file systems and synchronization services. Equally importantly, it performs competitively with unmodified systems. This suggests that the cost of correctness need not be high; it merely requires adequate interfaces and cooperation.

Acknowledgments

We thank the anonymous reviewers and Jason Flinn (our shepherd) for their comments. We also thank the members of the ADSL research group for their feedback. This material is based upon work supported by the NSF under CNS-1319405, CNS-1218405, and CCF-1017073 as well as donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Sony, and VMware. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] lvcreate(8) - linux man page.

[2] ZFS on Linux. http://zfsonlinux.org.

[3] Amazon. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.

[4] Apple. Technical Note TN1150. http://developer.apple.com/technotes/tn/tn1150.html, March 2004.

[5] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[6] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs˙last.pdf, 2007.

[7] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[8] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[9] Jonathan Corbet. Improving ext4: bigalloc, inline data, and metadata checksums. http://lwn.net/Articles/469805/, November 2011.

[10] Idilio Drago, Marco Mellia, Maurizio M. Munafò, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proceedings of the 2012 ACM conference on Internet measurement conference (IMC '12)*, Boston, MA, November 2012.

[11] Dropbox. The dropbox tour. https://www.dropbox.com/tour.

[12] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[13] Google. Google drive. http://www.google.com/drive/about.html.

[14] David Greaves, Junio Hamano, et al. git-read-tree(1): - linux man page. http://linux.die.net/man/1/git-read-tree.

[15] Tyler Harter, Charlotte Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 71–83, Cascais, Portugal.

[16] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[17] Minwen Ji, Alistair C Veitch, and John Wilkes. Seneca: remote mirroring done write. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.

[18] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 127–141, San Jose, California, February 2008.

[19] Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Y. Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. Efficient Batched Synchronization in Dropbox-like Cloud Storage Services. In *Proceedings of the 14th International Middleware Conference (Middleware 13')*, Beijing, China, December 2013.

[20] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, and Bull S.A.S. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[21] Microsoft. How ntfs works. http://technet.microsoft.com/en-us/library/cc781134(v=ws.10).aspx, March 2003.

[22] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[23] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9:1–9:32, August 2013.

[24] Seafile. Seafile. http://seafile.com/en/home/.

[25] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[26] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[27] Zev Weiss, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ROOT: Replaying Multithreaded Traces with Resource-Oriented Ordering. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[28] Microsoft Windows. Skydrive. http://windows.microsoft.com/en-us/skydrive/download.

[29] Erez Zadok, Ion Badulescu, and Alex Shender. Extending File Systems Using Stackable Templates. In *Proceedings of the USENIX Annual Technical Conference (USENIX '99)*, Monterey, California, June 1999.

[30] Yupu Zhang, Charlotte Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. *-Box: Towards Reliabil-ity and Consistency in Dropbox-like File Synchronization Services. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, California, June 2013.

[31] Yupu Zhang, Daniel S. Myers, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Zettabyte Reliability with Flexible End-to-end Data Integrity. In *Proceedings of the 29th IEEE Conference on Massive Data Storage (MSST '13)*, Long Beach, CA, May 2013.

CRAID: Online RAID Upgrades Using Dynamic Hot Data Reorganization

A. Miranda[§], T. Cortes[§‡]
[§]*Barcelona Supercomputing Center (BSC–CNS)* [‡]*Technical University of Catalonia (UPC)*

Abstract

Current algorithms used to upgrade RAID arrays typically require large amounts of data to be migrated, even those that move only the minimum amount of data required to keep a balanced data load. This paper presents CRAID, a self-optimizing RAID array that performs an online block reorganization of frequently used, long-term accessed data in order to reduce this migration even further. To achieve this objective, CRAID tracks frequently used, long-term data blocks and copies them to a dedicated partition spread across all the disks in the array. When new disks are added, CRAID only needs to extend this process to the new devices to redistribute this partition, thus greatly reducing the overhead of the upgrade process. In addition, the reorganized access patterns within this partition improve the array's performance, amortizing the copy overhead and allowing CRAID to offer a performance competitive with traditional RAIDs.

We describe CRAID's motivation and design and we evaluate it by replaying seven real-world workloads including a file server, a web server and a user share. Our experiments show that CRAID can successfully detect hot data variations and begin using new disks as soon as they are added to the array. Also, the usage of a dedicated partition improves the sequentiality of relevant data access, which amortizes the cost of reorganizations. Finally, we prove that a full-HDD CRAID array with a small distributed partition ($<$1.28% per disk) can compete in performance with an ideally restriped RAID-5 and a hybrid RAID-5 with a small SSD cache.

1 Introduction

Storage architectures based on Redundant Arrays of Independent Disks (RAID) [36, 10] are a popular choice to provide reliable, high performance storage at an acceptable economic and spatial cost. Due to the ever-increasing demand of storage capabilities, however, applications often require larger storage capacity or higher performance, which is normally achieved by adding new devices to the existing RAID volume. Nevertheless, several challenges arise when upgrading RAID arrays in this manner:

1. To regain uniformity in the data distribution, certain blocks must be moved to the new disks. Traditional approaches that try to preserve the round-robin order [15, 7, 49] end up redistributing large amounts of data between old and new disks, regardless of the number of new and old disks.
2. Alternative methods that migrate a minimum amount of data, can have problems to keep a uniform data distribution after several upgrade operations (like the Semi-RR algorithm [13]) or limit the array's performance (GSR [47]).
3. Existing RAID solutions with redundancy mechanisms, like RAID-5 and RAID-6, have the additional overhead of recalculating and updating the associated parities, as well as the necessary metadata updates associated to stripe migration.
4. RAID solutions are widely used in online services where clients and applications need to access data constantly. In these services, the downtime cost can be extremely high [35], and thus any strategy to upgrade RAID arrays should be able to interleave its job with normal I/O operations.

To address the challenges above, in this paper we propose a novel approach called CRAID, whose purpose is to minimize the overhead of the upgrade process by redistributing only "relevant data" in real-time. To do that, CRAID tracks data that is currently being used by clients and reorganizes it in a specific partition. This partition allows the volume to maintain the performance and distribution uniformity of the data that is actually being used by clients and, at the same time, significantly reduce the amount of data that must be migrated to new devices.

Our proposal is based on the notion that providing good levels of performance and load balance for the current working set suffices to preserve the QoS[1] of the RAID array. This notion is born from the following observations about long-term access patterns in storage: (i) data in a storage system displays a *non-uniform access frequency distribution*: when considering coarse-granularity time spans, "frequently accessed" data is usually a small fraction of the total data; (ii) this active data set exhibits *long-term temporal locality* and is *stable*, with small amounts of data losing or gaining importance gradually; (iii) even

[1] In this paper, the term QoS refers to the performance and load distribution levels offered by the RAID array.

Trace	Year	Workload	Reads (GB)		Writes (GB)		R/W ratio	Total accessed data (GB)	Accesses to Top 20% data
			Total	Unique	Total	Unique			
cello99	1999	research	73.73	10.52	129.91	10.92	0.62	203.65	65.77%
deasna	2002	research/email	672.4	23.32	231.57	45.45	2.54	903.97	86.88%
home02	2001	NFS share	269.29	9.07	66.35	4.49	3.94	335.64	61.36%
webresearch	2009	web server	–	–	3.37	0.51	–	3.37	51.33%
webusers	2009	web server	1.16	0.45	6.85	0.50	0.09	8.01	56.17%
wdev	2007	test server	2.76	0.2	8.77	0.42	0.21	11.54	72.44%
proj	2007	file server	2152.74	1238.86	367.05	168.88	7.33	2519.79	57.64%

Table 1: Summary statistics of 1-week long traces from seven different systems.

within the active data set, usage is heavily skewed, with "really popular" data receiving over 90% accesses [29].

These observations are largely intuitive and similar to the findings on short-term access patterns of other researchers [14, 20, 38, 2, 37, 42, 41, 5]. To our knowledge, however, there have not been any attempts to apply this information to the upgrade process of RAID arrays.

This paper makes the following contributions: we prove that using a large cache-like partition that uses all storage devices can be better than using dedicated devices due to the improved parallelism, in some cases even when the dedicated devices are faster. Additionally, we demonstrate that information about hot data can be used to reduce the overhead of rebalancing a storage system.

The paper is organized as follows: (i) we study the characteristics of several I/O workloads and show how the findings motivate CRAID (§2), (ii) we present the design of an online block reorganization system that adapts to changes in the I/O working set (§3), (iii) we evaluate several well-known cache management algorithms and their effectiveness in capturing long-term access patterns (§4), and (iv) we simulate CRAID under several real-system workloads to evaluate its merits and weaknesses (§5).

2 Characteristics of I/O Workloads

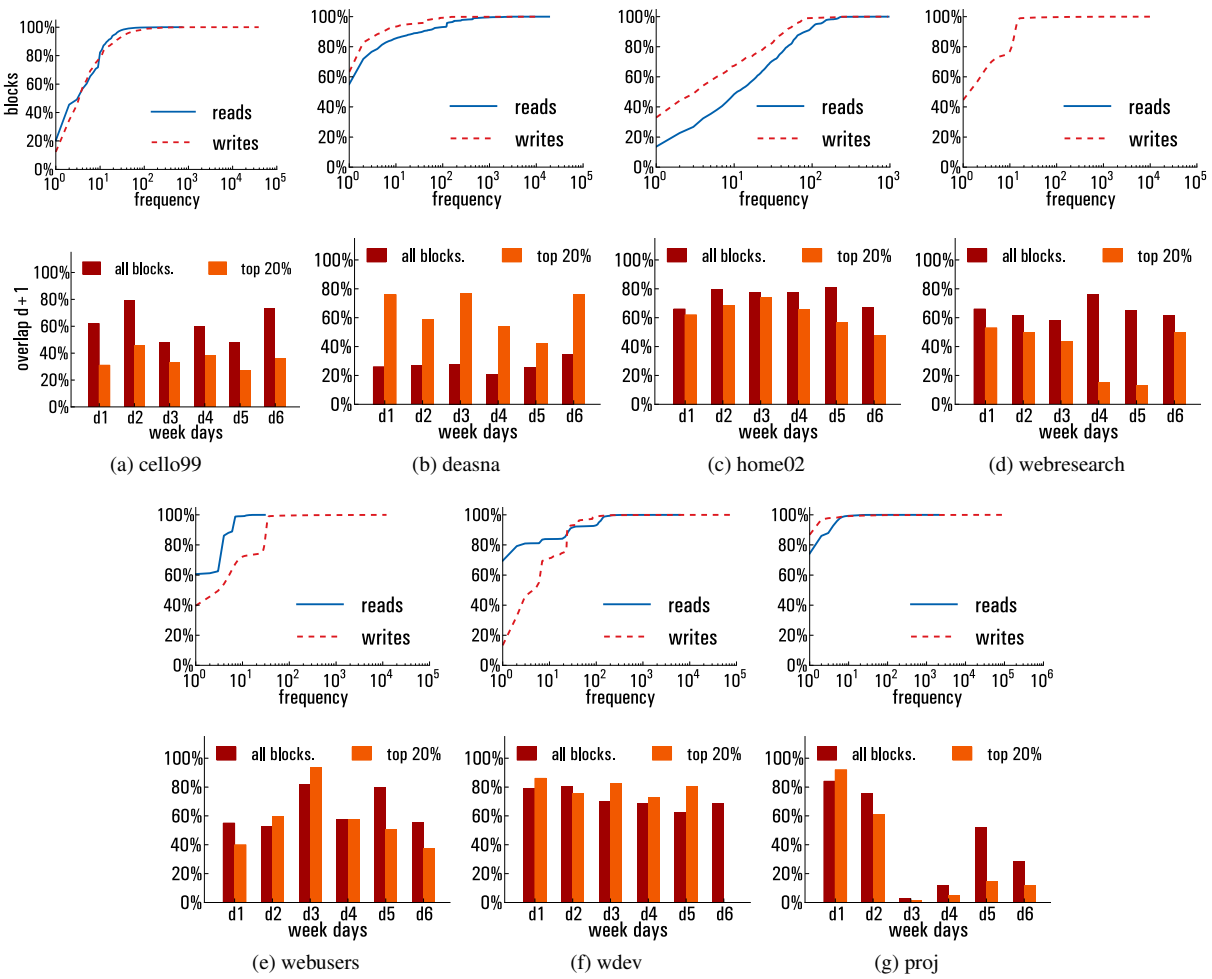
In this section we investigate the characteristics of several I/O workloads, focusing on those properties that directly motivate CRAID. In order for CRAID to be successful, the cost of reorganizing data must be lower than the potential gain obtained from the improved distribution, or it would not make sense to reorganize this data. Thus, we need to prove that long-term working sets exist and that they account for a large fraction of I/O. To do that, we analyzed a collection of well-known traces taken from several real systems. To increase the scope of our analysis, we use traces representing many different workloads and collected at different points in time over the last 13 years. Even if some of these traces are rather old, they can be helpful to establish a historical perspective on long-term hot data. Table 1 summarizes key statistics for one week of these traces, which we describe in detail below:

- The *cello99* traces are a set of well-known block-level traces used in many storage-related studies [22, 34, 46, 51]. Collected at HP Labs in 1999, they include one year of I/O workloads from a research cluster.
- The *deasna* traces [12] were taken from the NFS system at Harvard's Department of Engineering and Applied Sciences over the course of six weeks, in mid-fall 2002. Workload is a mix of research and email.
- The *home02* traces [12] were collected in 2001 from one of fourteen disk arrays in the Harvard CAMPUS NFS system. This system served over 10,000 school and administration accounts and consisted of three NFS servers connected to fourteen 53GB disk arrays. The traces collect six weeks worth of I/O operations.
- The *MSRC* traces [31] are block-level traces of storage volumes collected over one week at Microsoft Research Cambridge data center in 2007. The traces collected I/O requests on 36 volumes in 13 servers (179 disks). We use the *wdev* and *proj* servers, a test web server (4 volumes) and a server of project files (5 volumes), as they contain the most requests.
- The *SRCMap* traces are block-level traces collected by the Systems Research Laboratory (SyLab) at Florida International University in 2009 [41]. The traces were collected for three weeks at three production systems with several workloads. We use the *webresearch* and *webusers* workloads as they include the most requests. The first was an Apache web server managing FIU research projects, and the second a web server hosting faculty, staff, and graduate student web sites.

Our analysis of the traces shows that the following observations are consistent across all traces and, thus, validate the theoretical applicability of CRAID.

Obs. 1 *Data usage is highly skewed with a small percentage of blocks being heavily accessed.*

Fig. 1 (top row), shows the CDF for block access frequency for each workload. All traces show that the distribution of access frequencies is highly skewed: for read



(a) cello99 (b) deasna (c) home02 (d) webresearch

(e) webusers (f) wdev (g) proj

Figure 1: Block-frequency and working-set overlap for 1-week traces from seven different systems. The top row plots depict the CDF of block accesses for different frequencies: a point (f, p_1) on the block percentage curve indicates that p_1% of total blocks were accessed at most f times. Bottom row plots depict changes in the daily working-sets of the workloads: a bar (d, p_2) indicates that days d and $d+1$ had p_2% blocks in common. This is shown for all blocks and for the 20% blocks receiving more accesses.

requests ≈76–98% blocks are accessed 50 times or less, while for write requests this value rises to ≈89–98%. On the other hand, a small fraction of blocks (≈0.05–0.7%) is very heavily accessed in all cases (read or write requests).

This skew can also be seen in Table 1: the top 20% most frequently accessed blocks account for a large fraction (≈51–83%) of all accesses, which are similar results to those reported in previous studies [14, 24, 5, 41, 29].

Obs. 2 *Working-sets remain stable over long durations.*

Based on the first observation, we hypothesize that data usage exhibits long-term temporal locality. By long-term, we refer to a locality of hours or days, rather than seconds or minutes which is more typical of memory accesses. It is fairly common for a developer to work on a limited number of projects or for a user to access only a fraction of his data (like personal pictures or videos) over a few days or weeks. Even in servers, the popularity of the data accessed may result in long-term temporal locality. For instance, a very popular video uploaded to the web will receive bursts of accesses for several weeks or months.

Fig. 1 (bottom row), depicts the changes in the daily working-sets for each of the workloads. Each bar represents the percentage of unique blocks that are accessed both in day d and $d+1$. Most workloads show a significant overlap (≈55%–80%) between the blocks accessed in immediately successive days, and we also observe that there is a substantial overlap even when considering the top 20% most accessed blocks. Trace *deasna* is particularly interesting because it shows low values of overlap (≈20%–35%) when considering all accesses, which increases to ≈55%–80% for the top 20% blocks. This means that the working-set for this particular workload is more diverse but still contains a significant amount of heavily reused blocks. Based on the observations above, it seems reasonable that exploiting long-term temporal

locality and non-uniform access distribution can deal performance benefits. CRAID's goal is to use these to amortize the cost of data rebalancing during RAID upgrades.

3 CRAID Overview

The goal behind CRAID is to reduce the amount of data that needs to be migrated in reconfigurations while providing QoS levels similar to those of traditional RAID.

CRAID claims a small portion of each device and uses it to create a *cache partition* (P_C) that will be used to place copies of heavily accessed data blocks. The aim of this partition is to separate data that is currently important for clients from data that is rarely (if ever) used. Data not currently being accessed is kept in an *archive partition* (P_A) that uses the remainder of the disks. Notice that this partition can be managed by any data allocation strategy, but it is important that the archive can grow gracefully and any archived data is accessed with acceptable performance.

Effectively optimizing the layout of heavily used blocks within a small partition is beneficial for several reasons:

(i) It is possible to create a large cache by using a small fraction of all available disks, which allows important data to be cache-resident for longer periods.

(ii) A disk-based cache is a persistent cache: any optimized layout continues to be valid as long as it is warranted by access semantics, even if it is necessary to shutdown or reconfigure the storage system.

(iii) The size of the partition can be easily configured by an administrator or an automatic process to better suit storage demands.

(iv) Clustering frequently accessed data together offers the opportunity to improve access patterns: data accesses that were originally scattered can be sequentialized if the layout is appropriate. This also helps reduce seek times and rotational delays in all disks since "hot" blocks are placed close to each other.

(v) Whenever new devices are added, current strategies need to redistribute large amounts of data to be able to use them effectively and also to maintain QoS levels (e.g. performance or load balance). A disk-based cache offers a unique possibility to maintain QoS by redistributing only most accessed data. This should reduce the cost of the upgrade process significantly.

(vi) Extending the partition over all devices has three advantages over using dedicated devices. First, it maximizes the potential parallelism offered by the storage system. Second, it is much more likely to saturate a reduced set of dedicated devices than a large array. Third, benefits can be gained with the existing set of devices, without having to acquire more.

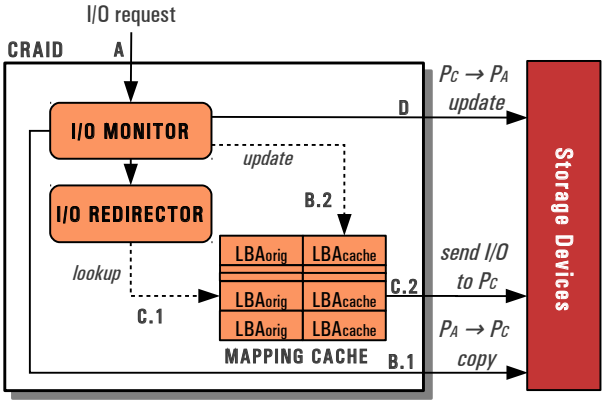


Figure 2: CRAID's I/O control flow.

Fig. 2 shows the control flow supported by CRAID's architecture: when an I/O request enters the system (A), it is captured by CRAID's *I/O monitor* which determines if the accessed data must be considered "active". If so, data blocks are copied to the caching partition if they are not already in it (B.1) and an appropriate mapping $\langle LBA_{original}, LBA_{cache} \rangle$ is stored in the *mapping cache* (B.2). From this point on, an *I/O redirector* will redirect all future accesses to $LBA_{original}$ to the caching partition (C.1 and C.2). This continues until the I/O monitor decides that data is no longer active and removes the entry from the mapping cache. Any update to the contents of the data is then written back to P_A (D). This flow means that the upgrade process begins immediately when a new disk is added to CRAID (which forces P_C to grow), and is interleaved with the array's normal I/O operation. This permits CRAID to use the new disks from the moment they are added to the array.

4 Detailed Design

This section elaborates on CRAID's design details by discussing its individual components mentioned in §3: the *I/O monitor*, the *I/O redirector* and the *mapping cache*.

4.1 I/O Monitor

The *I/O monitor* is responsible for analyzing I/O requests to identify the working set and schedule the appropriate operations to copy data between partitions. The I/O monitor uses a conservative definition of working set that includes the latest k distinct blocks that have been *more active*, where k is P_C's current capacity.

When a request forces an eviction in P_C, the I/O monitor checks if the cached copy is dirty and, if so, schedules the corresponding I/O operations to update the original data. Otherwise, the data is replaced by the newly cached block. Currently, the I/O monitor supports the following simple policies in order to make replacement decisions:

- *Least Recently Used* (LRU) uses recency of access to decide if a block has to be replaced.

- *Least Frequently Used with Dynamic Aging* (LFUDA) uses popularity of access and replaces the block with the smallest key $K_i = (C_i * F_i) + L$, where C_i is the retrieval cost, F_i is a frequency count and L is a running age factor that starts at 0 and is updated for each replaced block [3].
- *Greedy-Dual-Size with Frequency* (GDSF) includes the size of the original request, S_i, and replaces the block with minimum $K_i = (C_i * F_i)/S_i + L$ [21, 9, 3].
- *Adaptive Replacement Cache* (ARC) [28] balances between recency and frequency in an online and self-tuning fashion. ARC adapts to changes in the workload by tracking *ghost hits* (recently evicted entries) and replaces either the LRU or LFU block depending on recent history.
- *Weighted LRU* (WLRU_w) is a simple extension of the LRU algorithm that tries to find the least recently used block that is also *clean* (i.e. not dirty). In order to avoid lengthy $O(k)$ traversals it uses a parameter $w \in \mathbb{R}$ to limit the number of blocks that will be evaluated to $k * w$. If no suitable candidate is found in $k * w$ steps, the LRU block is replaced.

We evaluate the effectiveness of these basic strategies to accurately predict the workload in §5.1. We implemented these basic strategies instead of more complex ones because these algorithms are typically extremely efficient and consume few resources, which makes them suitable to be included in a RAID controller. Furthermore, their prediction rates are usually quite high. Exploring more sophisticated strategies and/or data mining approaches to model complex data interrelations is left for the future.

The I/O monitor is also in charge of rebalancing P_C. When new devices are added, the I/O monitor invalidates all the blocks contained in P_C (writing back to P_A the copies that need updating) and starts filling it with the current working set when blocks are requested. This conservative approach allows us to create long sequential chains of potentially related blocks, which improves the sequentiality and parallelism of the data in P_C. Note that since P_C always holds 'hot blocks', the rebalancing is never completely finished unless the working set remains stable for a long time. Nevertheless, as we show in §5, the cost of this 'on-line' monitoring and rebalancing is amortized by the performance obtained.

4.2 Mapping Cache

The *mapping cache* is an in-memory data structure used to translate block addresses in the P_A to their corresponding copies in P_C. The structure stores, for each block copied to P_C, the block's LBA in P_A, the corresponding LBA in P_C and a flag indicating if the cached copy has been modified.

Our current implementation uses a tree-based binary structure to handle mappings, which ensures that the total time complexity for a lookup operation is given by $O(log\, k)$. Concerning memory, for every block in P_C, CRAID stores 4 bytes for each LBA and 1 dirty bit, plus 8 additional bytes for the structure pointer. Assuming that all k blocks are occupied, that the configured block size is 4KB and P_C size of S GB, the worst case memory requirement is $2 \times S$ MB for LBAs, $S/2^5$ for the dirty information, and $4 \times S$ MB for the tree pointers. Thus, in the worst case, CRAID requires memory of 0.58% the size of the cache partition, or ≈5.9MB per GB, an acceptable requirement for a RAID controller.

Notice that the destruction of the mapping cache can lead to data loss since block updates are performed in place in the cache partition. Failure resilience of the mapping cache is provided by maintaining a persistent log of which cached data blocks have been modified and their translations. This ensures that these blocks, whose cached copies were not identical to the original data, can be successfully recovered. Blocks that were not dirty in P_C don't need to be recovered and are invalidated.

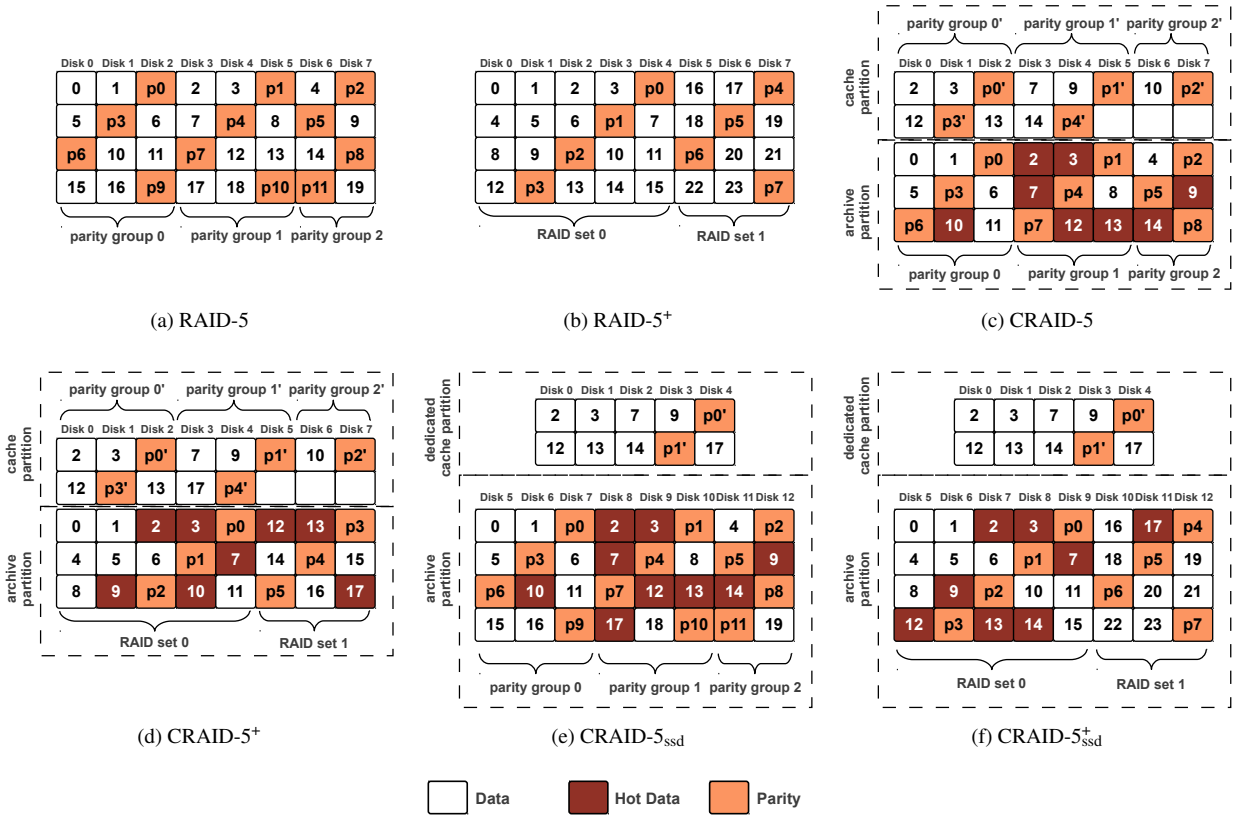
4.3 I/O Redirector

The *I/O redirector* is responsible for intercepting all read and write requests sent to the CRAID volume and redirect them to the appropriate partition. For each request, it first checks the mapping cache for an existing cached copy. If none is found, the request is served from P_A. Otherwise, the request is dispatched to the appropriate location in P_C. Multi-block I/Os are split as required.

5 Evaluation

In this section we evaluate CRAID's behavior using a storage simulator. We seek to answer the following questions: (1) How well does CRAID capture working sets? (2) How does CRAID impact performance? (3) How sensitive is load balance to CRAID's I/O redirection? To answer these questions, we evaluate CRAID under realistic workloads, using detailed simulations where we replay the real-time storage traces described in §2. Since some of these traces include data collected over several weeks or months, which makes them intractable for fine-grained simulations, we simulate an entire continuous week (168 hours) chosen at random from each dataset. Note that in this paper, we only describe the evaluations of several CRAID variants that use RAID-5 in P_C. For brevity's sake, we do not include similar results with RAID-0 [4].

Simulation system. The simulator consists of a workload generator and a simulated storage subsystem composed of



(a) RAID-5 (b) RAID-5$^+$ (c) CRAID-5

(d) CRAID-5$^+$ (e) CRAID-5$_{ssd}$ (f) CRAID-5$^+_{ssd}$

Figure 3: Overview of the different allocation policies evaluated.

an array controller and appropriate storage components. For each request recorded in the traces, the workload generator issues a corresponding I/O request at the appropriate time and sends it down to the array controller.

The array controller's main component is the *I/O processor* which encompasses the functions of both the I/O monitor and the I/O redirector. According to the incoming I/O address, it checks the *mapping cache* and forwards it to the caching partition's segment of the appropriate disk. The workload generator, the mapping cache and the I/O processor are implemented in C++, while the different storage components are implemented in DiskSim. DiskSim [8] is an accurate and thoroughly validated disk system simulator developed in the Carnegie Mellon University, which has been used extensively in research projects to study storage architectures [1, 32, 50, 25].

All experiments use a simulated testbed consisting of Seagate Cheetah 15,000 RPM disks [39], each with a capacity of 146GB and 16MB of cache. This is the latest (validated) disk model available to Disksim. Though somewhat old, we decided to use these disks in order to use the detailed simulation model offered by Disksim, rather than a less detailed one. Besides, since our analysis is a comparative one, the disks' performance should benefit or harm all strategies equally. For the simulations involving SSDs, we use Microsoft Research's idealized SSD model [1]. Since the capacity and number of disks in the original traced systems differs from our testbed, we determine the datasets for each trace via static analysis. These datasets are mapped onto the simulated disks uniformly so that all disks have the same access probability.

Trace	LRU	LFUDA	GDSF	ARC	WLRU$_{0.5}$
cello99	**65.23**	65.23	48.75	**65.66**	65.22
deasna	89.63	**89.90**	67.24	89.65	**89.73**
home02	**93.91**	93.86	77.93	**93.92**	93.90
webresearch	81.14	78.92	54.41	**82.38**	**82.14**
webusers	80.40	78.72	60.49	**81.01**	**81.40**
wdev	91.04	**91.88**	32.78	**91.06**	91.02
proj	75.55	75.73	25.43	**75.58**	**75.65**

Table 2: Hit ratio (%) for each cache partition management algorithm. Best and second best shown in bold.

Strategies evaluated. All experiments evaluate the six following allocation policies, an overview of which is shown in Fig. 3:

- **RAID-5:** A RAID-5 configuration that uses all disks available. Stripes are as long as possible but are divided into *parity groups* to improve the robustness and recov-

Trace	LRU	LFUDA	GDSF	ARC	$WLRU_{0.5}$
cello99	34.76	34.76	51.24	**34.31**	**33.76**
deasna	10.36	**10.09**	32.74	10.34	**10.34**
home02	6.08	6.13	22.06	**6.07**	**6.08**
webresearch	18.84	21.06	45.58	**17.60**	**18.83**
webusers	19.58	21.26	39.50	**18.98**	**19.28**
wdev	8.88	**8.04**	67.13	8.85	**8.58**
proj	24.42	**24.24**	74.55	**24.39**	24.72

Table 3: Replacement ratio (%) for each cache partition management algorithm. Best and second best in bold.

erability of the array (Fig. 3a). This policy will help establish a comparison baseline as it provides maximum parallelism and ideal workload distribution. Notice, however, that expanding such an array in real life can be prohibitively expensive.

- **RAID-5⁺:** A RAID-5 configuration that has been expanded and restriped several times. Each expansion phase adds 30% additional disks [27] that constitute a new independent RAID-5. Thus the system can be considered a collection of independent RAID-5 arrays (or *sets*), each with its own stripe size, that have been added to expand the storage capacity (see Fig. 3b). This serves as a comparison baseline to a realistic system upgraded many times.
- **CRAID-5** and **CRAID-5⁺:** CRAID configurations that use RAID-5 for the caching partition. CRAID-5 also uses RAID-5 for the archive partition while CRAID-5⁺ uses RAID-5⁺. The first one serves to evaluate the performance impact of using CRAID on an ideally restriped RAID-5 and the effects on performance of data transfers from/to the cache. With the second one, we evaluate the benefits of using CRAID in a storage system that has grown several times, with a P_A that grows by aggregation.
- **CRAID-5_{ssd}** and **CRAID-5^{+}_{ssd}:** CRAID configurations analogous to CRAID-5 and CRAID-5⁺ but using a fixed number of SSDs for the cache partition. This allows us to evaluate the advantages, if any, of using disk-based CRAID against using dedicated SSDs, which is a common solution offered by storage vendors.

We simulate RAID-5 and RAID-5⁺ in their ideal state, i.e., when the dataset has been completely restriped. The reason is that since CRAID is permanently in an "expansion" phase and sacrifices a small amount of capacity, in order to be useful its performance should be closer to an optimum RAID-5 array, rather than one being restriped. All the arrays simulated use 50 disks, a number chosen based on the datasets of the traces examined, except those for CRAID-5_{ssd} and CRAID-5^{+}_{ssd} that include 5 additional SSDs (10%) for the dedicated cache. RAID-5 uses a parity group size of 10 disks both as a stand-alone

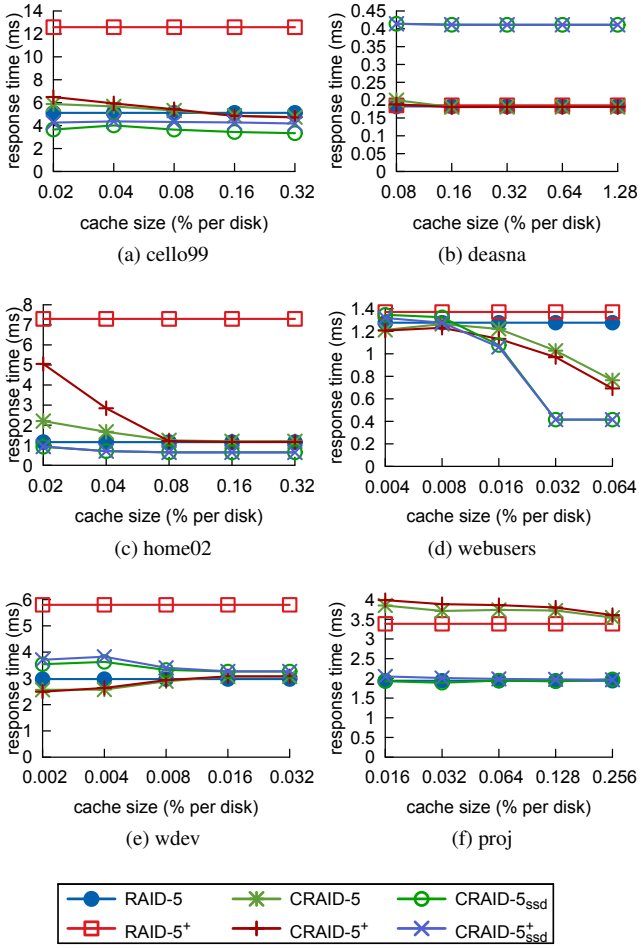


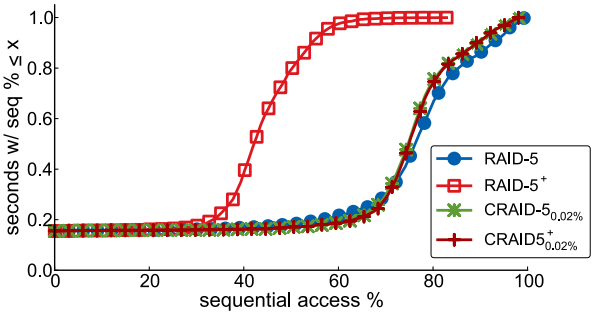
Figure 4: Comparison of I/O response time (read requests).

allocation policy and as a part of a CRAID configuration. Similarly, RAID-5⁺ begins with 10 disks and adds a new array of 3, 4, 5, 7, 9 and 12 disks (+30%) in each expansion step until the 50-disk mark is reached. The stripe unit for all policies is 128KB based on Chen's and Lee's work [11]. In all experiments, the cache partition begins in a cold state.

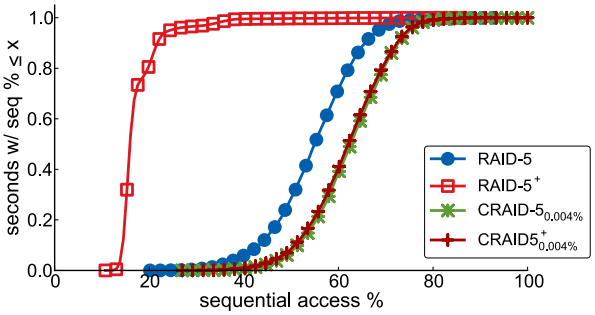
5.1 Cache Partition Management

Here we evaluate the effectiveness to capture the working set of the different cache algorithms supported by the I/O monitor (refer to §4.1). In this experiment we are concerned with the ideal results of the prediction algorithms to select the best one for CRAID. Thus, we use a simplified disk model that resolves each I/O instantly, and allows us to measure the properties of each algorithm with no interferences. The remaining experiments use the more realistic disk model.

Tables 2 and 3 show, respectively, the hit and replacement ratio delivered by each algorithm using a P_C size of 0.1% the weekly working set. We observe that, for each trace, all algorithms except one show similar hit and re-



(a) cello99



(b) webusers

Figure 5: Sequential access distribution (CDF) for the cello99 *and* webusers *traces. Sequentiality percentages captured each second. Other traces show similar results.*

placement ratios with the ARC algorithm showing the best results in both evaluations. The only exception is the GDSF algorithm, which shows significantly worse results due to the addition of the request size as a metric which does not seem very useful in this kind of scenario.

For CRAID strategies based on RAID-5, however, evictions of clean blocks are preferred as long as the effectiveness of the algorithm is not compromised. This is because evicting a dirty block forces CRAID to update the original blocks and its parity in the P_A, which requires 4 additional I/Os (2 reads and 2 writes). In this regard, the WLRU strategy is more suitable since it helps reduce the number of I/O operations needed to keep consistency: if the data block replaced has not been modified, there's no need to copy it back to P_A. Thus, in the following experiments we configure the I/O monitor with the $\text{WLRU}_{0.5}$ algorithm since it shows hit and replacement ratios similar to ARC, and reduces the amount of dirty evictions.

5.2 Response Time

In this section we evaluate the performance impact of using CRAID. For each allocation policy and configuration, we measure the response time of each read and write request occurred during the simulations. Figs. 4 and 6 show the response time measurements[2] of each CRAID variant, compared to the RAID-5 and RAID-5$^+$ baselines.

Note that each strategy was simulated with different cache partition sizes in order to estimate the influence of this parameter on performance. In the results shown in this section, the cache partition is successively doubled until no evictions have to be performed. This represents the best case for CRAID since data movement between the partitions is reduced to a minimum.

Read requests. The results for read requests are shown in Fig. 4. First, we observe that requests take notably longer to complete in RAID-5$^+$ than in RAID-5 in all cases. This is to be expected since the longer stripes in RAID-5 increase its potential parallelism and provide a more effective workload distribution.

Second, in most traces, hybrid strategies CRAID-5 and CRAID-5$^+$ offer performance comparable to that of an ideal RAID-5, or even better for certain cache sizes (e.g. *webusers* trace, Fig. 4d). The explanation lies in the fact that CRAID's cache partition is able to better exploit the spatial locality available in commonly used data: colocating hot data in a small area of each disk helps reduce seek times when compared to the same data being randomly spread over the entire disk, and also increases the sequentiality of access patterns. This can be seen in Fig. 5, that shows the probability distribution (CDF) of the *sequential access percentage* for the *cello99* and *webusers* traces (computed as $\frac{\#Seq\_Access}{\#Accesses}$ and aggregated per second of simulation). Here we see that access sequentiality in CRAID-5 and CRAID-5$^+$ is similar to that of RAID-5 and significantly better than that of RAID-5$^+$. This helps reduce the response time per request and contributes to the overall performance of the array.

Nevertheless, CRAID's effectiveness depends on how well hot data is predicted. Despite the good results shown in §5.1, Fig. 4f shows that performance results for the *proj* trace are not as good as in the other traces. Table 4 shows that CRAID's best hit ratio for the *proj* trace is lower than in other traces (e.g. 85.25% vs. 99.51% in *home02*) and that its eviction count is higher. These two factors contribute to more data being transferred to the cache partition and explain the drop in performance.

Most interestingly, the performance and sequentiality provided by CRAID-5$^+$ is similar to that of CRAID-5, even though it uses a RAID-5$^+$ strategy for the archive partition. This proves that the cache partition is absorbing most of the I/O, and the array behaves like an ideal RAID-5, regardless of the strategy used for stale data.

Third, increasing the size of the cache partition improves read response times in all CRAID-5 variants. This is to be expected since a larger cache partition increases the probability of a cache hit and also decreases the number of evictions, which greatly improves the effectiveness of the strategy. In most traces, however, once a certain partition

[2] 95% confidence interval.

Trace	Best hit ratio		Worst eviction ratio	
	reads	writes	reads	writes
cello99	97.85%	98.88%	21.28%	9.53%
deasna	99.53%	97.80%	0.92%	3.17%
home02	99.51%	99.53%	3.32%	2.59%
webresearch	-	98.76%	-	7.66%
webusers	94.95%	99.33%	16.65%	6.56%
wdev	98.62%	99.40%	1.90%	10.76%
proj	85.25%	88.45%	21.97%	9.13%

Table 4: Best hit ratio and worst eviction ratio (all simulations).

Strategy	Mean		99^{th} pctile		Max	
	Ioq	*Cdev*	*Ioq*	*Cdev*	*Ioq*	*Cdev*
CRAID-5$^+$	2.11	8.65	20	44	381	50
CRAID-5$^+_{ssd}$	4.74	6.49	45	23	427	40

Table 5: Comparison of CRAID's SSD-dedicated vs. full-HDD approach. Ioq: ioqueue size, Cdev: concurrent devices. Trace: wdev*, P_C size: 0.002%. Other traces show similar results.*

size S_M is reached, response times stop improving (e.g. *deasna* with $S_M = 0.16\%$ or *home02* with $S_M = 0.08\%$, Figs. 4b and 4c, respectively). Examination of these traces shows that CRAID is able to achieve a near maximum hit ratio with a partition of size S_M, and increasing it further provides barely noticeable benefits.

Finally, we see that the performance with dedicated SSDs is better than using a distributed partition for most traces. This is to be expected since SSDs are significantly faster than HDDs, and requests can be completed fast enough to avoid saturating the devices. Note, however, that for some P_C sizes, full-HDD CRAID is able to offer similar performance levels (Figs. 4a, 4b, 4d, and 4e), and, given the difference in $/GB between SSDs and HDDs, it might be an appropriate option when it is not possible to add 10% SSDs to the storage architecture. Additionally, a full-SSD RAID should also benefit from the improved parallelism offered by an optimized P_C.

Write requests. The results for write requests are shown in Fig. 6. Similarly to read requests, we observe that write requests are significantly slower in RAID-5$^+$ than in RAID-5, for all traces. Most importantly, the hybrid strategies CRAID-5 and CRAID-5$^+$ perform better than traditional RAID-5 in all traces except *webusers*, where performance is slightly below that of RAID-5.

These improved response times can be explained by two reasons. First, since write requests are always served from the cache partition (except in the case of an eviction), response times benefit greatly from the improved spatial locality and sequentiality provided by the cache partition.[3] Second, the smaller the P_C fragment for each disk

[3]Obviously, as long as the prediction of the working set is accurate.

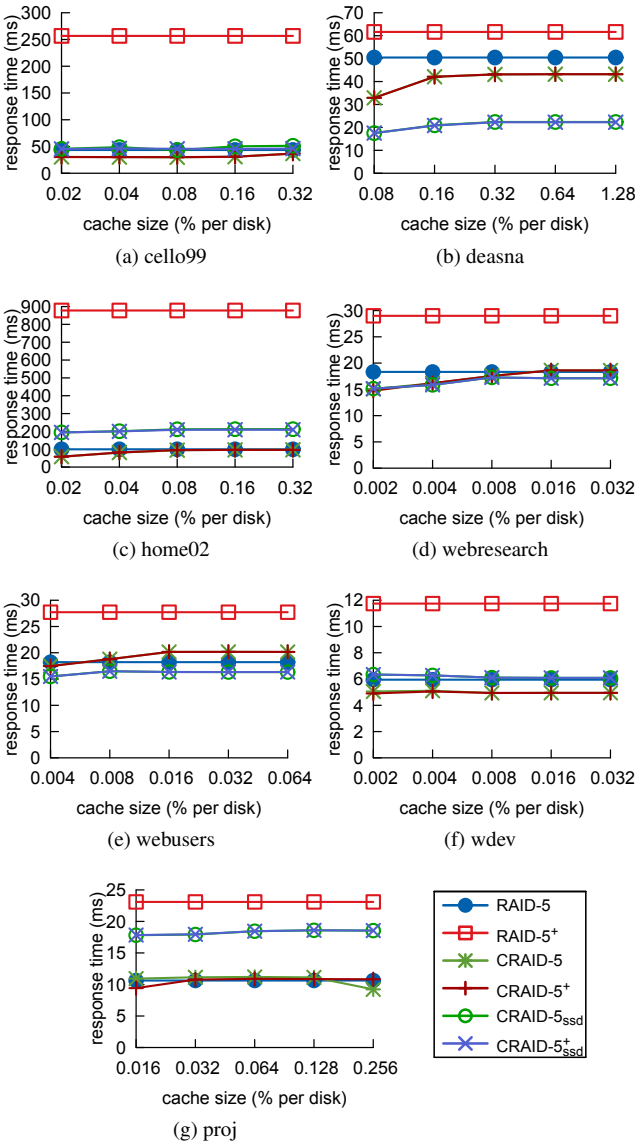


Figure 6: Comparison of I/O response time (write requests).

is, the more likely it is that accesses to this fragment benefit from the disk's internal cache. This explains why response times in Fig. 6 increase slightly for larger partition sizes: a smaller P_C means more evictions in CRAID, but it also means a smaller fragment for each disk and a more effective use of its internal cache. The effect of this internal cache is highly beneficial, to the point that it amortizes the additional work produced by extra evictions.

On the other hand, SSD-based strategies CRAID-5$_{ssd}$ and CRAID-5$^+_{ssd}$ show significantly worse response times than their full-HDD counterparts in some traces (see Figs. 6a, 6c, 6f, or 6g). Examination of these traces reveals that the I/O queues in the dedicated SSDs have significantly more pending requests than those in full-HDD CRAID. Also, the number of concurrently active disks during the simulation is lower (see Table 5). In ad-

Trace	CRAID-5 P_C		CRAID-5$^+$ P_C	
	best c_v	**worst** c_v	**best** c_v	**worst** c_v
cello99	0.02%	0.32%	0.02%	0.32%
deasna	0.08%	1.28%	0.08%	1.28%
home02	0.02%	0.32%	0.02%	0.32%
webresearch	0.002%	0.032%	0.002%	0.032%
webusers	0.004%	0.064%	0.004%	0.064%
wdev	0.002%	0.032%	0.002%	0.032%
proj	0.016%	0.256%	0.016%	0.256%

Table 6: Influence of P_C size on workload distribution.

dition, we discovered that Disksim's SSD model does not simulate a read/write cache. Thus, the lower number of pending requests coupled with the HDD cache benefit explained above, makes full-HDD CRAID faster for write requests in some traces.

5.3 Workload Distribution

In this experiment we evaluate CRAID's ability to maintain a uniform workload distribution. For each second of simulation we measure the I/O load in MB received by each disk and we compute the *coefficient of variation* as a metric to evaluate the uniformity of its distribution. The coefficient of variation (c_v) expresses the standard deviation as a percentage of the average ($\frac{\sigma}{\mu}$), and can be interpreted as how the actual workload deviates from an ideal distribution.[4] We perform this experiment for all strategies described and uses the same P_C sizes of §5.2.

Impact of CRAID. Figs. 7a and 7b show CDFs of c_v per % of samples (seconds) for the *deasna* and *wdev* traces, respectively. Notice that for CRAID strategies we show both the *best* and *worst* curves obtained (Table 6 shows the correspondence with actual P_C sizes) and we compare them with the results for RAID-5 and RAID-5$^+$.
We observe that there is a significant difference between the workload distribution provided by RAID-5 and that of RAID-5$^+$, which is to be expected since the "segmented" nature of RAID-5$^+$ naturally hinders a uniform workload distribution. Most interestingly, all CRAID strategies demonstrate a workload distribution very similar to (and sometimes better than) RAID-5. More importantly, this benefit appears in even those CRAID configurations that use RAID-5$^+$ for the archive partition, despite its poor performance and uneven distribution. This proves that the cache partition is successful in absorbing most I/O, and that it behaves close to an ideal RAID-5 despite the cost of additional data transfers.

Influence of the cache partition size. Though barely noticeable, an unexpected result is that, in all traces, the workload distribution degrades as the cache partition grows (see Table 6). Examination of the traces shows that a larger cache partition slightly increases the probability that certain subsets of disks are more used than others due to the different layout of data blocks. This is reasonable since our current prototype doesn't perform direct actions to enforce a certain workload distribution, but rather relies on the strategy used for the cache partition. Improving CRAID to employ workload-aware layouts is one of the subjects of our future investigation.

Workload with dedicated SSDs. The curves shown in Figs. 7a and 7b show a worse workload distribution for CRAID-5_{ssd} and CRAID-5^+_{ssd} when compared to the full-HDD strategies. This is to be expected since the dedicated SSDs absorb much of the I/O workload and end up degrading the global workload of the system. Note that this does not necessarily mean that the workload directed to the dedicated disks is unbalanced, but rather that the other devices are underutilized. This proves that a spread partition has a higher chance of producing a balanced workload, and can compete in performance, than a dedicated one, even if the devices used for the latter are faster.

6 Discussion and Future Work

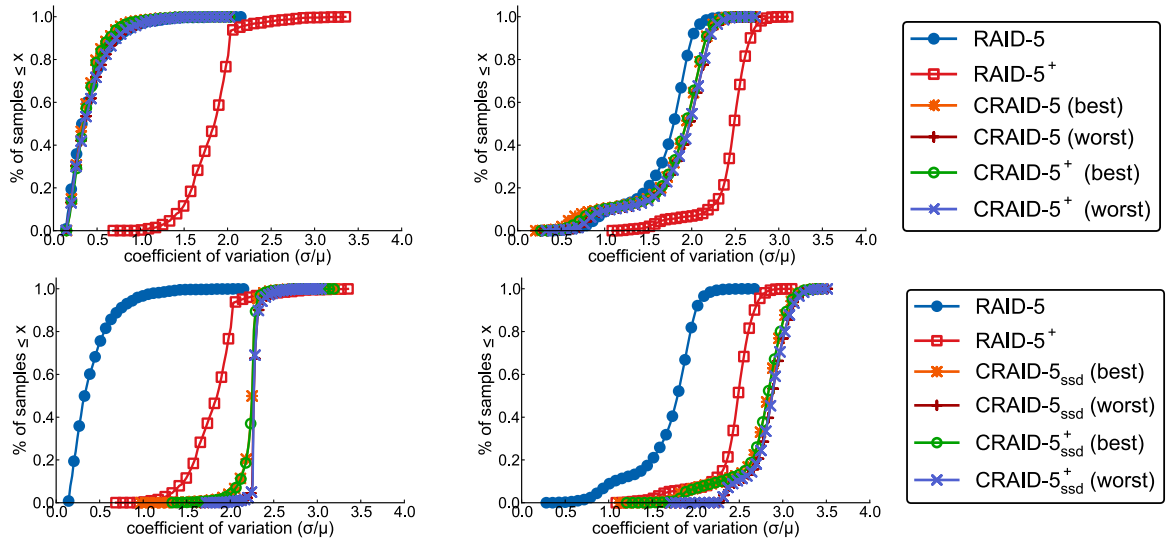
While our experiences with CRAID have been positive in RAID-0 and RAID-5 storage, we believe that they can also be applied to RAID-6 or more general erasure codes, since the overall principle still applies: rebalancing hot data should require less work than producing an ideal distribution. The main caveat of our solution, however, is the cost of additional parity computations and I/O operations for dirty blocks, which directly increases with the number of parity blocks required. Whether this cost can be leveraged by the performance benefits obtained, will be explored in a fully-fledged prototype.

It should also be possible to extend the proposed solution beyond RAID arrays, adapting the techniques to distributed or tiered storage. Specifically, we believe the monitoring of interesting data could be adapted to work with pseudo-randomized data distributions like CRUSH [43] or Random Slicing [30] in order to reduce data migration during upgrades. What to do with blocks that stop being interesting is a promising line of research.

Additionally, while the current CRAID prototype has served to verify that it is possible to amortize the cost of a RAID upgrade by using knowledge about hot data blocks, it uses simple algorithms for prediction and expansion. We envision several ways to improve the current prototype that can serve as subjects of future research.

Smarter prediction. The current version of CRAID does not take into account the relations between blocks in order to copy them to the caching partition, but rather relies on the fact that blocks accessed consecutively in a short

[4] The smaller c_v is, the more uniform the data distribution.



(a) deasna (b) wdev

Figure 7: CRAID workload distribution: full-HDD (top) vs SSD-dedicated (bottom). Figures show CDFs of c_v per % of samples (seconds) for traces deasna *and* wdev. *Other traces show similar results.*

period of time tend to be related. More sophisticated techniques to detect block correlations could improve CRAID significantly, allowing the I/O monitor to migrate data to P_C before it is actually needed.

Smarter rebalancing. The current invalidation of the entire P_C when new disks are added is overkill. Though it benefits the parallelism of the data distribution and new disks can be used immediately, the current strategy was devised to test if our hypothesis held in the simplest case, without complex algorithms. Since working sets should not change drastically, CRAID could benefit greatly from strategies to rebalance the small amount of data in P_C more intelligently, like those in §7.2.

Improved data layout. Similarly, currently CRAID does not make any effort to allocate related blocks close to each other. Alternate layout strategies more focused on preserving semantic relations between blocks might yield great benefits. For instance, it might be interesting to evaluate the effect of copying entire stripes to the cache partition as a way to preserve spatial locality. Besides, this could help reduce the number of parity computations, thus reducing the background I/O present in the array.

7 Related Work

We examine the literature by organizing it into data layout optimization techniques and RAID upgrade strategies.

7.1 Data Layout Optimization

Early works on optimized data layouts by Wong [45], Vongsathorn *et al.* [42] and Ruemmler and Wilkes [37] argued that placing frequently accessed data in the center of the disk served to minimize the expected head movement. Specifically, the latter proved that the best results in I/O performance came from infrequent shuffling (weekly) with small (block/track) granularity. Akyurek and Salem also showed the importance of reorganization at the block level, and the advantages of copying over shuffling [2].

Hu *et al.* [48, 33] proposed an architecture called Disk Caching Disk (DCD), where an additional disk (or partition) is used as a cache to convert small random writes into large log appends, thus improving overall I/O performance. Similarly to DCD, iCache [16] adds a log-disk along with a piece of NVRAM to create a two-level cache hierarchy for iSCSI requests, coalescing small requests into large ones before writing data. HP's AutoRAID [44], on the other hand, extends traditional RAID by partitioning storage in a mirrored zone and a RAID-5 zone. Writes are initially made to the mirrored zone and later migrated in large chunks to RAID-5, thus reducing the space overhead of redundancy information and increasing parallel bandwidth for subsequent reads of active data.

Li *et al.* proposed C-Miner [26], which used data mining techniques to model the correlations between different block I/O requests. Hidrobo and Cortes [18] accurately model disk behavior and compute placement alternatives to estimate the benefits of each distribution. Similar techniques could be used in CRAID to infer complex access patterns and reorganize hot data more effectively.

ALIS [20] and, more recently, BORG [5], reorganize frequently accessed blocks (and block sequences) so that they are placed sequentially on a dedicated disk area. Contrary to CRAID, neither explores multi-disk systems.

7.2 RAID Upgrade Strategies

There are several deterministic approaches to improve the extensibility of RAID-5. HP's AutoRAID allows an online capacity expansion without data migration, by which newly created RAID volumes use all disks and previously created ones use only the original disks.

Conventional approaches redistribute data and preserve the round-robin order. Gonzalez and Cortes proposed a Gradual Assimilation (GA) algorithm [15] to control the overhead of expanding a RAID-5 system, but it has a large redistribution cost since all parities still need to be modified after data migration. US patent #6000010 presents a method to scale RAID-5 volumes that eliminates the need to rewrite data and parity blocks to the original disks [23]. This, however, may lead to an uneven distribution of parity blocks and penalize write requests.

MDM [17] reduces data movement by exchanging some blocks between the original and new disks. It also eliminates parity modification costs since all parity blocks are maintained, but it is unable to increase (only keep) the storage efficiency by adding new disks. FastScale [50] minimizes data migration by moving only data blocks between old and new disks. It also optimizes the migration process by accessing physically sequential data with a single I/O request and by minimizing the number of metadata writes. At the moment, however, it cannot be used in RAID-5. More recently, GSR [47] divides data on the original array into two sections and moves the second one onto the new disks keeping the layout of most stripes. Its main limitation is performance: after upgrades, accesses to the first section are served by original disks, and accesses to the second are served only by newer disks.

Due to the development of object-based storage, randomized RAID is becoming more popular, since it seems to have better scalability. The cut-and-paste strategy proposed by Brinkmann *et al.* [6] uses a randomized function to place data across disks. When a disk is added to disks, it cuts off ranges of data $[1/(n+1), 1/n]$ from the original n disks, and pastes them to the new disk. Also based on a random hash function, Seo and Zimmermann [40] proposed finding a sequence of disks additions that minimized the data migration cost. On the other hand, the algorithm proposed in SCADDAR [13] moves a data block only if the destination disk is one of the newly added disks. This reduces migration significantly, but produces an unbalanced distribution after several expansions.

RUSH [19] and CRUSH [43] are the first methods with dedicated support for replication, and offer a probabilistically optimal data distribution with minimal migration. Their main drawback is that they require new capacity to be added in chunks and the number of disks in a chunk must be enough to hold a complete redundancy group. More recently, Miranda *et al.*'s Random Slicing [30] used a small table with information on insertion and removal operations to reduce the required randomness and deliver a uniform load distribution with minimal migration.

These randomized strategies are designed for object-based storage systems, and focus only on how blocks are mapped to disks, ignoring the inner data layout of each individual disk. In this regard, CRAID manages blocks rather than objects and is thus more similar to deterministic (and extensible) RAID algorithms. To our knowledge, however, it is the first strategy that uses information about data blocks to reduce the overhead of the upgrade process.

8 Conclusions

In this paper, we propose and evaluate CRAID, a self-optimizing RAID architecture that automatically reorganizes frequently used data in a dedicated *caching partition*. CRAID is designed to accelerate the upgrade process of traditional RAID architectures by limiting it to this partition, which contains the data that is currently important and on which certain QoS levels must be kept.

We analyze CRAID using seven real-world traces of different workloads and collected at several times in the last decade. Our analysis shows that CRAID is highly successful in predicting the data workload and its variations. Further, if an appropriate data distribution is used for the cache partition, CRAID optimizes the performance of read and write traffic due to the increased locality and sequentiality of frequently accessed data. Specifically, we show that it is possible to achieve a QoS competitive with an ideal RAID-5 or RAID+SSD array, by creating a small RAID-5 partition of at most 1.28% the available storage, regardless of the layout outside the partition.

In summary, we believe that CRAID is a novel approach to building RAID architectures that can offer reduced expansion times and I/O performance improvements. In addition, its ability to combine several layouts can serve as a starting point to design newer allocation strategies more conscious about data semantics.

Acknowledgments

We wish to thank anonymous reviewers and our shepherd C.S. Lui for their comments and suggestions for improvement. Special thanks go to André Brinkmann, María S. Pérez and BSC's SSRG team for insightful feedback that improved initial drafts significantly. This work was partially supported by the Spanish and Catalan Governments (grants SEV-2011-00067, TIN2012-34557, 2009-SGR-980), and EU's FP7/2007–2013 (grant RI-283493).

References

[1] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference* (2008), pp. 57–70.

[2] AKYÜREK, S., AND SALEM, K. Adaptive block rearrangement. *ACM Transactions on Computer Systems (TOCS) 13*, 2 (1995), 89–121.

[3] ARLITT, M., CHERKASOVA, L., DILLEY, J., FRIEDRICH, R., AND JIN, T. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Performance Evaluation Review 27*, 4 (2000), 3–11.

[4] ARTIAGA, E., AND MIRANDA, A. PRACE-2IP Deliverable D12.4. Performance Optimized Lustre. *INFRA-2011-2.3.5 – Second Implementation Phase of the European High Performance Computing (HPC) service PRACE* (2012).

[5] BHADKAMKAR, M., GUERRA, J., USECHE, L., BURNETT, S., LIPTAK, J., RANGASWAMI, R., AND HRISTIDIS, V. BORG: block-reORGanization for self-optimizing storage systems. In *Proccedings of the 7th conference on File and storage technologies* (2009), USENIX Association, pp. 183–196.

[6] BRINKMANN, A., SALZWEDEL, K., AND SCHEIDELER, C. Efficient, Distributed Data Placement Strategies for Storage Area Networks. In *Proceedings of the* 12^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (2000), pp. 119–128.

[7] BROWN, N. Online RAID-5 resizing. drivers/md/raid5. c in the source code of Linux Kernel 2.6. 18, 2006.

[8] BUCY, J., SCHINDLER, J., SCHLOSSER, S., AND GANGER, G. The DiskSim Simulation Environment Version 4.0 Reference Manual (CMU-PDL-08-101). *Parallel Data Laboratory* (2008), 26.

[9] CAO, P., AND IRANI, S. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems* (1997), vol. 193.

[10] CHEN, P., LEE, E., GIBSON, G., KATZ, R., AND PATTERSON, D. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR) 26*, 2 (1994), 145–185.

[11] CHEN, P. M., AND LEE, E. K. *Striping in a RAID level 5 disk array*, vol. 23. ACM, 1995.

[12] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), USENIX Association, pp. 203–216.

[13] GOEL, A., SHAHABI, C., YAO, S., AND ZIMMERMANN, R. SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In *Data Engineering, 2002. Proceedings. 18th International Conference on* (2002), IEEE, pp. 473–482.

[14] GÓMEZ, M., AND SANTONJA, V. Characterizing temporal locality in I/O workload. In *Proc. of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems* (2002).

[15] GONZALEZ, J., AND CORTES, T. Increasing the capacity of RAID5 by online gradual assimilation. In *Proceedings of the international workshop on Storage network architecture and parallel I/Os* (2004), ACM, pp. 17–24.

[16] HE, X., YANG, Q., AND ZHANG, M. A caching strategy to improve iSCSI performance. In *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on* (2002), IEEE, pp. 278–285.

[17] HETZLER, S. R., ET AL. Data storage array scaling method and system with minimal data movement. US Patent 8,239,622.

[18] HIDROBO, F., AND CORTES, T. Autonomic storage system based on automatic learning. In *High Performance Computing-HiPC 2004*. Springer, 2005, pp. 399–409.

[19] HONICKY, R., AND MILLER, E. L. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International* (2004), IEEE, p. 96.

[20] HSU, W., SMITH, A., AND YOUNG, H. The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems (TOCS) 23*, 4 (2005), 424–473.

[21] JIN, S., AND BESTAVROS, A. GreedyDual* Web caching algorithm: exploiting the two sources of temporal locality in Web request streams. *Computer Communications 24*, 2 (2001), 174–183.

[22] LEE, S., AND BAHN, H. Data allocation in MEMS-based mobile storage devices. *Consumer Electronics, IEEE Transactions on 52*, 2 (2006), 472–476.

[23] LEGG, C. Method of increasing the storage capacity of a level five RAID disk array by adding, in a single step, a new parity block and N–1 new data blocks which respectively reside in a new columns, where N is at least two, Dec. 7 1999. US Patent 6,000,010.

[24] LEUNG, A., PASUPATHY, S., GOODSON, G., AND MILLER, E. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference* (2008), pp. 213–226.

[25] LI, D., AND WANG, J. EERAID: energy efficient redundant and inexpensive disk array. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop* (2004), ACM, p. 29.

[26] LI, Z., CHEN, Z., SRINIVASAN, S., AND ZHOU, Y. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004), vol. 186, USENIX Association.

[27] LYMAN, P. How much information? 2003. *http://www.sims.berkeley.edu/research/projects/how-much-info-2003/* (2003).

[28] MEGIDDO, N., AND MODHA, D. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), pp. 115–130.

[29] MIRANDA, A., AND CORTES, T. Analyzing Long-Term Access Locality to Find Ways to Improve Distributed Storage Systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on* (2012), IEEE, pp. 544–553.

[30] MIRANDA, A., EFFERT, S., KANG, Y., MILLER, E. L., BRINKMANN, A., AND CORTES, T. Reliable and randomized data distribution strategies for large scale storage systems. In *High Performance Computing (HiPC), 2011 18th International Conference on* (2011), IEEE, pp. 1–10.

[31] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS) 4*, 3 (2008), 10.

[32] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating server storage to SSDs: analysis of tradeoffs. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), ACM, pp. 145–158.

[33] NIGHTINGALE, T., HU, Y., AND YANG, Q. The design and implementation of DCD device driver for UNIX. In *Proceedings of the 1999 USENIX Technical Conference* (1999), pp. 295–308.

[34] PARK, J., CHUN, H., BAHN, H., AND KOH, K. G-MST: A dynamic group-based scheduling algorithm for MEMS-based mobile storage devices. *Consumer Electronics, IEEE Transactions on 55*, 2 (2009), 570–575.

[35] PATTERSON, D., ET AL. A simple way to estimate the cost of downtime. In *Proc. 16th Systems Administration Conf.— LISA* (2002), pp. 185–8.

[36] PATTERSON, D., GIBSON, G., AND KATZ, R. *A case for redundant arrays of inexpensive disks (RAID)*, vol. 17. ACM, 1988.

[37] RUEMMLER, C., AND WILKES, J. Disk shuffling. Tech. rep., Technical Report HPL-91-156, Hewlett Packard Laboratories, 1991.

[38] RUEMMLER, C., AND WILKES, J. UNIX disk access patterns. In *Proceedings of the Winter 1993 USENIX Technical Conference* (1993), pp. 405–420.

[39] Seagate Cheetah 15K.5 FC product manual. `http://www.seagate.com/staticfiles/support/disc/manuals/enterprise/cheetah/15K.5/FC/100384772f.pdf` Last retrieved Sept. 9, 2013.

[40] SEO, B., AND ZIMMERMANN, R. Efficient disk replacement and data migration algorithms for large disk subsystems. *ACM Transactions on Storage (TOS) 1*, 3 (2005), 316–345.

[41] VERMA, A., KOLLER, R., USECHE, L., AND RANGASWAMI, R. SRCMap: energy proportional storage using dynamic consolidation. In *Proceedings of the 8th USENIX conference on File and storage technologies* (2010), USENIX Association, pp. 20–20.

[42] VONGSATHORN, P., AND CARSON, S. A system for adaptive disk rearrangement. *Software: Practice and Experience 20*, 3 (1990), 225–242.

[43] WEIL, S. A., BRANDT, S. A., MILLER, E. L., AND MALTZAHN, C. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (2006), ACM, p. 122.

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

[45] WONG, C. Minimizing expected head movement in one-dimensional and two-dimensional mass storage systems. *ACM Computing Surveys (CSUR) 12*, 2 (1980), 167–178.

[46] WONG, T., GANGER, G., WILKES, J., ET AL. *My Cache Or Yours?: Making Storage More Exclusive*. School of Computer Science, Carnegie Mellon University, 2000.

[47] WU, C., AND HE, X. Gsr: A global stripe-based redistribution approach to accelerate raid-5 scaling. In *Parallel Processing (ICPP), 2012 41st International Conference on* (2012), IEEE, pp. 460–469.

[48] YANG, Q., AND HU, Y. DCD—disk caching disk: A new approach for boosting I/O performance. In *Computer Architecture, 1996 23rd Annual International Symposium on* (1996), IEEE, pp. 169–169.

[49] ZHANG, G., SHU, J., XUE, W., AND ZHENG, W. SLAS: An efficient approach to scaling round-robin striped volumes. *ACM Transactions on Storage (TOS) 3*, 1 (2007), 3.

[50] ZHENG, W., AND ZHANG, G. FastScale: accelerate RAID scaling by minimizing data migration. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)* (2011).

[51] ZHU, Q., CHEN, Z., TAN, L., ZHOU, Y., KEETON, K., AND WILKES, J. Hibernator: helping disk arrays sleep through the winter. In *ACM SIGOPS Operating Systems Review* (2005), vol. 39, ACM, pp. 177–190.

STAIR Codes: A General Family of Erasure Codes for Tolerating Device and Sector Failures in Practical Storage Systems

Mingqiang Li and Patrick P. C. Lee
The Chinese University of Hong Kong
mingqiangli.cn@gmail.com, pclee@cse.cuhk.edu.hk

Abstract

Practical storage systems often adopt erasure codes to tolerate device failures and sector failures, both of which are prevalent in the field. However, traditional erasure codes employ device-level redundancy to protect against sector failures, and hence incur significant space overhead. Recent sector-disk (SD) codes are available only for limited configurations due to the relatively strict assumption on the coverage of sector failures. By making a relaxed but practical assumption, we construct a general family of erasure codes called *STAIR codes*, which efficiently and provably tolerate both device and sector failures without any restriction on the size of a storage array and the numbers of tolerable device failures and sector failures. We propose the *upstairs encoding* and *downstairs encoding* methods, which provide complementary performance advantages for different configurations. We conduct extensive experiments to justify the practicality of STAIR codes in terms of space saving, encoding/decoding speed, and update cost. We demonstrate that STAIR codes not only improve space efficiency over traditional erasure codes, but also provide better computational efficiency than SD codes based on our special code construction.

1 Introduction

Mainstream disk drives are known to be susceptible to both *device failures* [25, 37] and *sector failures* [1, 36]: a device failure implies the loss of all data in the failed device, while a sector failure implies the data loss in a particular disk sector. In particular, sector failures are of practical concern not only in disk drives, but also in emerging solid-state drives as they often appear as worn-out blocks after frequent program/erase cycles [8, 14, 15, 43]. In the face of device and sector failures, practical storage systems often adopt *erasure codes* to provide data redundancy [32]. However, existing erasure codes often build on tolerating device failures and provide device-level redundancy only. To tolerate additional sector failures, an erasure code must be constructed with extra parity disks. A representative example is RAID-6, which uses two parity disks to tolerate one device failure together with one sector failure in another non-failed device [21, 39]. If the sector failures can span a number of devices, the same number of parity disks must be provisioned. Clearly, dedicating an entire parity disk for tolerating a sector failure is too extravagant.

To tolerate both device and sector failures in a space-efficient manner, sector-disk (SD) codes [27, 28] and the earlier PMDS codes [5] (which are a subset of SD codes) have recently been proposed. Their idea is to introduce parity sectors, instead of entire parity disks, to tolerate a given number of sector failures. However, the constructions of SD codes are known only for limited configurations (e.g., the number of tolerable sector failures is no more than three), and some of the known constructions rely on exhaustive searches [6, 27, 28]. An open issue is to provide a general construction of erasure codes that can efficiently tolerate both device and sector failures without any restriction on the size of a storage array, the number of tolerable device failures, or the number of tolerable sector failures.

In this paper, we make the first attempt to develop such a generalization, which we believe is of great theoretical and practical interest to provide space-efficient fault tolerance for today's storage systems. After carefully examining the assumption of SD codes on failure coverage, we find that although SD codes have relaxed the assumption of the earlier PMDS codes to comply with how most storage systems really fail, the assumption remains too strict. By reasonably relaxing the assumption of SD codes on sector failure coverage, we construct a general family of erasure codes called *STAIR codes*, which efficiently tolerate both device and sector failures.

Specifically, SD codes devote s sectors per stripe to coding, and tolerate the failure of any s sectors per stripe. We relax this assumption in STAIR codes by limiting the number of devices that may simultaneously contain sector failures, and by limiting the number of simultaneous sector failures per device. The new assumption of STAIR codes is based on the strong locality of sector failures found in practice: sector failures tend to come in short bursts, and are concentrated in small address space [1, 36]. Consequently, as shown in §2, STAIR codes are constructed to protect the sector failure coverage defined by a vector $\mathbf{e}$, rather than all combinations of s sector failures.

With the relaxed assumption, the construction of STAIR codes can be based on existing erasure codes. For example, STAIR codes can build on Reed-Solomon codes (including standard Reed-Solomon codes [26, 30, 34] and Cauchy Reed-Solomon codes [7, 33]), which have no restriction on code length and fault tolerance.

We first define the notation and elaborate how the sector failure coverage is formulated for STAIR codes in §2. Then the paper makes the following contributions:

- We present a baseline construction of STAIR codes. Its idea is to run two orthogonal encoding phases based on Reed-Solomon codes. See §3.
- We propose an *upstairs decoding* method, which systematically reconstructs the lost data due to both device and sector failures. The proof of fault tolerance of STAIR codes follows immediately from the decoding method. See §4.
- Inspired by upstairs decoding, we extend the construction of STAIR codes to regularize the code structure. We propose two encoding methods: *upstairs encoding* and *downstairs encoding*, both of which reuse computed parity results in subsequent encoding. The two encoding methods provide complementary performance advantages for different configuration parameters. See §5.
- We extensively evaluate STAIR codes in terms of space saving, encoding/decoding speed, and update cost. We show that STAIR codes achieve significantly higher encoding/decoding speed than SD codes through parity reuse. Most importantly, we show the versatility of STAIR codes in supporting any size of a storage array, any number of tolerable device failures, and any number of tolerable sector failures. See §6.

We review related work in §7, and conclude in §8.

2 Preliminaries

We consider a storage system with n devices, each of which has its storage space logically segmented into a sequence of continuous *chunks* (also called *strips*) of the same size. We group each of the n chunks at the same position of each device into a *stripe*, as depicted in Figure 1. Each chunk is composed of r sectors (or blocks). Thus, we can view the stripe as a $r \times n$ array of sectors. Using coding theory terminology, we refer to each sector as a *symbol*. Each stripe is independently protected by an erasure code for fault tolerance, so our discussion focuses on a single stripe.

Storage systems are subject to both device and sector failures. A device failure can be mapped to the failure of an entire chunk of a stripe. We assume that the stripe can tolerate at most m ($< n$) chunk failures, in which all symbols are lost. In addition to device failures, we

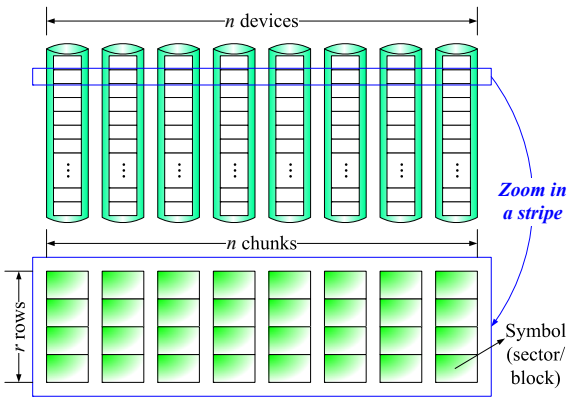


Figure 1: A stripe for $n = 8$ and $r = 4$.

assume that sector failures can occur in the remaining $n - m$ devices. Each sector failure is mapped to a lost symbol in the stripe. Suppose that besides the m failed chunks, the stripe can tolerate sector failures in at most m' ($\leq n - m$) remaining chunks, each of which has a maximum number of sector failures defined by a vector $\mathbf{e} = (e_0, e_1, \cdots, e_{m'-1})$. Without loss of generality, we arrange the elements of $\mathbf{e}$ in monotonically increasing order (i.e., $e_0 \leq e_1 \leq \cdots \leq e_{m'-1}$). For example, suppose that sector failures can only simultaneously appear in at most three chunks (i.e., $m' = 3$), among which at most one chunk has two sector failures and the remaining have one sector failure each. Then, we can express $\mathbf{e} = (1, 1, 2)$. Also, let $s = \sum_{i=0}^{m'-1} e_i$ be the total number of sector failures defined by $\mathbf{e}$. Our study assumes that the configuration parameters n, r, m, and $\mathbf{e}$ (which then determines m' and s) are the inputs selected by system practitioners for the erasure code construction.

Erasure codes have been used by practical storage systems to protect against data loss [32]. We focus on a class of erasure codes with optimal storage efficiency called *maximum distance separable (MDS)* codes, which are defined by two parameters η and κ ($< \eta$). We define an (η, κ)-code as an MDS code that transforms κ symbols into η symbols collectively called a *codeword* (this operation is called *encoding*), such that any κ of the η symbols can be used to recover the original κ uncoded symbols (this operation is called *decoding*). Each codeword is encoded from κ uncoded symbols by multiplying a row vector of the κ uncoded symbols with a $\kappa \times \eta$ *generator matrix* of coefficients based on Galois Field arithmetic. We assume that the (η, κ)-code is *systematic*, meaning that the κ uncoded symbols are kept in the codeword. We refer to the κ uncoded symbols as *data* symbols, and the $\eta - \kappa$ coded symbols as *parity* symbols. We use systematic MDS codes as the building blocks of STAIR codes. Examples of such codes are standard Reed-Solomon codes [26, 30, 34] and Cauchy Reed-Solomon codes [7, 33].

Given parameters n, r, m, and $\mathbf{e}$ (and hence m' and s), our goal is to construct a STAIR code that tolerates both m failed chunks and s sector failures in the remaining $n-m$ chunks defined by $\mathbf{e}$. Note that some special cases of $\mathbf{e}$ have the following physical meanings:

- If $\mathbf{e} = (1)$, the corresponding STAIR code is equivalent to a PMDS/SD code with $s = 1$ [5, 27, 28]. In fact, the STAIR code is a new construction of such a PMDS/SD code.
- If $\mathbf{e} = (r)$, the corresponding STAIR code has the same function as a systematic $(n, n-m-1)$-code.
- If $\mathbf{e} = (\epsilon, \epsilon, \cdots, \epsilon)$ with $m' = n-m$ and some constant $\epsilon < r$, the corresponding STAIR code has the same function as an intra-device redundancy (IDR) scheme [10, 11, 36] that adopts a systematic $(r, r-\epsilon)$-code.

We argue that STAIR codes can be configured to provide more general protection than SD codes [6, 27, 28]. One major use case of STAIR codes is to protect against bursts of contiguous sector failures [1, 36]. Let β be the maximum length of a sector failure burst found in a chunk. Then we should set $\mathbf{e}$ with its largest element $e_{m'-1} = \beta$. For example, when $\beta = 2$, we may set $\mathbf{e}$ as our previous example $\mathbf{e} = (1, 1, 2)$, or a weaker and lower-cost $\mathbf{e} = (1, 2)$. In some extreme cases, some disk models may have longer sector failure bursts (e.g., with $\beta > 3$) [36]. Take $\beta = 4$ for example. Then we can define $\mathbf{e} = (1, 4)$, so that the corresponding STAIR code can tolerate a burst of four sector failures in one chunk together with an additional sector failure in another chunk. In contrast, such an extreme case cannot be handled by SD codes, whose current construction can only tolerate at most three sector failures in a stripe [6, 27, 28]. Thus, although the numbers of device and sector failures (i.e., m and s, respectively) are often small in practice, STAIR codes support a more general coverage of device and sector failures, especially for extreme cases.

STAIR codes also provide more space-efficient protection than the IDR scheme [10, 11, 36]. To protect against a burst of β sector failures in *any* data chunk of a stripe, the IDR scheme requires β additional redundant sectors in each of the $n-m$ data chunks. This is equivalent to setting $\mathbf{e} = (\beta, \beta, \cdots, \beta)$ with $m' = n-m$ in STAIR codes. In contrast, the general construction of STAIR codes allows a more flexible definition of $\mathbf{e}$, where m' can be less than $n-m$, and all elements of $\mathbf{e}$ except the largest element $e_{m'-1}$ can be less than β. For example, to protect against a burst of $\beta = 4$ sector failures for $n = 8$ and $m = 2$ (i.e., a RAID-6 system with eight devices), the IDR scheme introduces a total of $4 \times 6 = 24$ redundant sectors per stripe; if we define $\mathbf{e} = (1, 4)$ in STAIR codes as above, then we only introduce five redundant sectors per stripe.

3 Baseline Encoding

For general configuration parameters n, r, m, and $\mathbf{e}$, the main idea of STAIR encoding is to run two orthogonal encoding phases using two systematic MDS codes. First, we encode the data symbols using one code and obtain two types of parity symbols: *row parity symbols*, which protect against device failures, and *intermediate parity symbols*, which will then be encoded using another code to obtain *global parity symbols*, which protect against sector failures. In the following, we elaborate the encoding of STAIR codes and justify our naming convention.

We label different types of symbols for STAIR codes as follows. Figure 2 shows the layout of an exemplary stripe of a STAIR code for $n = 8$, $r = 4$, $m = 2$, and $\mathbf{e} = (1, 1, 2)$ (i.e., $m' = 3$ and $s = 4$). A stripe is composed of $n-m$ data chunks and m row parity chunks. We also assume that there are m' intermediate parity chunks and s global parity symbols outside the stripe. Let $d_{i,j}$, $p_{i,k}$, $p'_{i,l}$, and $g_{h,l}$ denote a data symbol, a row parity symbol, an intermediate parity symbol, and a global parity symbol, respectively, where $0 \le i \le r-1$, $0 \le j \le n-m-1$, $0 \le k \le m-1$, $0 \le l \le m'-1$, and $0 \le h \le e_l - 1$.

Figure 2 depicts the steps of the two orthogonal encoding phases of STAIR codes. In the first encoding phase, we use an $(n+m', n-m)$-code denoted by $\mathcal{C}_{row}$ (which is an (11,6)-code in Figure 2). We encode via $\mathcal{C}_{row}$ each row of $n-m$ data symbols to obtain m row parity symbols and m' intermediate parity symbols in the same row:

Phase 1: For $i = 0, 1, \cdots, r-1$,

$$d_{i,0}, d_{i,1}, \cdots, d_{i,n-m-1} \overset{\mathcal{C}_{row}}{\Longrightarrow} p_{i,0}, p_{i,1}, \cdots, p_{i,m-1}, p'_{i,0}, p'_{i,1}, \cdots, p'_{i,m'-1},$$

where $\overset{\mathcal{C}}{\Longrightarrow}$ describes that the input symbols on the left are used to generate the output symbols on the right using some code $\mathcal{C}$. We call each $p_{i,k}$ a "row" parity symbol since it is only encoded from the same row of data symbols in the stripe, and we call each $p'_{i,l}$ an "intermediate" parity symbol since it is not actually stored but is used in the second encoding phase only.

In the second encoding phase, we use a $(r+e_{m'-1}, r)$-code denoted by $\mathcal{C}_{col}$ (which is a (6,4)-code in Figure 2). We encode via $\mathcal{C}_{col}$ each chunk of r intermediate parity symbols to obtain at most $e_{m'-1}$ global parity symbols:

Phase 2: For $l = 0, 1, \cdots, m'-1$,

$$p'_{0,l}, p'_{1,l}, \cdots, p'_{r-1,l} \overset{\mathcal{C}_{col}}{\Longrightarrow} \overbrace{g_{0,l}, g_{1,l}, \cdots, g_{e_l-1,l}, *, \cdots, *}^{e_{m'-1}},$$

where "$*$" represents a "dummy" global parity symbol that will not be generated when $e_l < e_{m'-1}$, and we only need to compute the "real" global parity symbols $g_{0,l}, g_{1,l}, \cdots, g_{e_l-1,l}$. The intermediate parity symbols will be discarded after this encoding phase. Note that

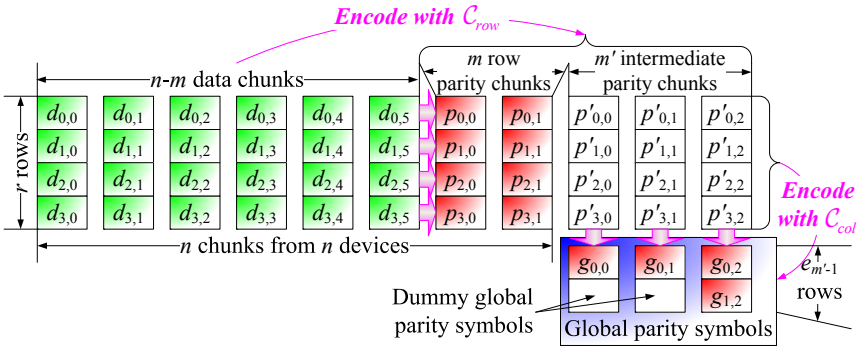


Figure 2: Exemplary configuration: a STAIR code stripe for $n = 8$, $r = 4$, $m = 2$, and $\mathbf{e} = (1, 1, 2)$ (i.e., $m' = 3$ and $s = 4$). Throughout this paper, we use this configuration to explain the operations of STAIR codes.

each $g_{h,l}$ is in essence encoded from all the data symbols in the stripe, and thus we call it a "global" parity symbol.

We point out that $\mathcal{C}_{row}$ and $\mathcal{C}_{col}$ can be any systematic MDS codes. In this work, we implement both $\mathcal{C}_{row}$ and $\mathcal{C}_{col}$ using Cauchy Reed-Solomon codes [7, 33], which have no restriction on code length and fault tolerance.

From Figure 2, we see that the logical layout of global parity symbols looks like a stair. This is why we name this family of erasure codes *STAIR codes*.

In the following discussion, we use the exemplary configuration in Figure 2 to explain the detailed operations of STAIR codes. To simplify our discussion, we first assume that the global parity symbols are kept outside a stripe and are always available for ensuring fault tolerance. In §5, we will extend the encoding of STAIR codes when the global parity symbols are kept inside the stripe and are subject to both device and sector failures.

4 Upstairs Decoding

In this section, we justify the fault tolerance of STAIR codes defined by m and $\mathbf{e}$. We introduce an *upstairs decoding* method that systematically recovers the lost symbols when both device and sector failures occur.

4.1 Homomorphic Property

The proof of fault tolerance of STAIR codes builds on the concept of a *canonical stripe*, which is constructed by augmenting the existing stripe with additional *virtual parity symbols*. To illustrate, Figure 3 depicts how we augment the stripe of Figure 2 into a canonical stripe. Let $d^*_{h,j}$ and $p^*_{h,k}$ denote the virtual parity symbols encoded with $\mathcal{C}_{col}$ from a data chunk and a row parity chunk, respectively, where $0 \le j \le n-m-1$, $0 \le k \le m-1$, and $0 \le h \le e_{m'-1}-1$. Specifically, we use $\mathcal{C}_{col}$ to generate virtual parity symbols from the data and row parity chunks as follows:

For $j = 0, 1, \cdots, n-m-1$,

$$d_{0,j}, d_{1,j}, \cdots, d_{r-1,j} \overset{\mathcal{C}_{col}}{\Longrightarrow} d^*_{0,j}, d^*_{1,j}, \cdots, d^*_{e_{m'-1}-1,j};$$

and for $k = 0, 1, \cdots, m-1$,

$$p_{0,k}, p_{1,k}, \cdots, p_{r-1,k} \overset{\mathcal{C}_{col}}{\Longrightarrow} p^*_{0,k}, p^*_{1,k}, \cdots, p^*_{e_{m'-1}-1,k}.$$

The virtual parity symbols $d^*_{h,j}$'s and $p^*_{h,k}$'s, along with the real and dummy global parity symbols, form $e_{m'-1}$ augmented rows of $n + m'$ symbols. To make our discussion simpler, we number the rows and columns of the canonical stripe from 0 to $r + e_{m'-1} - 1$ and from 0 to $n + m' - 1$, respectively, as shown in Figure 3.

Referring to Figure 3, we know that the upper r rows of $n + m'$ symbols are codewords of $\mathcal{C}_{row}$. We argue that each of the lower $e_{m'-1}$ augmented rows is in fact also a codeword of $\mathcal{C}_{row}$. We call this the *homomorphic property*, since the encoding of each chunk in the column direction preserves the coding structure in the row direction. We formally prove the homomorphic property in Appendix. We use this property to prove the fault tolerance of STAIR codes.

4.2 Proof of Fault Tolerance

We prove that for a STAIR code with configuration parameters n, r, m, and $\mathbf{e}$, as long as the failure pattern is within the failure coverage defined by m and $\mathbf{e}$, the corresponding lost symbols can always be recovered (or decoded). In addition, we present an *upstairs decoding* method, which systematically recovers the lost symbols for STAIR codes.

For a stripe of the STAIR code, we consider the worst-case recoverable failure scenario where there are m failed chunks (due to device failures) and m' additional chunks that have $e_0, e_1, \cdots, e_{m'-1}$ lost symbols (due to sector failures), where $0 < e_0 \le e_1 \le \cdots \le e_{m'-1}$. We prove that all the m' chunks with sector failures can be recovered with global parity symbols. In particular, we show that these m' chunks can be recovered in the order of $e_0, e_1, \cdots, e_{m'-1}$. Finally, the m failed chunks due to device failures can be recovered with row parity chunks.

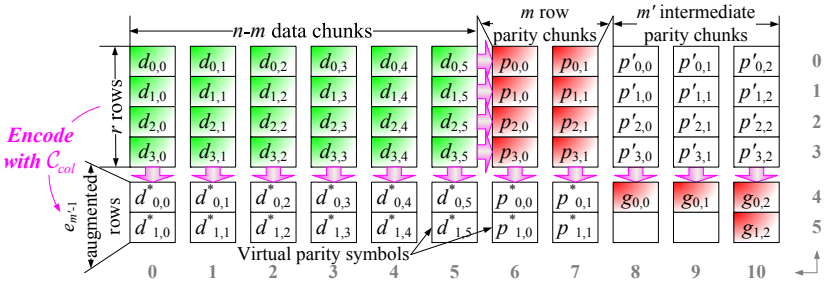


Figure 3: A canonical stripe augmented from the stripe in Figure 2. The rows and columns are labeled from 0 to 5 and 0 to 10, respectively, for ease of presentation.

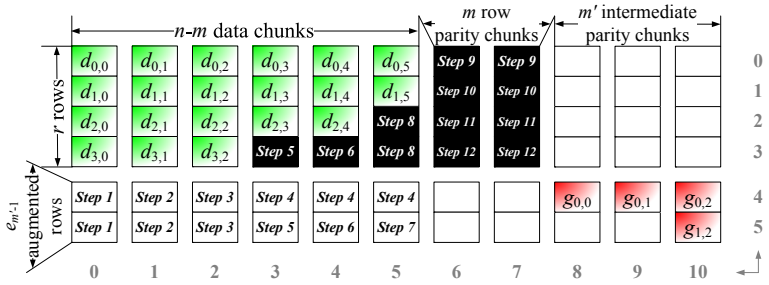


Figure 4: Upstairs decoding based on the canonical stripe in Figure 3.

4.2.1 Example

We demonstrate via our exemplary configuration how we recover the lost data due to both device and sector failures. Figure 4 shows the sequence of our decoding steps. Without loss of generality, we logically assign the column identities such that the m' chunks with sector failures are in Columns $n-m-m'$ to $n-m-1$, with e_0, e_1, $\cdots$, $e_{m'-1}$ lost symbols, respectively, and the m failed chunks are in Columns $n-m$ to $n-1$. Also, the sector failures all occur in the bottom of the data chunks. Thus, the lost symbols form a stair, as shown in Figure 4.

The main idea of upstairs decoding is to recover the lost symbols from left to right and bottom to top. First, we see that there are $n-m-m'=3$ good chunks (i.e., Columns 0-2) without any sector failure. We encode via $\mathcal{C}_{col}$ (which is a (6,4)-code) each such good chunk to obtain $e_{m'-1}=2$ virtual parity symbols (Steps 1-3). In Row 4, there are now six available symbols. Thus, all the unavailable symbols in this row can be recovered using $\mathcal{C}_{row}$ (which is a (11,6)-code) due to the homomorphic property (Step 4). Note that we only need to recover the $m'=3$ symbols that will later be used to recover sector failures. Column 3 (with $e_0=1$ sector failure) now has four available symbols. Thus, we can recover one lost symbol and one virtual parity symbol using $\mathcal{C}_{col}$ (Step 5). Similarly, we repeat the decoding for Column 4 (with $e_1=1$ sector failure) (Step 6). We see that Row 5 now contains six available symbols, so we can recover one unavailable virtual parity symbol (Step 7). Then Column 5 (with $e_2=2$ sector failures) now has four available symbols, so we can recover two lost symbols (Step 8). Now all chunks with sector failures are recovered. Finally, we recover the $m=2$ lost chunks row by row using $\mathcal{C}_{row}$ (Steps 9-12). Table 1 lists the detailed decoding steps of our example in Figure 4.

Steps	Detailed Descriptions
1	$d_{0,0}, d_{1,0}, d_{2,0}, d_{3,0} \Rightarrow d^*_{0,0}, d^*_{1,0}$
2	$d_{0,1}, d_{1,1}, d_{2,1}, d_{3,1} \Rightarrow d^*_{0,1}, d^*_{1,1}$
3	$d_{0,2}, d_{1,2}, d_{2,2}, d_{3,2} \Rightarrow d^*_{0,2}, d^*_{1,2}$
4	$d^*_{0,0}, d^*_{0,1}, d^*_{0,2}, g_{0,0}, g_{0,1}, g_{0,2} \Rightarrow d^*_{0,3}, d^*_{0,4}, d^*_{0,5}$
5	$d_{0,3}, d_{1,3}, d_{2,3}, d^*_{0,3} \Rightarrow d_{3,3}, d^*_{1,3}$
6	$d_{0,4}, d_{1,4}, d_{2,4}, d^*_{0,4} \Rightarrow d_{3,4}, d^*_{1,4}$
7	$d^*_{1,0}, d^*_{1,1}, d^*_{1,2}, d^*_{1,3}, d^*_{1,4}, g_{1,2} \Rightarrow d^*_{1,5}$
8	$d_{0,5}, d_{1,5}, d^*_{0,5}, d^*_{1,5} \Rightarrow d_{2,5}, d_{3,5}$
9	$d_{0,0}, d_{0,1}, d_{0,2}, d_{0,3}, d_{0,4}, d_{0,5} \Rightarrow p_{0,1}, p_{0,2}$
10	$d_{1,0}, d_{1,1}, d_{1,2}, d_{1,3}, d_{1,4}, d_{1,5} \Rightarrow p_{1,1}, p_{1,2}$
11	$d_{2,0}, d_{2,1}, d_{2,2}, d_{2,3}, d_{2,4}, d_{2,5} \Rightarrow p_{2,1}, p_{2,2}$
12	$d_{3,0}, d_{3,1}, d_{3,2}, d_{3,3}, d_{3,4}, d_{3,5} \Rightarrow p_{3,1}, p_{3,2}$

Table 1: Upstairs decoding: detailed steps for the example in Figure 4. Steps 4, 7, and 9-12 use $\mathcal{C}_{row}$, while Steps 1-3, 5-6, and 8 use $\mathcal{C}_{col}$.

4.2.2 General Case

We now generalize the steps of upstairs decoding.

(1) *Decoding of the chunk with e_0 sector failures:* It is clear that there are $n-(m+m')$ good chunks without any sector failure in the stripe. We use $\mathcal{C}_{col}$ to encode each such good chunk to obtain $e_{m'-1}$ virtual parity symbols. Then each of the first e_0 augmented rows must now have $n-m$ available symbols: $n-(m+m')$

virtual parity symbols that have just been encoded and m' global parity symbols. Since an augmented row is a codeword of $\mathcal{C}_{row}$ due to the homomorphic property, all the unavailable symbols in this row can be recovered using $\mathcal{C}_{row}$. Then, for the column with e_0 sector failures, it now has r available symbols: $r - e_0$ good symbols and e_0 virtual parity symbols that have just been recovered. Thus, we can recover the e_0 sector failures as well as the $e_{m'-1} - e_0$ unavailable virtual parity symbols using $\mathcal{C}_{col}$.

(2) *Decoding of the chunk with e_i sector failures ($1 \leq i \leq m' - 1$):* If $e_i = e_{i-1}$, we repeat the decoding for the chunk with e_{i-1} sector failures. Otherwise, if $e_i > e_{i-1}$, each of the next $e_i - e_{i-1}$ augmented rows now has $n - m$ available symbols: $n - (m + m')$ virtual parity symbols that are first recovered from the good chunks, i virtual parity symbols that are recovered while the sector failures are recovered, and $m' - i$ global parity symbols. Thus, all the unavailable virtual parity symbols in these $e_i - e_{i-1}$ augmented rows can be recovered. Then the column with e_i sector failures now has r available symbols: $r - e_i$ good symbols and e_i virtual parity symbols that have been recovered. This column can then be recovered using $\mathcal{C}_{col}$. We repeat this process until all the m' chunks with sector failures are recovered.

(3) *Decoding of the m failed chunks:* After all the m' chunks with sector failures are recovered, the m failed chunks can be recovered row by row using $\mathcal{C}_{row}$.

4.3 Decoding in Practice

In §4.2, we describe an upstairs decoding method for the worst case. In practice, we often have fewer lost symbols than the worst case defined by m and $\mathbf{e}$. To achieve efficient decoding, our idea is to recover as many lost symbols as possible via row parity symbols. The reason is that such decoding is local and involves only the symbols of the same row, while decoding via global parity symbols involves almost all data symbols within the stripe. In our implementation, we first locally recover any lost symbols using row parity symbols whenever possible. Then, for each chunk that still contains lost symbols, we count the number of its remaining lost symbols. Next, we globally recover the lost symbols with global parity symbols using upstairs decoding as described in §4.2, except those in the m chunks that have the most lost symbols. These m chunks can be finally recovered via row parity symbols after all other lost symbols have been recovered.

5 Extended Encoding: Relocating Global Parity Symbols Inside a Stripe

We thus far assume that there are always s available global parity symbols that are kept outside a stripe. However, to maintain the regularity of the code structure and to avoid provisioning extra devices for keeping the global parity symbols, it is desirable to keep all global parity symbols inside a stripe. The idea is that in each stripe, we store the global parity symbols in some sectors that originally store the data symbols. A challenge is that such *inside global parity symbols* are also subject to both device and sector failures, so we must maintain their fault tolerance during encoding. In this section, we propose two encoding methods, namely *upstairs encoding* and *downstairs encoding*, which support the construction of inside global parity symbols, while preserving the homomorphic property and hence the fault tolerance of STAIR codes. These two encoding methods produce the same values for parity symbols, but differ in computational complexities for different configurations. We show how to deduce parity relations from the two encoding methods, and also show that the two encoding methods have complementary performance advantages for different configurations.

5.1 Two New Encoding Methods

5.1.1 Upstairs Encoding

We let $\hat{g}_{h,l}$ ($0 \leq l \leq m' - 1$ and $0 \leq h \leq e_l - 1$) be an inside global parity symbol. Figure 5 illustrates how we place the inside global parity symbols. Without loss of generality, we place them at the bottom of the rightmost data chunks, following the stair layout. Specifically, we choose the $m' = 3$ rightmost data chunks in Columns 3-5 and place $e_0 = 1$, $e_1 = 1$, and $e_2 = 2$ global parity symbols at the bottom of these data chunks, respectively. That is, the original data symbols $d_{3,3}$, $d_{3,4}$, $d_{2,5}$, and $d_{3,5}$ are now replaced by the inside global parity symbols $\hat{g}_{0,0}$, $\hat{g}_{0,1}$, $\hat{g}_{0,2}$, and $\hat{g}_{1,2}$, respectively.

To obtain the inside global parity symbols, we extend the upstairs decoding method in §4.2 and propose a recovery-based encoding approach called *upstairs encoding*. We first set all the outside global parity symbols to be zero (see Figure 5). Then we treat all $m = 2$ row parity chunks and all $s = 4$ inside global parity symbols as lost chunks and lost sectors, respectively. Now we "recover" all inside global parity symbols, followed by the $m = 2$ row parity chunks, using the upstairs decoding method in §4.2. Since all outside global parity symbols are set to be zero, we need not store them. The homomorphic property, and hence the fault tolerance property, remain the same as discussed in §4. Thus, in failure mode, we can still use upstairs decoding to reconstruct lost symbols. We call this encoding method "upstairs encoding" because the parity symbols are encoded from bottom to top as described in §4.2.

5.1.2 Downstairs Encoding

In addition to upstairs encoding, we present a different encoding method called *downstairs encoding*, in which we generate parity symbols from top to bottom and right to left. We illustrate the idea in Figure 6, which depicts

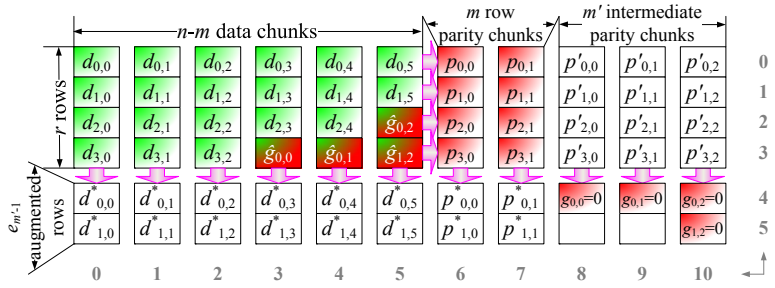


Figure 5: Upstairs encoding: we set outside global parity symbols to be zero and reconstruct the inside global parity symbols using upstairs decoding (see §4.2).

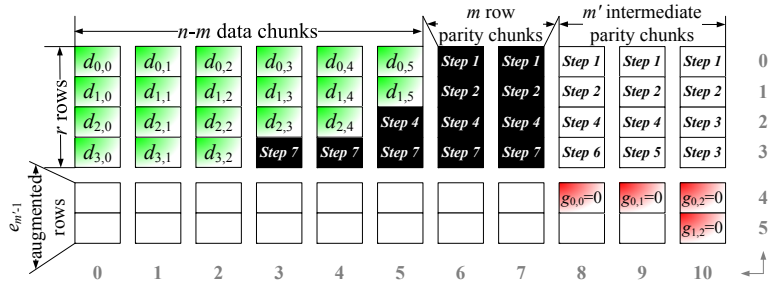


Figure 6: Downstairs encoding: we compute the parity symbols from top to bottom and right to left.

the sequence of generating parity symbols. We still set the outside global parity symbols to be zero. First, we encode via $\mathcal{C}_{row}$ the $n - m = 6$ data symbols in each of the first $r - e_{m'-1} = 2$ rows (i.e., Rows 0 and 1) and generate $m + m' = 5$ parity symbols (including two row parity symbols and three intermediate parity symbols) (Steps 1-2). The rightmost column (i.e., Column 10) now has $r = 4$ available symbols, including the two intermediate parity symbols that are just encoded and two zeroed outside global parity symbols. Thus, we can recover $e_{m'-1} = 2$ intermediate parity symbols using $\mathcal{C}_{col}$ (Step 3). We can generate $m + m' = 5$ parity symbols (including one inside global parity symbol, two row parity symbols, and two intermediate parity symbols) for Row 2 using $\mathcal{C}_{row}$ (Step 4), followed by $e_{m'-2} = 1$ and $e_{m'-3} = 1$ intermediate parity symbols in Columns 9 and 8 using $\mathcal{C}_{col}$, respectively (Steps 5-6). Finally, we obtain the remaining $m + m' = 5$ parity symbols (including three global parity symbols and two row parity symbols) for Row 3 using $\mathcal{C}_{row}$ (Step 7). Table 2 shows the detailed steps of downstairs encoding for the example in Figure 6.

In general, we start with encoding via $\mathcal{C}_{row}$ the rows from top to bottom. In each row, we generate $m + m'$ symbols. When no more rows can be encoded because of insufficient available symbols, we encode via $\mathcal{C}_{col}$ the columns from right to left to obtain new intermediate parity symbols (initially, we obtain $e_{m'-1}$ symbols, followed by $e_{m'-2}$ symbols, and so on). We alternately encode rows and columns until all parity symbols are

Steps	Detailed Descriptions
1	$d_{0,0}, d_{0,1}, d_{0,2}, d_{0,3}, d_{0,4}, d_{0,5} \Rightarrow p_{0,0}, p_{0,1}, p'_{0,0}, p'_{0,1}, p'_{0,2}$
2	$d_{1,0}, d_{1,1}, d_{1,2}, d_{1,3}, d_{1,4}, d_{1,5} \Rightarrow p_{1,0}, p_{1,1}, p'_{1,0}, p'_{1,1}, p'_{1,2}$
3	$p'_{0,2}, p'_{1,2}, g_{0,2} = 0, g_{1,2} = 0 \Rightarrow p'_{2,2}, p'_{3,2}$
4	$d_{2,0}, d_{2,1}, d_{2,2}, d_{2,3}, d_{2,4}, p'_{2,2} \Rightarrow \hat{g}_{0,2}, p_{2,0}, p_{2,1}, p'_{2,0}, p'_{2,1}$
5	$p'_{0,1}, p'_{1,1}, p'_{2,1}, g_{0,1} = 0 \Rightarrow p'_{3,1}$
6	$p'_{0,0}, p'_{1,0}, p'_{2,0}, g_{0,0} = 0 \Rightarrow p'_{3,0}$
7	$d_{3,0}, d_{3,1}, d_{3,2}, p'_{3,0}, p'_{3,1}, p'_{3,2} \Rightarrow \hat{g}_{0,0}, \hat{g}_{0,1}, \hat{g}_{1,2}, p_{3,0}, p_{3,1}$

Table 2: Downstairs decoding: detailed steps for the example in Figure 6. Steps 1-2, 4, and 7 use $\mathcal{C}_{row}$, while Steps 3 and 5-6 use $\mathcal{C}_{col}$.

formed. We can generalize the steps as in §4.2.2, but we omit the details in the interest of space.

It is important to note that the downstairs encoding method cannot be generalized for decoding lost symbols. For example, referring to our exemplary configuration, we consider a worst-case recoverable failure scenario in which both row parity chunks are entirely failed, and the data symbols $d_{0,3}$, $d_{1,4}$, $d_{2,2}$, and $d_{3,2}$ are lost. In this case, we cannot recover the lost symbols in the top row first, but instead we must resort to upstairs decoding as described in §4.2. Upstairs decoding works because we limit the maximum number of chunks with lost symbols (i.e., at most $m+m'$). This enables us to first recover the leftmost virtual parity symbols of the augmented rows first and gradually reconstruct lost symbols. On the other

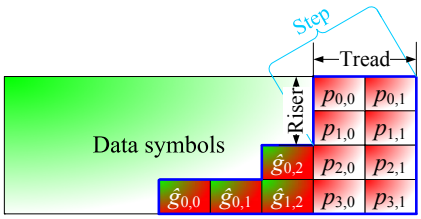


Figure 7: A stair step with a tread and a riser.

hand, we do not limit the number of rows with lost symbols in our configuration, so the downstairs method cannot be used for general decoding.

5.1.3 Discussion

Note that both upstairs and downstairs encoding methods always generate the *same* values for all parity symbols, since both of them preserve the homomorphic property, fix the outside global parity symbols to be zero, and use the same schemes $\mathcal{C}_{row}$ and $\mathcal{C}_{col}$ for encoding.

Also, both of them reuse parity symbols in the intermediate steps to generate additional parity symbols in subsequent steps. On the other hand, they differ in encoding complexity, due to the different ways of reusing the parity symbols. We analyze this in §5.3.

5.2 Uneven Parity Relations

Before relocating the global parity symbols inside a stripe, each data symbol contributes to m row parity symbols and all s outside global parity symbols. However, after relocation, the parity relations become uneven. That is, some row parity symbols are also contributed by the data symbols in other rows, while some inside global parity symbols are contributed by only a subset of data symbols in the stripe. Here, we discuss the uneven parity relations of STAIR codes so as to better understand the encoding and update performance of STAIR codes in subsequent analysis.

To analyze how exactly each parity symbol is generated, we revisit both upstairs and downstairs encoding methods. Recall that the row parity symbols and the inside global parity symbols are arranged in the form of stair steps, each of which is composed of a *tread* (i.e., the horizontal portion of a step) and a *riser* (i.e., the vertical portion of a step), as shown in Figure 7. If upstairs encoding is used, then from Figure 4, the encoding of each parity symbol does not involve any data symbol on its right. Also, among the columns spanned by the same tread, the encoding of parity symbols in each column does not involve any data symbol in other columns. We can make similar arguments for downstairs encoding. If downstairs encoding is used, then from Figure 6, the encoding of each parity symbol does not involve any data symbol below it. Also, among the rows spanned by the same riser, the encoding of parity symbols in each row does not involve any data symbol in other rows.

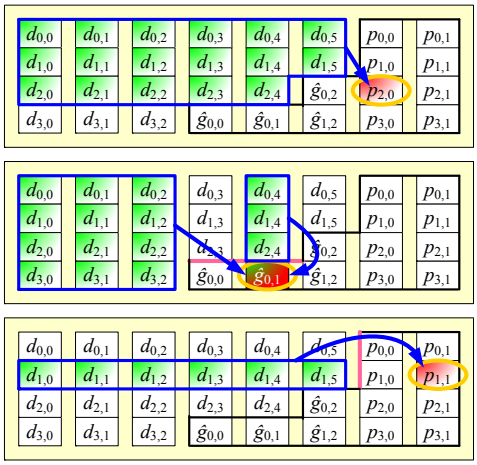


Figure 8: The data symbols that contribute to parity symbols $p_{2,0}$, $\hat{g}_{0,1}$, and $p_{1,1}$, respectively.

As both upstairs and downstairs encoding methods generate the same values of parity symbols, we can combine the above arguments into the following property of how each parity symbol is related to data symbols.

Property 1 (Parity relations in STAIR codes): In a STAIR code stripe, a (row or inside global) parity symbol in Row i_0 and Column j_0 (where $0 \leq i_0 \leq r-1$ and $n-m-m' \leq j_0 \leq n-1$) depends only on the data symbols $d_{i,j}$'s where $i \leq i_0$ and $j \leq j_0$. Moreover, each parity symbol is unrelated to any data symbol in any other column (row) spanned by the same tread (riser).

Figure 8 illustrates the above property. For example, $p_{2,0}$ depends only on the data symbols $d_{i,j}$'s in Rows 0-2 and Columns 0-5. Note that $\hat{g}_{0,1}$ in Column 4 is unrelated to any data symbol in Column 3, which is spanned by the same tread as Column 4. Similarly, $p_{1,1}$ in Row 1 is unrelated to any data symbol in Row 0, which is spanned by the same riser as Row 1.

5.3 Encoding Complexity Analysis

We have proposed two encoding methods for STAIR codes: upstairs encoding and downstairs encoding. Both of them alternately encode rows and columns to obtain the parity symbols. We can also obtain parity symbols using the standard encoding approach, in which each parity symbol is computed directly from a linear combination of data symbols as in classical Reed-Solomon codes. We now analyze the computational complexities of these three methods for different configuration parameters of STAIR codes.

STAIR codes perform encoding over a Galois Field, in which linear arithmetic can be decomposed into the basic operations `Mult_XORs` [31]. We define

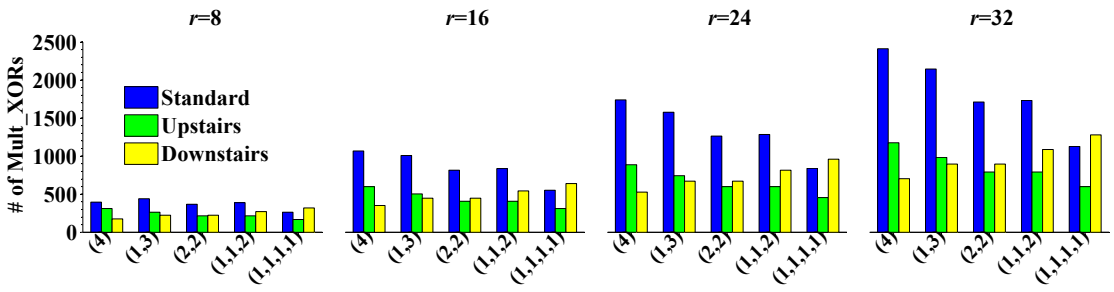


Figure 9: Numbers of Mult_XORs (per stripe) of the three encoding methods for STAIR codes versus different $\mathbf{e}$'s when $n = 8$, $m = 2$, and $s = 4$.

Mult_XOR$(\mathbf{R}_1, \mathbf{R}_2, \alpha)$ as an operation that first multiplies a region $\mathbf{R}_1$ of bytes by a w-bit constant α in Galois Field $GF(2^w)$, and then applies XOR-summing to the product and the target region $\mathbf{R}_2$ of the same size. For example, $\mathbf{Y} = \alpha_0 \cdot \mathbf{X}_0 + \alpha_1 \cdot \mathbf{X}_1$ can be decomposed into two Mult_XORs (assuming $\mathbf{Y}$ is initialized as zero): Mult_XOR$(\mathbf{X}_0, \mathbf{Y}, \alpha_0)$ and Mult_XOR$(\mathbf{X}_1, \mathbf{Y}, \alpha_1)$. Clearly, fewer Mult_XORs imply a lower computational complexity. To evaluate the computational complexity of an encoding method, we count its number of Mult_XORs (per stripe).

For upstairs encoding, we generate $m \cdot r$ row parity symbols and s virtual parity symbols along the row direction, as well as s inside global parity symbols and $(n-m) \cdot e_{m'-1} - s$ virtual parity symbols along the column direction. Its number of Mult_XORs (denoted by $\mathcal{X}_{up}$) is:

$$\mathcal{X}_{up} = \overbrace{(n-m) \times (m \cdot r + s)}^{\text{row direction}} + \overbrace{r \times [(n-m) \cdot e_{m'-1}]}^{\text{column direction}}. \quad (1)$$

For downstairs encoding, we generate $m \cdot r$ row parity symbols, s inside global parity symbols, and $m' \cdot r - s$ intermediate parity symbols along the row direction, as well as s intermediate parity symbols along the column direction. Its number of Mult_XORs (denoted by $\mathcal{X}_{down}$) is:

$$\mathcal{X}_{down} = \overbrace{(n-m) \times \left[(m+m') \cdot r\right]}^{\text{row direction}} + \overbrace{r \times s}^{\text{column direction}}. \quad (2)$$

For standard encoding, we compute the number of Mult_XORs by summing the number of data symbols that contribute to each parity symbol, based on the property of uneven parity relations discussed in §5.2.

We show via a case study how the three encoding methods differ in the number of Mult_XORs. Figure 9 depicts the numbers of Mult_XORs of the three encoding methods for different $\mathbf{e}$'s in the case where $n = 8$, $m = 2$, and $s = 4$. Upstairs encoding and downstairs encoding incur significantly fewer Mult_XORs than standard encoding most of the time. The main reason is that both upstairs encoding and downstairs encoding often reuse the computed parity symbols in subsequent encoding steps. We also observe that for a given s, the number of Mult_XORs of upstairs encoding increases with $e_{m'-1}$ (see Equation (1)), while that of downstairs encoding increases with m' (see Equation (2)). Since larger m' often implies smaller $e_{m'-1}$, the value of m' often determines which of the two encoding methods is more efficient: when m' is small, downstairs encoding wins; when m' is large, upstairs encoding wins.

In our encoding implementation of STAIR codes, for given configuration parameters, we always pre-compute the number of Mult_XORs for each of the encoding methods, and then choose the one with the fewest Mult_XORs.

6 Evaluation

We evaluate STAIR codes and compare them with other related erasure codes in different practical aspects, including storage space saving, encoding/decoding speed, and update penalty.

6.1 Storage Space Saving

The main motivation for STAIR codes is to tolerate simultaneous device and sector failures with significantly lower storage space overhead than traditional erasure codes (e.g., Reed-Solomon codes) that provide only device-level fault tolerance. Given a failure scenario defined by m and $\mathbf{e}$, traditional erasure codes need $m + m'$ chunks per stripe for parity, while STAIR codes need only m chunks and s symbols (where $m' \leq s$). Thus, STAIR codes save $r \times m' - s$ symbols per stripe, or equivalently, $m' - \frac{s}{r}$ devices per system. In short, the saving of STAIR codes depends on only three parameters s, m', and r (where s and m' are determined by $\mathbf{e}$).

Figure 10 plots the number of devices saved by STAIR codes for $s \leq 4$, $m' \leq s$, and $r \leq 32$. As r increases, the number of devices saved is close to m'. The saving reaches the highest when $m' = s$.

We point out that the recently proposed SD codes [27,28] are also motivated for reducing the storage space

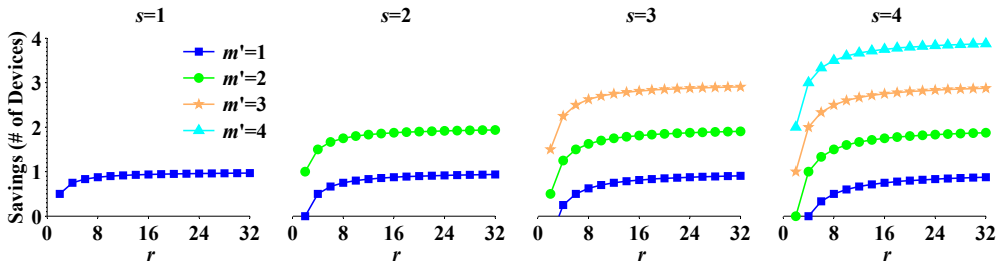


Figure 10: Space saving of STAIR codes over traditional erasure codes in terms of s, m', and r.

over traditional erasure codes. Unlike STAIR codes, SD codes always achieve a saving of $s - \frac{s}{r}$ devices, which is the maximum saving of STAIR codes. While STAIR codes apparently cannot outperform SD codes in space saving, it is important to note that the currently known constructions of SD codes are limited to $s \leq 3$ only [6,27,28], implying that SD codes can save no more than three devices. On the other hand, STAIR codes do not have such limitations. As shown in Figure 10, STAIR codes can save more than three devices for larger s.

6.2 Encoding/Decoding Speed

We evaluate the encoding/decoding speed of STAIR codes. Our implementation of STAIR codes is written in C. We leverage the GF-Complete open source library [31] to accelerate Galois Field arithmetic using Intel SIMD instructions. Our experiments compare STAIR codes with the state-of-the-art SD codes [27, 28]. At the time of this writing, the open-source implementation of SD codes encodes stripes in a decoding manner without any parity reuse. For fair comparisons, we extend the SD code implementation to support the standard encoding method mentioned in §5.3. We run our performance tests on a machine equipped with an Intel Core i5-3570 CPU at 3.40GHz with SSE4.2 support. The CPU has a 256KB L2-cache and a 6MB L3-cache.

6.2.1 Encoding

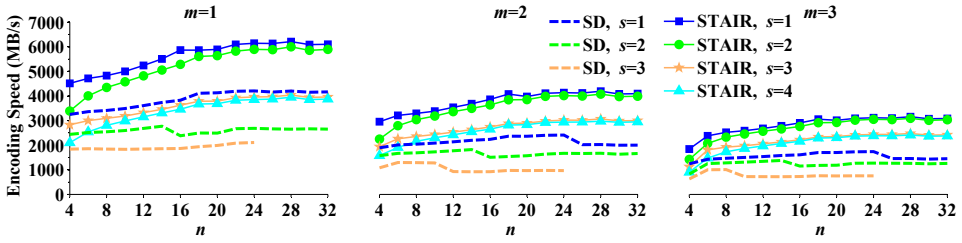
We compare the encoding performance of STAIR codes and SD codes for different values of n, r, m, and s. For SD codes, we only consider the range of configuration parameters where $s \leq 3$, since no code construction is available outside this range [6, 27, 28]. In addition, the SD code constructions for $s = 3$ are only available in the range $n \leq 24$, $r \leq 24$, and $m \leq 3$ [27, 28]. For STAIR codes, a single value of s can imply different configurations of $\mathbf{e}$ (e.g., see Figure 9 in §5.3), each of which has different encoding performance. Here, we take a conservative approach to analyze the worst-case performance of STAIR codes, that is, we test all possible configurations of $\mathbf{e}$ for a given s and pick the one with the lowest encoding speed.

Note that the encoding performance of both STAIR codes and SD codes heavily depends on the word size w of the adopted Galois Field $GF(2^w)$, where w is often set to be a power of 2. A smaller w often means a higher encoding speed [31]. STAIR codes work as long as $n + m' \leq 2^w$ and $r + e_{m'-1} \leq 2^w$. Thus, we choose $w = 8$ since it suffices for all of our tests. However, SD codes may choose among $w = 8$, $w = 16$, and $w = 32$, depending on configuration parameters. We choose the smallest w that is feasible for the SD code construction.

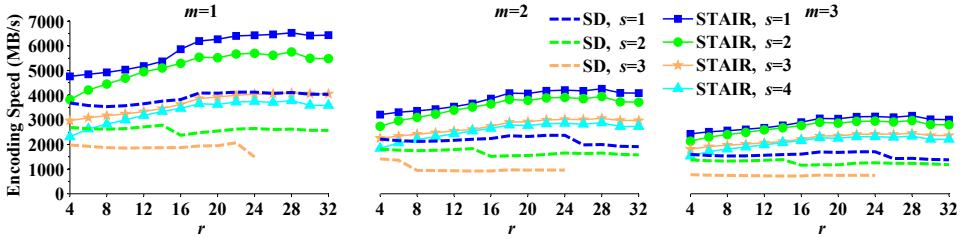
We consider the metric *encoding speed*, defined as the amount of data encoded per second. We construct a stripe of size roughly 32MB in memory as in [27, 28]. We put random bytes in the stripe, and divide the stripe into $r \times n$ sectors, each mapped to a symbol. We obtain the averaged results over 10 runs.

Figures 11(a) and 11(b) present the encoding speed results for different values of n when $r = 16$ and for different values of r when $n = 16$, respectively. In most cases, the encoding speed of STAIR codes is over 1000MB/s, which is significantly higher than the disk write speed in practice (note that although disk writes can be parallelized in disk arrays, the encoding operations can also be parallelized with modern multi-core CPUs). The speed increases with both n and r. The intuitive reason is that the proportion of parity symbols decreases with n and r. Compared to SD codes, STAIR codes improve the encoding speed by 106.03% on average (in the range from 29.30% to 225.14%). The reason is that STAIR codes reuse encoded parity information in subsequent encoding steps by upstairs/downstairs encoding (see §5.3), while such an encoding property is not exploited in SD codes.

We also evaluate the impact of stripe size on the encoding speed of STAIR codes and SD codes for given n and r. We fix $n = 16$ and $r = 16$, and vary the stripe size from 128KB to 512MB. Note that a stripe of size 128KB implies a symbol of size 512 bytes, the standard sector size in practical disk drives. Figure 12 presents the encoding speed results. As the stripe size increases, the encoding speed of both STAIR codes and SD codes first increases and then drops, due to the mixed effects of SIMD instructions adopted in GF-Complete [31] and CPU cache. Nevertheless, the encoding speed advantage of STAIR codes over SD codes remains unchanged.



(a) Varying n when $r = 16$



(b) Varying r when $n = 16$

Figure 11: Encoding speed of STAIR codes and SD codes for different combinations of n, r, m, and s.

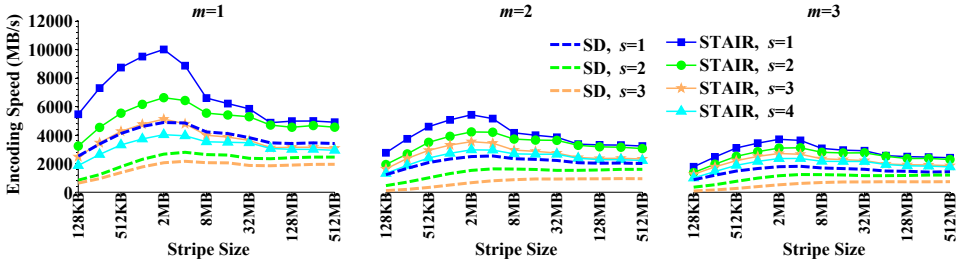


Figure 12: Encoding speed of STAIR codes and SD codes for different stripe sizes when $n = 16$ and $r = 16$.

6.2.2 Decoding

We measure the decoding performance of STAIR codes and SD codes in recovering lost symbols. Since the decoding time increases with the number of lost symbols to be recovered, we consider a particular worst case in which the m leftmost chunks and s additional symbols in the following m' chunks defined by $\mathbf{e}$ are all lost. The evaluation setup is similar to that in §6.2.1, and in particular, the stripe size is fixed at 32MB.

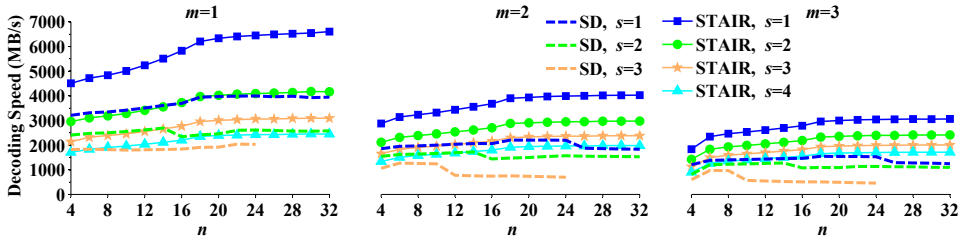
Figures 13(a) and 13(b) present the decoding speed results for different n when $r = 16$ and for different r when $n = 16$, respectively. The results of both figures can be viewed in comparison to those of Figures 11(a) and 11(b), respectively. Similar to encoding, the decoding speed of STAIR codes is over 1000MB/s in most cases and increases with both n and r. Compared to SD codes, STAIR codes improve the decoding speed by 102.99% on average (in the range from 1.70% to 537.87%).

In practice, we often have fewer lost symbols than the worst case (see §4.3). One common case is that there are only failed chunks due to device failures (i.e., $s = 0$), so the decoding of both STAIR and SD codes is identical to that of Reed-Solomon codes. In this case, the decoding speed of STAIR/SD codes can be significantly higher than that of $s = 1$ for STAIR codes in Figure 13. For example, when $n = 16$ and $r = 16$, the decoding speed increases by 79.39%, 29.39%, and 11.98% for $m = 1$, 2, and 3, respectively.

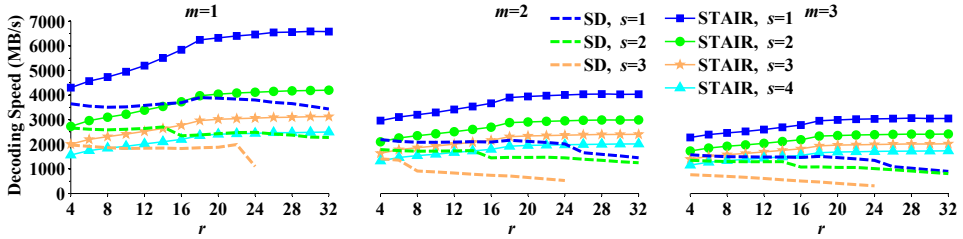
6.3 Update Penalty

We evaluate the update cost of STAIR codes when data symbols are updated. For each data symbol in a stripe being updated, we count the number of parity symbols being affected (see §5.2). Here, we define the *update penalty* as the average number of parity symbols that need to be updated when a data symbol is updated.

Clearly, the update penalty of STAIR codes increases with m. We are more interested in how $\mathbf{e}$ influences the update penalty of STAIR codes. Figure 14 presents the update penalty results for different $\mathbf{e}$'s when $n = 16$ and $s = 4$. For different $\mathbf{e}$'s with the same s, the update penalty of STAIR codes often increases with $e_{m'-1}$. Intuitively, a larger $e_{m'-1}$ implies that more rows of row parity symbols are encoded from inside global parity



(a) Varying n when $r = 16$



(b) Varying r when $n = 16$

Figure 13: Decoding speed of STAIR codes and SD codes for different combinations of n, r, m, and s.

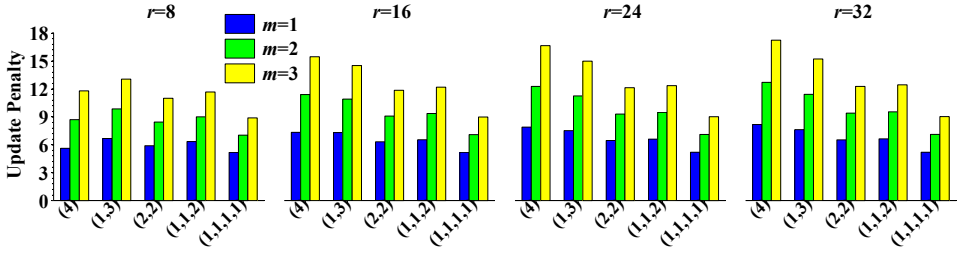


Figure 14: Update penalty of STAIR codes for different $\mathbf{e}$'s when $n = 16$ and $s = 4$.

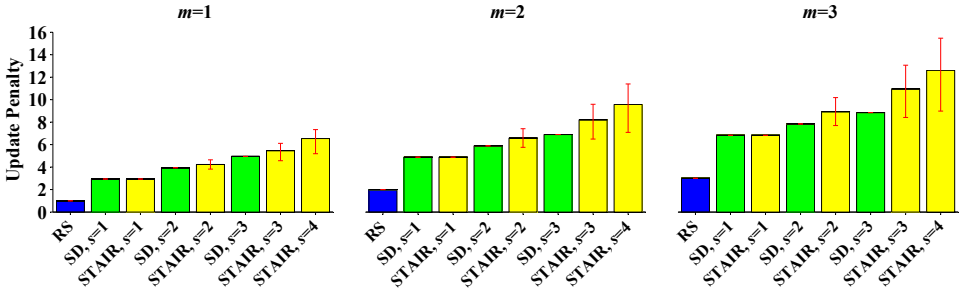


Figure 15: Update penalty of STAIR codes, SD codes, and Reed-Solomon (RS) codes when $n = 16$ and $r = 16$. For STAIR codes, we plot the error bars for the maximum and minimum update penalty values among all possible configurations of $\mathbf{e}$.

symbols, which are further encoded from almost all data symbols (see §5.2).

We compare STAIR codes with SD codes [27,28]. For STAIR codes with a given s, we test all possible configurations of $\mathbf{e}$ and find the average, minimum, and maximum update penalty. For SD codes, we only consider s between 1 and 3. We also include the update penalty results of Reed-Solomon codes for reference. Figure 15 presents the update penalty results when $n = 16$ and $r = 16$ (while similar observations are made for other n and r). For a given s, the range of update penalty of STAIR codes covers that of SD codes, although the average is sometimes higher than that of SD codes (same for $s = 1$, by 7.30% to 14.02% for $s = 2$, and by 10.47% to 23.72% for $s = 3$). Both STAIR codes and SD codes have higher update penalty than Reed-Solomon codes due to more parity symbols in a stripe, and hence are suitable for storage systems with rare updates (e.g., backup

or write-once-read-many (WORM) systems) or systems dominated by full-stripe writes [27, 28].

7 Related Work

Erasure codes have been widely adopted to provide fault tolerance against device failures in storage systems [32]. Classical erasure codes include standard Reed-Solomon codes [34] and Cauchy Reed-Solomon codes [7], both of which are MDS codes that provide general constructions for all possible configuration parameters. They are usually implemented as systematic codes for storage applications [26,30,33], and thus can be used to implement the construction of STAIR codes. In addition, Cauchy Reed-Solomon codes can be further transformed into *array codes*, whose encoding computations purely build on efficient XOR operations [33].

In the past decades, many kinds of array codes have been proposed, including MDS array codes (e.g., [2–4,9, 12,13,20,22,29,41,42]) and non-MDS array codes (e.g., [16, 17, 23]). Array codes are often designed for specific configuration parameters. To avoid compromising the generality of STAIR codes, we do not suggest to adopt array codes in the construction of STAIR codes. Moreover, recent work [31] has shown that Galois Field arithmetic can be implemented to be extremely fast (sometimes at cache line speeds) using SIMD instructions in modern processors.

Sector failures are not explicitly considered in traditional erasure codes, which focus on tolerating device-level failures. To cope with sector failures, ad hoc schemes are often considered. One scheme is *scrubbing* [24, 36, 38], which proactively scans all disks and recovers any spotted sector failure using the underlying erasure codes. Another scheme is *intra-device redundancy* [10, 11, 36], in which contiguous sectors in each device are grouped together to form a segment and are then encoded with redundancy within the device. Our work targets a different objective and focuses on constructing an erasure code that explicitly addresses sector failures.

To simultaneously tolerate device and sector failures with minimal redundancy, SD codes [27, 28] (including the earlier PMDS codes [5], which are a subset of SD codes) have recently been proposed. As stated in §1, SD codes are known only for limited configurations and some of the known constructions rely on extensive searches. A relaxation of the SD property has also been recently addressed as a future work in [27], which assumes that each row has no more than a given number of sector failures. It is important to note that the relaxation of [27] is different from ours, in which we limit the maximum number of devices with sector failures and the maximum number of sector failures that simultaneously occur in each such device. It turns out that our relaxation enables us to derive a general code construction. Another similar kind of erasure codes is the family of locally repairable codes (LRCs) [18, 19, 35]. Pyramid codes [18] are designed for improving the recovery performance for small-scale device failures and have been implemented in archival storage [40]. Huang *et al.*'s and Sathiamoorthy *et al.*'s LRCs [19, 35] can be viewed as generalizations of Pyramid codes and are recently adopted in commercial storage systems. In particular, Huang *et al.*'s LRCs [19] achieve the same fault tolerance property as PMDS codes [5], and thus can also be used as SD codes. However, the construction of Huang *et al.*'s LRCs is limited to $m = 1$ only. To our knowledge, STAIR codes are the first general family of erasure codes that can efficiently tolerate both device and sector failures.

8 Conclusions

We present STAIR codes, a general family of erasure codes that can tolerate simultaneous device and sector failures in a space-efficient manner. STAIR codes can be constructed for tolerating any numbers of device and sector failures subject to a pre-specified sector failure coverage. The special construction of STAIR codes also makes efficient encoding/decoding possible through parity reuse. Compared to the recently proposed SD codes [5, 27, 28], STAIR codes not only support a much wider range of configuration parameters, but also achieve higher encoding/decoding speed based on our experiments.

In future work, we explore how to correctly configure STAIR codes in practical storage systems based on empirical failure characteristics [1, 25, 36, 37].

The source code of STAIR codes is available at **http://ansrlab.cse.cuhk.edu.hk/software/stair**.

Acknowledgments

We would like to thank our shepherd, James S. Plank, and the anonymous reviewers for their valuable comments. This work was supported in part by grants from the University Grants Committee of Hong Kong (project numbers: AoE/E-02/08 and ECS CUHK419212).

References

[1] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, pages 289–300, San Diego, CA, June 2007.

[2] M. Blaum. A family of MDS array codes with minimal number of encoding operations. In *Proceedings of the 2006 IEEE International Symposium on*

Information Theory (ISIT '06), pages 2784–2788, Seattle, WA, July 2006.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, 44(2):192–202, 1995.

[4] M. Blaum, J. Bruck, and A. Vardy. MDS array codes with independent parity symbols. *IEEE Transactions on Information Theory*, 42(2):529–542, 1996.

[5] M. Blaum, J. L. Hafner, and S. Hetzler. Partial-MDS codes and their application to RAID type of architectures. *IEEE Transactions on Information Theory*, 59(7):4510–4519, July 2013.

[6] M. Blaum and J. S. Plank. Construction of sector-disk (SD) codes with two global parity symbols. IBM Research Report RJ10511 (ALM1308-007), Almaden Research Center, IBM Research Division, Aug. 2013.

[7] J. Blomer, 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.

[8] S. Boboila and P. Desnoyers. Write endurance in flash drives: Measurements and analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, pages 115–128, San Jose, CA, Feb. 2010.

[9] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, pages 1–14, San Francisco, CA, Mar. 2004.

[10] A. Dholakia, E. Eleftheriou, X.-Y. Hu, I. Iliadis, J. Menon, and K. Rao. A new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. *ACM Transactions on Storage*, 4(1):1–42, 2008.

[11] A. Dholakia, E. Eleftheriou, X.-Y. Hu, I. Iliadis, J. Menon, and K. Rao. Disk scrubbing versus intradisk redundancy for RAID storage systems. *ACM Transactions on Storage*, 7(2):1–42, 2011.

[12] G. Feng, R. Deng, F. Bao, and J. Shen. New efficient MDS array codes for RAID Part I: Reed-Solomon-like codes for tolerating three disk failures. *IEEE Transactions on Computers*, 54(9):1071–1080, 2005.

[13] G. Feng, R. Deng, F. Bao, and J. Shen. New efficient MDS array codes for RAID Part II: Rabin-like codes for tolerating multiple (≥ 4) disk failures. *IEEE Transactions on Computers*, 54(12):1473–1483, 2005.

[14] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO '09)*, pages 24–33, New York, NY, Dec. 2009.

[15] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '12)*, pages 17–24, San Jose, CA, Feb. 2012.

[16] J. L. Hafner. WEAVER codes: Highly fault tolerant erasure codes for storage systems. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 211–224, San Francisco, CA, Dec. 2005.

[17] J. L. Hafner. HoVer erasure codes for disk arrays. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN '06)*, pages 1–10, Philadelphia, PA, June 2006.

[18] C. Huang, M. Chen, and J. Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *ACM Transactions on Storage*, 9(1):1–28, Mar. 2013.

[19] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure storage. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*, pages 15–26, Boston, MA, June 2012.

[20] C. Huang and L. Xu. STAR: An efficient coding scheme for correcting triple storage node failures. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 889–901, San Francisco, CA, Dec. 2005.

[21] Intel Corporation. Intelligent RAID 6 theory — overview and implementation. White Paper, 2005.

[22] M. Li and J. Shu. C-Codes: Cyclic lowest-density MDS array codes constructed using starters for RAID 6. IBM Research Report RC25218 (C1110-004), China Research Laboratory, IBM Research Division, Oct. 2011.

[23] M. Li, J. Shu, and W. Zheng. GRID codes: Strip-based erasure codes with high fault tolerance for storage systems. *ACM Transactions on Storage*, 4(4):1–22, 2009.

[24] A. Oprea and A. Juels. A clean-slate look at disk scrubbing. In *Proceedings of the 8th USENIX Con-*

ference on File and Storage Technologies (FAST '10), pages 1–14, San Jose, CA, Feb. 2010.

[25] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX conference on File and Storage Technologies (FAST '07)*, pages 17–28, San Jose, CA, Feb. 2007.

[26] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software — Practice & Experience*, 27(9):995–1012, 1997.

[27] J. S. Plank and M. Blaum. Sector-disk (SD) erasure codes for mixed failure modes in RAID systems. Technical Report CS-13-708, University of Tennessee, May 2013.

[28] J. S. Plank, M. Blaum, and J. L. Hafner. SD codes: Erasure codes designed for how storage systems really fail. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST '13)*, pages 95–104, San Jose, CA, Feb. 2013.

[29] J. S. Plank, A. L. Buchsbaum, and B. T. Vander Zanden. Minimum density RAID-6 codes. *ACM Transactions on Storage*, 6(4):1–22, May 2011.

[30] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software — Practice & Experience*, 35(2):189–194, 2005.

[31] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois Field arithmetic using Intel SIMD instructions. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST '13)*, pages 299–306, San Jose, CA, Feb. 2013.

[32] J. S. Plank and C. Huang. Tutorial: Erasure coding for storage applications. Slides presented at FAST-2013: 11th Usenix Conference on File and Storage Technologies, Feb. 2013.

[33] J. S. Plank and L. Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *Proceedings of the 5th IEEE International Symposium on Network Computing and Applications (NCA '06)*, pages 173–180, Cambridge, MA, July 2006.

[34] 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.

[35] M. Sathiamoorthy, M. Asteris, D. Papailiopoulous, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB '13)*, pages 325–336, Trento, Italy, Aug. 2013.

[36] B. Schroeder, S. Damouras, and P. Gill. Understanding latent sector errors and how to protect against them. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, pages 71–84, San Jose, CA, Feb. 2010.

[37] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX conference on File and Storage Technologies (FAST '07)*, pages 1–16, San Jose, CA, Feb. 2007.

[38] T. J. E. Schwarz, Q. Xin, E. L. Miller, and D. D. E. Long. Disk scrubbing in large archival storage systems. In *Proceedings of the 12th Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)*, pages 409–418, Volendam, Netherlands, Oct. 2004.

[39] J. White and C. Lueth. RAID-DP: NetApp implementation of double-parity RAID for data protection. Technical Report TR-3298, NetApp, Inc., May 2010.

[40] A. Wildani, T. J. E. Schwarz, E. L. Miller, and D. D. Long. Protecting against rare event failures in archival systems. In *Proceedings of the 17th Annual Meeting of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '09)*, pages 1–11, London, UK, Sept. 2009.

[41] L. Xu, V. Bohossian, J. Bruck, and D. G. Wagner. Low-density MDS codes and factors of complete graphs. *IEEE Transactions on Information Theory*, 45(6):1817–1826, Sept. 1999.

[42] L. Xu and J. Bruck. X-Code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999.

[43] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge. Understanding the robustness of SSDs under power fault. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST '13)*, pages 271–284, San Jose, CA, Feb. 2013.

Appendix: Proof of Homomorphic Property

We formally prove the homomorphic property described in §4.1. We state the following theorem.

Theorem 1 *In the construction of the canonical stripe of STAIR codes, the encoding of each chunk in the column direction via C_{col} is homomorphic, such that each*

augmented row in the canonical stripe is a codeword of $\mathcal{C}_{row}$.

Proof: We prove by matrix operations. We define the matrices $\mathbf{D} = [d_{i,j}]_{r\times(n-m)}$, $\mathbf{P} = [p_{i,k}]_{r\times m}$, and $\mathbf{P'} = [p'_{i,l}]_{r\times m'}$. Also, we define the generator matrices $\mathbf{G}_{row}$ and $\mathbf{G}_{col}$ for the codes $\mathcal{C}_{row}$ and $\mathcal{C}_{col}$, respectively, as:

$$\begin{aligned}\mathbf{G}_{row} &= \left(\mathbf{I}_{(n-m)\times(n-m)} \mid \mathbf{A}_{(n-m)\times(m+m')}\right), \\ \mathbf{G}_{col} &= \left(\mathbf{I}_{r\times r} \mid \mathbf{B}_{r\times e_{m'-1}}\right),\end{aligned}$$

where $\mathbf{I}$ is an identity matrix, and $\mathbf{A}$ and $\mathbf{B}$ are the submatrices that form the parity symbols. The upper r rows of the stripe can be expressed as follows:

$$(\mathbf{D} \mid \mathbf{P} \mid \mathbf{P'}) = \mathbf{D}\cdot\mathbf{G}_{row}.$$

The lower $e_{m'-1}$ augmented rows are expressed as follows:

$$\begin{aligned}\left((\mathbf{D} \mid \mathbf{P} \mid \mathbf{P'})^T \cdot \mathbf{B}\right)^T &= \mathbf{B}^T \cdot (\mathbf{D}\cdot\mathbf{G}_{row}) \\ &= \left(\mathbf{B}^T\cdot\mathbf{D}\right)\cdot\mathbf{G}_{row}\end{aligned}$$

We can see that each of the lower $e_{m'-1}$ rows can be calculated using the generator matrix $\mathbf{G}_{row}$, and hence is a codeword of $\mathcal{C}_{row}$. □

Parity Logging with Reserved Space: Towards Efficient Updates and Recovery in Erasure-coded Clustered Storage

Jeremy C. W. Chan*, Qian Ding*, Patrick P. C. Lee, Helen H. W. Chan
The Chinese University of Hong Kong
{*cwchan,qding,pclee,hwchan*}*@cse.cuhk.edu.hk*

Abstract

Many modern storage systems adopt erasure coding to provide data availability guarantees with low redundancy. Log-based storage is often used to append new data rather than overwrite existing data so as to achieve high update efficiency, but introduces significant I/O overhead during recovery due to reassembling updates from data and parity chunks. We propose parity logging with reserved space, which comprises two key design features: (1) it takes a hybrid of in-place data updates and log-based parity updates to balance the costs of updates and recovery, and (2) it keeps parity updates in a reserved space next to the parity chunk to mitigate disk seeks. We further propose a workload-aware scheme to dynamically predict and adjust the reserved space size. We prototype an erasure-coded clustered storage system called CodFS, and conduct testbed experiments on different update schemes under synthetic and real-world workloads. We show that our proposed update scheme achieves high update and recovery performance, which cannot be simultaneously achieved by pure in-place or log-based update schemes.

1 Introduction

Clustered storage systems are known to be susceptible to component failures [17]. High data availability can be achieved by encoding data with redundancy using either replication or erasure coding. Erasure coding encodes original data chunks to generate new parity chunks, such that a subset of data and parity chunks can sufficiently recover all original data chunks. It is known that erasure coding introduces less overhead in storage and write bandwidth than replication under the same fault tolerance [37, 47]. For example, traditional 3-way replication used in GFS [17] and Azure [8] introduces 200% of redundancy overhead, while erasure coding can reduce the overhead to 33% and achieve higher availability [22]. Today's enterprise clustered storage systems [14, 22, 35, 39, 49] adopt erasure coding in production to reduce hardware footprints and maintenance costs.

For many real-world workloads in enterprise servers and network file systems [2, 30], data updates are dominant. There are two ways of performing updates: (1) *in-place updates*, where the stored data is read, modified, and written with the new data, and (2) *log-based updates*, where updates are inserted to the end of an append-only log [38]. If updates are frequent, in-place updates introduce significant I/O overhead in erasure-coded storage since parity chunks also need to be updated to be consistent with the data changes. Existing clustered storage systems, such as GFS [17] and Azure [8] adopt log-based updates to reduce I/Os by sequentially appending updates. On the other hand, log-based updates introduce additional disk seeks to the update log during sequential reads. This in particular hurts recovery performance, since recovery makes large sequential reads to the data and parity chunks in the surviving nodes in order to reconstruct the lost data.

This raises an issue of choosing the appropriate update scheme for an erasure-coded clustered storage system to achieve efficient updates and recovery simultaneously. Our primary goal is to mitigate the network transfer and disk I/O overheads, both of which are potential bottlenecks in clustered storage systems. In this paper, we make the following contributions.

First, we provide a taxonomy of existing update schemes for erasure-coded clustered storage systems. To this end, we propose a novel update scheme called *parity logging with reserved space*, which uses a hybrid of in-place data updates and log-based parity updates. It mitigates the disk seeks of reading parity chunks by putting deltas of parity chunks in a reserved space that is allocated next to their parity chunks. We further propose a workload-aware reserved space management scheme that effectively predicts the size of reserved space and reclaims the unused reserved space.

Second, we build an erasure-coded clustered storage system *CodFS*, which targets the common update-dominant workloads and supports efficient updates and recovery. CodFS offloads client-side encoding computations to the storage cluster. Its implementation is extensible for different erasure coding and update schemes, and is deployable on commodity hardware.

Finally, we conduct testbed experiments using synthetic and real-world traces. We show that our CodFS prototype achieves network-bound read/write perfor-

*The first two authors contributed equally to this work.

mance. Under real-world workloads, our proposed parity logging with reserved space gives a 63.1% speedup of update throughput over pure in-place updates and up to 10× speedup of recovery throughput over pure log-based updates. Also, our workload-aware reserved space management effectively shrinks unused reserved space with limited reclaim overhead.

The rest of the paper proceeds as follows. In §2, we analyze the update behaviors in real-world traces. In §3, we introduce the background of erasure coding. In §4, we present different update schemes and describe our approach. In §5, we present the design of CodFS. In §6, we present testbed experimental results. In §7, we discuss related work. In §8, we conclude the paper.

2 Trace Analysis

We study two sets of real-world storage traces collected from large-scale storage server environments and characterize their update patterns. Motivated by the fact that enterprises are considering erasure coding as an alternative to RAID for fault-tolerant storage [40], we choose these traces to represent the workloads of enterprise storage clusters and study the applicability of erasure coding to such workloads. We want to answer three questions: (1) *What is the average size of each update?* (2) *How common do data updates happen?* (3) *Are updates focused on some particular chunks?*

2.1 Trace Description

MSR Cambridge traces. We use the public block-level I/O traces of a storage cluster released by Microsoft Research Cambridge [30]. The traces are captured on 36 volumes of 179 disks located in 13 servers. They are composed of I/O requests, each specifying the timestamp, the server name, the disk number, the read/write type, the starting logical block address, the number of bytes transferred, and the response time. The whole traces span a one-week period starting from 5PM GMT on 22nd February 2007, and account for the workloads in various kinds of deployment including user home directories, project directories, source control, and media. Here, we choose 10 of the 36 volumes for our analysis. Each of the chosen volumes contains 800,000 to 4,000,000 write requests.

Harvard NFS traces. We also use a set of NFS traces (`DEAS03`) released by Harvard [13]. The traces capture NFS requests and responses of a NetApp file server that contains a mix of workloads including email, research, and development. The whole traces cover a 41-day period from 29th January 2003 to 10th March 2003. Each NFS request in the traces contains the timestamp, source and destination IP addresses, and the RPC function. Depending on the RPC function, the request may contain optional fields such as file handler, file offset and length.

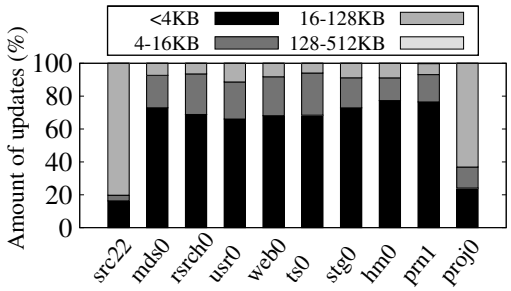


Figure 1: Distribution of update size in MSR Cambridge traces.

No. of Writes	172702071
WSS (GB)	174.73
Updated WSS (%)	68.39
Update Writes (%)	91.56
No. of Accessed Files	2039724
Updated Files (%)	12.10
Avg. Update Size Per Request (KB)	10.58

Table 1: Properties of Harvard `DEAS03` NFS traces.

While the traces describe the workloads of a single NFS server, they have also been used in trace-driven analysis for clustered storage systems [1, 20].

2.2 Key Observations

Updates are small. We study the update size, i.e., the number of bytes accessed by each update. Figure 1 shows the average update size ranges of the MSR Cambridge traces. We see that the updates are generally small in size. Although different traces show different update size compositions, all updates occurring in the traces are smaller than 512KB. Among the 10 traces, eight of them have more than 60% of updates smaller than 4KB. Similarly, the Harvard NFS traces comprise small updates, with average size of only 10.58KB, as shown in Table 1.

Updates are common. Unsurprisingly, updates are common in both storage traces. We analyze the write requests in the traces and classify them into two types: *first-write*, i.e., the address is first accessed, and *update*, i.e., the address is re-accessed. Table 1 shows the results of the Harvard NFS traces. Among nearly 173 million write requests, more than 91% of them are updates. Table 2 shows the results of the MSR Cambridge traces. All the volumes show more than 90% of updates among all write requests, except for the print server volume `prn1`. We see limited relationship between the working set size (WSS) and the intensity of writes. For example, the project volume `proj0` has a small WSS, but it has much more writes than the source control volume `src22` that has a large WSS.

Update coverage varies. Although data updates are common in all traces, the coverage of updates varies. We measure the update coverage by studying the frac-

Volume	Workload Type	No. of Writes	WSS (GB)	Updated WSS(%)	Update Writes(%)
src22	Source control	805955	20.17	99.57	99.68
mds0	Media server	1067061	3.09	29.27	95.77
rsrch0	Research	1300030	0.36	69.53	97.41
usr0	Home directory	1333406	2.44	42.54	96.08
web0	Web/SQL server	1423458	7.26	37.25	96.23
ts0	Terminal server	1485042	0.91	49.84	95.65
stg0	Web staging	1722478	6.31	21.04	97.82
hm0	HW monitor	2575568	2.31	73.16	93.21
prn1	Print server	2769610	80.9	18.55	73.43
proj0	Project directory	3697143	3.16	56.67	98.89

Table 2: Properties of MSR Cambridge traces: (1) number of writes shows the total number of write requests; (2) working set size refers to the size of unique data accessed in the trace; (3) percentage of updated working set size refers to the fraction of data in the working set that is updated at least once; and (4) percentage of update writes refers to the fraction of writes that update existing data.

tion of WSS that is updated at least once throughout the trace period. For example, from the MSR Cambridge traces in Table 2, the `src22` trace shows a 99.57% of updated WSS, while updates in the `mds0` trace only cover 29.27% of WSS. In other words, updates in the `src22` trace span across a large number of locations in the working set, while updates in the `mds0` trace are focused on a smaller set of locations. The variation in update coverage implies the need of a dynamic mechanism to improve update efficiency.

3 Background: Erasure Coding

We provide the background details of an erasure-coded storage system considered in this work. We refer readers to the tutorial [33] for the essential details of erasure coding in the context of storage systems.

We consider an erasure-coded storage cluster with M nodes (or servers). We divide data into *segments* and apply erasure coding independently on a per-segment basis. We denote an (n, k)-code as an erasure coding scheme defined by two parameters n and k, where $k < n$. An (n, k)-code divides a segment into k equal-size uncoded chunks called *data chunks*, and encodes the data chunks to form $n - k$ coded chunks called *parity chunks*. We assume $n < M$, and have the collection of n data and parity chunks distributed across n of the M nodes in the storage cluster. We consider *Maximum Distance Separable* erasure coding, i.e., the original segment can be reconstructed from any k of the n data and parity chunks.

Each parity chunk can be in general encoded by computing a linear combination of the data chunks. Mathematically, for an (n, k)-code, let $\{\gamma_{ij}\}_{1\le i\le n-k, 1\le j\le k}$ be a set of encoding coefficients for encoding the k data chunks $\{D_1, D_2, \cdots, D_k\}$ into $n - k$ parity chunks $\{P_1, P_2, \cdots, P_{n-k}\}$. Then, each parity chunk P_i ($1 \le i \le n - k$) can be computed by: $P_i = \sum_{j=1}^{k} \gamma_{ij} D_j$, where all arithmetic operations are performed in the Galois Field over the coding units called *words*.

The linearity property of erasure coding provides an alternative to computing new parity chunks when some data chunks are updated. Suppose that a data chunk D_l (for some $1 \le l \le k$) is updated to another data chunk D'_l. Then each new parity chunk P'_i ($1 \le i \le n - k$) can be computed by:

$$P'_i = \sum_{j=1, j\neq l}^{k} \gamma_{ij} D_j + \gamma_{il} D'_l = P_i + \gamma_{il}(D'_l - D_l).$$

Thus, instead of summing over all data chunks, we compute new parity chunks based on the change of data chunks. The above computation can be further generalized when only part of a data chunk is updated, but a subtlety is that a data update may affect different parts of a parity chunk depending on the erasure code construction (see [33] for details). Suppose now that a word of D_l at offset o is updated, and the word of P_l at offset $\hat{o}$ needs to be updated accordingly (where o and $\hat{o}$ may differ). Then we can express:

$$P'_i(\hat{o}) \quad = \quad P_i(\hat{o}) + \gamma_{il}(D'_l(o) - D_l(o)),$$

where $P'_i(\hat{o})$ and $P_i(\hat{o})$ denote the words at offset $\hat{o}$ of the new parity chunk P'_i and old parity chunk P_i, respectively, and $D'_l(o)$ and $D_l(o)$ denote the words at offset o of the new data chunk D'_l and old data chunk D_l, respectively. In the following discussion, we leverage this linearity property in parity updates.

4 Parity Updates

Data updates in erasure-coded clustered storage systems introduce performance overhead, since they also need to update parity chunks for consistency. We consider a deployment environment where network transfer and disk I/O are performance bottlenecks. Our goal is to design a parity update scheme that effectively mitigates both network transfer overhead and number of disk seeks.

We re-examine existing parity update schemes that fall into two classes: the RAID-based approaches and the delta-based approaches. We then propose a novel parity update approach that assigns a reserved space for keeping parity updates.

4.1 Existing Approaches

4.1.1 RAID-based Approaches

We describe three classical approaches of parity updates that are typically found in RAID systems [10, 45].

Full-segment writes. A full-segment write (or full-stripe write) updates all data and parity chunks in a segment. It is used in a large sequential write where the

write size is a multiple of segment size. To make a full-segment write work for small updates, one way is to pack several updates into a large piece until a full segment can be written in a single operation [28]. Full-segment writes do not need to read the old data or parity chunks, and hence achieve the best update performance.

Reconstruct writes. A reconstruct write first reads all the chunks from the segment that are not involved in the update. Then it computes the new parity chunks using the read chunks and the new chunks to be written, and writes all data and parity chunks.

Read-modify writes. A read-modify write leverages the linearity of erasure coding for parity updates (see §3). It first reads the old data chunk to be updated and all the old parity chunks in the segment. It then computes the change between the old and new data chunks, and applies the change to each of the parity chunks. Finally, it writes the new data chunk and all new parity chunks to their respective locations.

Discussion. Full-segment writes can be implemented through a log-based design to support small updates, but logging has two limitations. First, we need an efficient garbage collection mechanism to reclaim space by removing stale chunks, and this often hinders update performance [41]. Second, logging introduces additional disk seeks to retrieve the updates, which often degrades sequential read and recovery performance [27]. On the other hand, both reconstruct writes and read-modify writes are traditionally designed for a single host deployment. Although some recent studies implement read-modify writes in a distributed setting [15, 51], both approaches introduce significant network traffic since each update must transfer data or parity chunks between nodes for parity updates.

4.1.2 Delta-based Approaches

Another class of parity updates, called the *delta-based approaches*, eliminates redundant network traffic by only transferring a *parity delta* which is of the same size as the modified data range [9, 44]. A delta-based approach leverages the linearity of erasure coding described in §3. It first reads the range of the data chunk to be modified and computes the delta, which is the change between old and new data at the modified range of the data chunk, for each parity chunk. It then sends the modified data range and the parity deltas computed to the data node and all other parity nodes for updates, respectively. Instead of transferring the entire data and parity chunks as in read-modify writes, transferring the modified data range and parity deltas reduces the network traffic and is suitable for clustered storage. In the following, we describe some delta-based approaches proposed in the literature.

Full-overwrite (FO). Full-overwrite [4] applies in-place updates to both data and parity chunks. It merges the old data and parity chunks directly at specific offsets with the modified data range and parity deltas, respectively. Note that merging each parity delta requires an additional disk read of old parity chunk at the specific offset to compute the new parity content to be written.

Full-logging (FL). Full-logging saves the disk read overhead of parity chunks by appending all data and parity updates. That is, after the modified data range and parity deltas are respectively sent to the corresponding data and parity nodes, the storage nodes create logs to store the updates. The logs will be merged with the original chunks when the chunks are read subsequently. FL is used in enterprise clustered storage systems such as GFS [17] and Azure [8].

Parity-logging (PL). Parity-logging [24, 43] can be regarded as a hybrid of FO and FL. It saves the disk read overhead of parity chunks and additionally avoids merging overhead on data chunks introduced in FL. Since data chunks are more likely to be read than parity chunks, merging logs in data chunks can significantly degrade read performance. Hence, in PL, the original data chunk is overwritten in-place with the modified data range, while the parity deltas are logged at the parity nodes.

Discussion. Although the delta-based approaches reduce network traffic, they are not explicitly designed to reduce disk I/O. Both FL and PL introduce disk fragmentation and require efficient garbage collection. The fragmentations often hamper further accesses of those chunks with logs. Meanwhile, FO introduces additional disk reads for the old parity chunks on the update path, compared with FL and PL. Hence, to take a step further, we want to address the question: *Can we reduce the disk I/O on both the update path and further accesses?*

4.2 Our Approach

We propose a new delta-based approach called **parity-logging with reserved space (PLR)**, which further mitigates fragmentation and reduces the disk seek overhead of PL in storing parity deltas. The main idea is that the storage nodes reserve additional storage space next to each parity chunk for keeping parity deltas. This ensures that each parity chunk and its parity deltas can be sequentially retrieved. While the idea is simple, the challenging issues are to determine (1) the appropriate amount of reserved space to be allocated when a parity chunk is first stored and (2) the appropriate time when unused reserved space can be reclaimed to reduce the storage overhead.

4.2.1 An Illustrative Example

Figure 2 illustrates the differences of the delta-based approaches in §4.1.2 and PLR, using a (3,2)-code as an example. The incoming data stream describes the sequence of operations: (1) write the first segment with

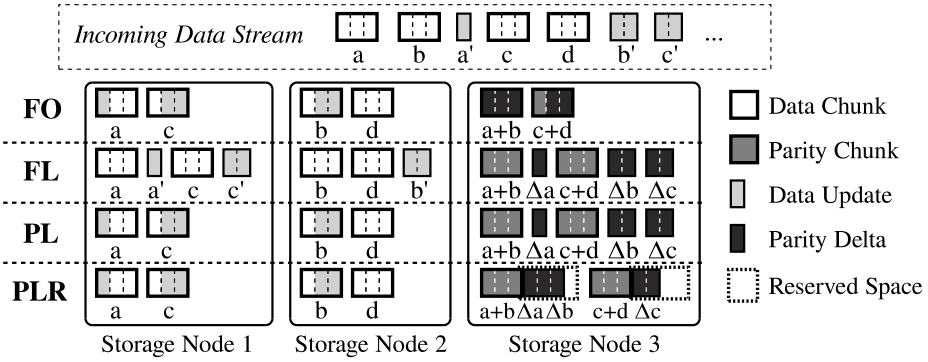


Figure 2: Illustration on different parity update schemes.

data chunks a and b, (2) update part of a with a′, (3) write a new segment with data chunks c and d, and finally (4) update parts of b and c with b′ and c′, respectively. We see that FO performs overwrites for both data updates and parity deltas; FL appends both data updates and parity deltas according to the incoming order; PL performs overwrites for data updates and appends parity deltas; and PLR appends parity deltas in reserved space.

Consider now that we read the up-to-date chunk b. FL incurs a disk seek to the update b′ when rebuilding chunk b, as b and b′ are in discontinuous physical locations on disk. Similarly, PL also incurs a disk seek to the parity delta Δb when reconstructing the parity chunk a+b. On the other hand, PLR incurs no disk seek when reading the parity chunk a+b since its parity deltas Δa and Δb are all placed in the contiguous reserved space following the parity chunk a+b.

4.2.2 Determining the Reserved Space Size

Finding the appropriate reserved space size is challenging. If the space is too large, then it wastes storage space. On the other hand, if the space is too small, then it cannot keep all parity deltas.

A baseline approach is to use a fixed reserved space size for each parity chunk, where the size is assumed to be large enough to fit all parity deltas. Note that this baseline approach can introduce significant storage overhead, since different segments may have different update patterns. For example, from the Harvard NFS traces shown in Table 1, although 91.56% of write requests are updates, only around 12% of files are actually involved. This uneven distribution implies that fixing a large, constant size of reserved space can imply unnecessary space wastage.

For some workloads, the baseline approach may reserve insufficient space to hold all deltas for a parity chunk. There are two alternatives to handle extra deltas, either logging them elsewhere like PL, or *merging* existing deltas with the parity chunk to reclaim the reserved space. We adopt the merge alternative since it preserves the property of no fragmentation in PLR.

Algorithm 1: Workload-aware Reserved Space Management

```
1 reserved ←DEFAULT_SIZE
2 while true do
3     sleep(period)
4     foreach chunk in parityChunkSet do
5         utility ← getUtility(chunk)
6         size ← computeShrinkSize(utility)
7         doShrink(size, chunk)
8         doMerge(chunk)
```

To this end, we propose a workload-aware reserved space management scheme that dynamically adjusts and predicts the reserved space size. The scheme has three main parts: (1) predicting the reserved space size of each parity chunk using the measured workload pattern for the next time interval, (2) shrinking the reserved space and releasing unused reserved space back to the system, and (3) merging parity deltas in the reserved space to each parity chunk. To avoid introducing small unusable holes of reclaimed space after shrinking, we require that both the reserved space size and the shrinking size be of multiples of the chunk size. This ensures that an entire data or parity chunk can be stored in the reclaimed space.

Algorithm 1 describes the basic framework of our workload-aware reserved space management. Initially, we set a default reserved space size that is sufficiently large to hold all parity deltas. Shrinking and prediction are then executed periodically on each storage node. Let $\mathcal{S}$ be the set of parity chunks in a node. For every time interval t and each parity chunk $p \in \mathcal{S}$, let $r_t(p)$ be the reserved space size and $u_t(p)$ be the reserved space utility. Intuitively, $u_t(p)$ represents the fraction of reserved space being used. We measure $u_t(p)$ at the end of each time interval t using exponential weighted moving average in getUtility:

$$u_t(p) = \alpha \frac{use(p)}{r_t(p)} + (1-\alpha)u_{t-1}(p),$$

where *use*(p) returns the reserved space size being used during the time interval, $r_t(p)$ is the current reserved space size for chunk p, and α is the smoothing factor. According to the utility, we decide the unnecessary space size $c(p)$ that can be reclaimed for the parity chunk p in `computeShrinkSize`. Here, we aggressively shrink all unused space $c(p)$ and round it down to be a multiple of the chunk size:

$$c(p) = \left\lfloor \frac{(1 - u_t(p))r_t(p)}{ChunkSize} \right\rfloor \times ChunkSize.$$

The `doShrink` function attempts to shrink the size $c(p)$ from the current reserved space $r_t(p)$. Thus, the reserved space $r_{t+1}(p)$ for p at time interval $t + 1$ is:

$$r_{t+1}(p) = r_t(p) - c(p).$$

If a chunk has no more reserved space after shrinking (i.e., $r_{t+1}(p) = 0$), any subsequent update requests to this chunk are applied in-place as in FO.

Finally, the `doMerge` function merges the deltas in the reserved space to the parity chunk p after shrinking and resets *use(p)* to zero. Hence we free the parity chunk from carrying any deltas to the next time interval, which could further reduce the reserved space size. The merge operations performed here are off the update path and have limited impact on the overall system performance.

The above workload-aware design of reserved space management is simple and can be replaced by a more advanced design. Nevertheless, we find that this simple heuristic works well enough under real-world workloads (see §6.3.2).

5 CodFS Design

We design CodFS, an erasure-coded clustered storage system that implements the aforementioned delta-based update schemes to support efficient updates and recovery.

5.1 Architecture

Figure 3 shows the CodFS architecture. The *metadata server (MDS)* stores and manages all file metadata, while multiple *object storage devices (OSDs)* perform coding operations and store the data and parity chunks. The MDS also plays a monitor role, such that it keeps track of the health status of the OSDs and triggers recovery when some OSDs fail. A CodFS client can access the storage cluster through a file system interface.

5.2 Work Flow

CodFS performs erasure coding on the write path as illustrated in Figure 3. To write a file, the client first splits the file into segments, and requests the MDS to store the metadata and identify the *primary OSD* for each segment. The client then sends each segment to its primary OSD, which encodes the segment into k data chunks and

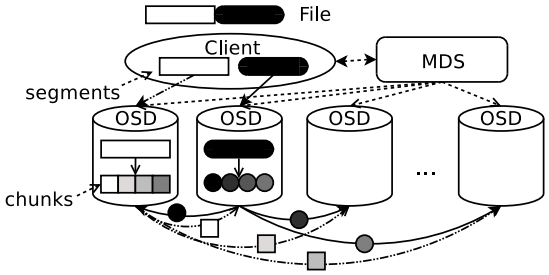


Figure 3: CodFS architecture.

$n - k$ parity chunks for some pre-specified parameters n and k. The primary OSD stores a data chunk locally, and distributes the remaining $n-1$ chunks to other OSDs called the *secondary OSDs* for the segment. The identities of the secondary OSDs are assigned by the MDS to keep the entire cluster load-balanced. Both primary and secondary OSDs are defined in a logical sense, such that each physical OSD can act as a primary OSD for some segments and a secondary OSD for others.

To read a segment, the client first queries MDS for the primary OSD. It then issues a read request to the primary OSD, which collects one data chunk locally and $k - 1$ data chunks from other secondary OSDs and returns the original segment to the client. In the normal state where no failure occurs, the primary OSD only needs the k data chunks of the segment for rebuilding.

CodFS adopts the delta-based approach for data updates. To update a segment, the client sends the modified data with the corresponding offsets to the segment's primary OSD, which first splits the update into sub-updates according to the offsets, such that each sub-update targets a single data chunk. The primary OSD then sends each sub-update to the OSD storing the targeted data chunk. Upon receiving a sub-update for a data chunk, an OSD computes the parity deltas and distributes them to the parity destinations. Finally, both the updates and parity deltas are saved according to the chosen update scheme.

CodFS switches to degraded mode when some OSDs fail (assuming the number of failed OSDs is tolerable). The primary OSD coordinates the degraded operations for its responsible segments. If the primary OSD of a segment fails, CodFS promotes another surviving secondary OSD of the segment to be the primary OSD. CodFS supports degraded reads and recovery. To issue a degraded read to a segment, the primary OSD follows the same read path as the normal case, except that it collects both data and parity chunks of the segment. It then decodes the collected chunks and returns the original segment. If an OSD failure is deemed permanent, CodFS can recover the lost chunks on a new OSD. That is, for each segment with lost chunks, the corresponding primary OSD first reconstructs the segment as in degraded reads, and then writes the lost chunk to the new OSD. Our current implementation of degraded reads and re-

covery uses the standard approach that reads k chunks for reconstruction, and it works for any number of failed OSDs no more than $n - k$. Nevertheless, our design is also compatible with efficient recovery approaches that read less data under single failures (e.g., [25, 50]).

5.3 Issues

We address several implementation issues in CodFS and justify our design choices.

Consistency. CodFS provides close-to-open consistency [21], which offers the same level of consistency as most Network File Systems (NFS) clients. Any open request to a segment always returns the version following the previous close request. CodFS directs all reads and writes of a segment through the corresponding primary OSD, which uses a lock-based approach to serialize the requests of all clients. This simplifies consistency implementation.

Offloading. CodFS offloads the encoding and reconstruction operations from clients. Client-side encoding generates more write traffic since the client needs to transmit parity chunks. Using the primary OSD design limits the fan-outs of clients and the traffic between the clients and the storage cluster. In addition, CodFS splits each file into segments, which are handled by different primary OSDs in parallel. Hence, the computational power of a single OSD will not become a bottleneck on the write path. Also, within each OSD, CodFS uses multi-threading to pipeline and parallelize the I/O and encoding operations, so as to mitigate the overhead in encoding computations.

Metadata Management. The MDS stores all metadata in a key-value database built on MongoDB [29]. CodFS can configure a backup MDS to serve the metadata operations in case the main MDS fails, similar to HDFS [5].

Caching. CodFS adopts simple caching techniques to boost the entire system performance. Each CodFS client is equipped with an LRU cache for segments so that frequent updates of a single segment can be batched and sent to the primary OSD. The LRU cache also favors frequent reads of a single segment, to avoid fetching the segment from the storage cluster in each read. We do not consider specific write mitigation techniques (e.g., lazy write-back and compression) or advanced caches (e.g., distributed caching or SSDs), although our system can be extended with such approaches.

Segment Size. CodFS supports flexible segment size from 16MB to 64MB and sets the default at 16MB. This size is chosen to fully utilize both the network bandwidth and disk throughput, as shown in our experiments (see §6.1). Smaller segments lead to more disk I/Os and degrade the write throughput, while larger segments cannot fully leverage the I/O parallelism across multiple OSDs.

5.4 Implementation Details

We design CodFS based on commodity configurations. We implement all the components including the client and the storage backend in C++ on Linux. CodFS leverages several third-party libraries for high-performance operations, including: (1) Threadpool [46], which manages a pool of threads that parallelize I/O and encoding operations, (2) Google Protocol Buffers [18], which serialize message communications between different entities, (3) Jerasure [32], which provides interfaces for efficient erasure coding implementation, and (4) FUSE [16], which provides a file system interface for clients.

We design the OSD via a modular approach. The *Coding Module* of each OSD provides a standard interface for implementation of different coding schemes. One can readily extend CodFS to support new coding schemes. The *Storage Module* inside each OSD acts as an abstract layer between the physical disk and the OSD process. We store chunk updates and parity deltas according to the update scheme configured in the *Storage Module*. By default, CodFS uses the PLR scheme. Each OSD is equipped with a *Monitor Module* to perform garbage collection in FL and PL and reserved space shrinking and prediction in PLR.

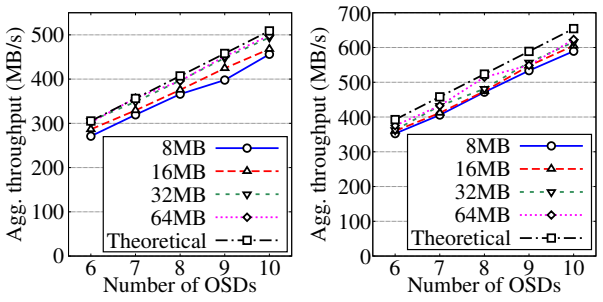
We adopt Linux Ext4 as the local filesystem of each OSD to support fast reserved space allocation. We preallocate the reserved space for each parity chunk using the Linux system call `fallocate`, which marks the allocated blocks as uninitialized. Shrinking of the reserved space is implemented by invoking `fallocate` with the `FALLOC_FL_PUNCH_HOLE` flag. Since we allocate or shrink the reserved space as a multiple of chunk size, we avoid creating unusable holes in the file system.

6 Evaluation

We evaluate different parity update schemes through our CodFS prototype. We deploy CodFS on a testbed with 22 nodes of commodity hardware configurations. Each node is a Linux machine running Ubuntu Server 12.04.2 with kernel version 3.5. The MDS and OSD nodes are each equipped with Intel Core i5-3570 3.4GHz CPU, 8GB RAM and two Seagate ST1000DM003 7200RPM 1TB SATA harddisk. For each OSD, the first harddisk is used as the OS disk while the entire second disk is used for storing chunks. The client nodes are equipped with Intel Core 2 Duo 6420 2.13GHz CPU, 2GB RAM and a Seagate ST3160815AS 7200RPM 160GB SATA harddisk. Each node has a Gigabit Ethernet card installed and all nodes are connected via a Gigabit full-duplex switch.

6.1 Baseline Performance

We derive the achievable aggregate read/write throughput of CodFS and analyze its best possible performance. Suppose that the encoding overhead can be entirely



(a) Sequential write (b) Sequential read

Figure 4: Aggregate read/write throughput of CodFS using the RDP code with $(n, k) = (6, 4)$.

masked by our parallel design. If our CodFS prototype can effectively mitigate encoding overhead and evenly distribute the operations among OSDs, then it should achieve the theoretical throughput.

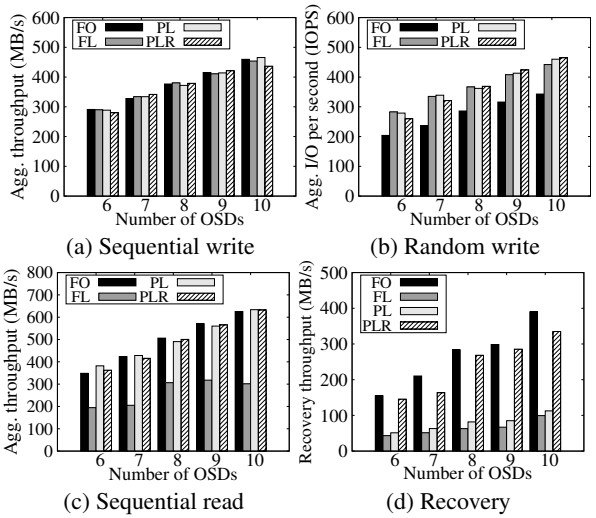
We define the notation as follows. Let M be the total number of OSDs in the system, and let B_{in} and B_{out} be the available inbound and outbound bandwidths (in network or disk) of each OSD, respectively. Each encoding scheme can be described by the parameters n and k, following the same definitions in §3.

We derive the effective aggregate write throughput (denoted by T_{write}). Each primary OSD, after encoding a segment, stores one chunk locally and distributes $n-1$ chunks to other secondary OSDs. This introduces an additional $(n-1)/k$ times of segment traffic among the OSDs. Similarly, for the effective aggregate read throughput (denoted by T_{read}), each primary OSD collects $(k-1)$ chunks for each read segment from the secondary OSDs. It introduces an additional $(k-1)/k$ times of segment traffic. Thus, T_{write} and T_{read} are given by:

$$T_{write} = \frac{M \times B_{in}}{1 + \frac{n-1}{k}}, \qquad T_{read} = \frac{M \times B_{out}}{1 + \frac{k-1}{k}}.$$

We evaluate the aggregate read/write throughput of CodFS, and compare the experimental results with our theoretical results. We first conduct measurements on our testbed and find that the effective disk and network bandwidths of each node are 144MB/s and 114.5MB/s, respectively. Thus, the nodes are network-bound, and we set $B_{in} = B_{out} = 114.5$MB/s in our model. We configure CodFS with one node as the MDS and M nodes as OSDs, where $6 \leq M \leq 10$. We consider the RAID-6 RDP code [12] with $(n, k) = (6, 4)$. The coded chunks are distributed over the M OSDs. We have 10 other nodes in the testbed as clients that transfer streams of segments simultaneously.

Figure 4 shows the aggregate read/write throughput of CodFS versus the number of OSDs for different segment sizes from 8MB to 64MB. We see that the throughput results match closely with the theoretical results, and



(a) Sequential write (b) Random write

(c) Sequential read (d) Recovery

Figure 5: Throughput of CodFS under different update schemes.

the throughput scales linearly with the number of OSDs. For example, when $M = 10$ OSDs are used, CodFS achieves read and write throughput of at least 580MB/s and 450MB/s, respectively.

We also evaluate the throughput results of CodFS configured with the Reed-Solomon (RS) codes [34]. We observe that both RDP and RS codes have almost identical throughput, although RS codes have higher encoding overhead [32]. The reason is that CodFS masks the encoding overhead through parallelization. We do not present the results here in the interest of space.

6.2 Evaluation on Synthetic Workload

We now evaluate the four delta-based parity update schemes (i.e., FO, FL, PL, and PLR) using our CodFS prototype under a synthetic workload. Unless otherwise stated, we use the RDP code [12] with $(n, k) = (6, 4)$, 16MB segment size, and the same cluster configuration as in §6.1. We measure the sequential write, random write, sequential read, and recovery performance of CodFS using IOzone [23]. For PLR, we use the baseline approach described in §4.2.2 and fix the size of reserved space to 4MB, which is equal to the chunk size in our configuration. We trigger a merge operation to reclaim the reserved space when it becomes full. Before running each test, we format the chunk partition of each OSD to restore the OSD to a clean state, and drop the buffer cache in all OSDs to ensure that any difference in performance is attributed to the update schemes.

We note that an update to a data chunk in RDP [12] involves more updates to parity chunks than in RS codes (see [33] for illustration), and hence generates larger-size parity deltas. This triggers more frequent merge operations as the reserved space becomes full faster.

		FO	FL	PL	PLR
Synthetic	Data	0	29.41	0	0
	Parity	0	117.66	117.66	0

Table 3: Average non-contiguous fragments per chunk (F_{avg}) after random writes for synthetic workload.

6.2.1 Sequential Write Performance

Figure 5a shows the aggregate sequential write throughput of CodFS under different update schemes, in which all clients simultaneously write 2GB of segments to the storage cluster. As expected, there is only negligible difference in sequential write throughput among the four update schemes as the experiment only writes new data.

6.2.2 Random Write Performance

We use IOzone to simulate intensive small updates, in which we issue uniform random writes with 128KB record length to all segments uploaded in §6.2.1. In total, we generate 16MB of updates for each segment, which is four times of the reserved space size in PLR. Thus, PLR performs at least four merge operations per parity chunk (more merges are needed if the coding scheme triggers the updates of multiple parts of a parity chunk for each data update). Figure 5b shows the numbers of I/Os per second (IOPS) of the four update schemes. Results show that FO performs the worst among the four, with at least 21.0% fewer IOPS than the other three schemes. This indicates that updating both the data and parity chunks in-place incurs extra disk seeks and parity read overhead, thereby significantly degrading update efficiency. The other three schemes give similar update performance with less than 4.1% difference in IOPS.

6.2.3 Sequential Read Performance

Sequential read and recovery performance are affected by disk fragmentation in data and parity chunks. To measure fragmentation, we define a metric F_{avg} as the *average number of non-contiguous fragments per chunk* that are read from disk to rebuild the up-to-date chunk. Empirically, F_{avg} is found by reading the physical block addresses of each chunk in the underlying file system of the OSDs using the `filefrag -v` command which is available in the `e2fsprogs` utility. For each chunk, we obtain the number of non-contiguous fragments by analyzing its list of physical block addresses and lengths. We then take the average over the chunks in all OSDs.

Table 3 shows the value of F_{avg} measured after random writes in §6.2.2. Both FO and PLR have $F_{avg} = 0$ as they either store updates and deltas in-place or in a contiguous space next to their parity chunks. FL is the only scheme that contains non-contiguous fragments for data chunks, and it has $F_{avg} = 29.41$ in the synthetic benchmark. Logging parity deltas introduces higher level of disk fragmentation. On average, both FL and PL produce 117.66 non-contiguous fragments per parity chunk in the synthetic benchmark. We see that F_{avg} of parity chunks is about $4\times$ that of data chunks. This conforms to our RDP configuration with $(n, k) = (6, 4)$ since each segment consists of four data chunks and modifying each of them once will introduce a total of four parity deltas to each parity chunk.

Figure 5c shows a scenario which we execute a sequential read after intensive random writes. We measure the aggregate sequential read throughput under different update schemes. In this experiment, all clients simultaneously read the segments after performing the updates described in §6.2.2.

Since CodFS only reads data chunks when there are no node failures, no performance difference in sequential read is observed for FO, PL and PLR. However, the sequential read performance of FL drops by half when compared with the other three schemes. This degradation is due to the combined effect of disk seeking and merging overhead for data chunk updates. The result also agrees with the measured level of disk fragmentation shown in Table 3 where FL is the only scheme that contains non-contiguous fragments for data chunks.

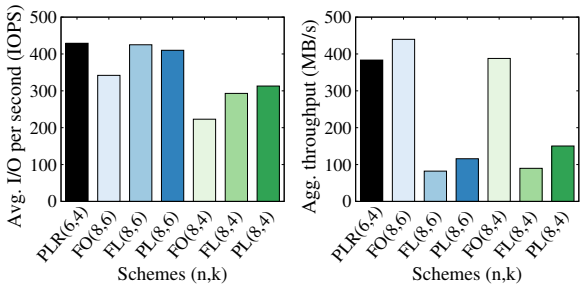
6.2.4 Recovery Performance

We evaluate the recovery performance of CodFS under a double failure scenario, and compare the results among different update schemes. We trigger the recovery procedure by sending `SIGKILL` to the CodFS process in two of the OSDs. We measure the time between sending the kill signal and receiving the acknowledgement from the MDS reporting all data from the failed OSDs are reconstructed and redistributed among the available OSDs.

Figure 5d shows the measured recovery throughput for different update schemes. FO is the fastest in recovery and achieves substantial difference in recovery throughput (up to $4.5\times$) compared with FL due to the latter suffering from merging and disk seeking overhead for both data and parity chunks. By keeping data chunks updates in-place, PL achieves a modest increase in recovery throughput compared with FL. We also see the benefits of PLR for keeping delta updates next to their parity chunks. PLR gains a $3\times$ improvement on average in recovery throughput when compared with PL.

6.2.5 Reserved Space versus Update Efficiency

We thus far evaluate the parity update schemes under the same coding parameters (n, k). Since PLR trades storage space for update efficiency, we also compare PLR with other schemes that use the reserved space for storage. Here, we set the reserved space size to be equal to the chunk size in PLR with $(n, k) = (6, 4)$. This implies that a size of two extra chunks is reserved per segment. For FO, FL, and PL, we substitute the reserved space



(a) Random write (b) Recovery

Figure 6: Throughput comparison under the same storage overhead using Cauchy RS codes with various (n, k).

with either two data chunks or two parity chunks. We realize the substitutions with erasure coding using two coding parameters: $(n, k) = (8, 6)$ and $(n, k) = (8, 4)$, which in essence store two additional data chunks, and two additional parity chunks over $(n, k) = (6, 4)$, respectively. Since RDP requires $n - k = 2$, we choose the Cauchy RS code [7] as the coding scheme. We also fix the chunk size to be 4MB, so we ensure that each coded segment in all 7 configurations takes 32MB of storage including data, parity, and reserved space.

Figure 6 shows the performance of random writes and recovery under the same synthetic workload described in §6.2.2. Results show that the $(8, 4)$ schemes perform significantly worse than the $(8, 6)$ schemes in random writes, since having more parity chunks implies more parity updates. Also, we see that FO $(8, 6)$ is slower than PLR $(6, 4)$ by at least 20% in terms of IOPS, indicating that allocating more data chunks does not necessarily boost update performance. Results of recovery agree with those in §6.2.4, i.e., both FO and PLR give significantly higher recovery throughput than FL and PL.

6.2.6 Summary of Results

We make the following observations from our synthetic evaluation. First, although our configuration has twice as many data chunks as parity chunks, updating data chunks in-place in PL does not help much in recovery throughput. This implies that the time spent on reading and rebuilding parity chunks dominates the recovery performance. Second, as shown in Table 3, both FO and PLR do not produce disk seeks. Thus, we can attribute the difference in recovery throughput between FO and PLR solely to the merging overhead for parity updates. We see that PLR incurs less than 9.2% in recovery throughput on average compared with FO. We regard this as a reasonable trade-off since recovery itself is a less common operation than random writes.

6.3 Evaluation on Real-world Traces

Next, we evaluate CodFS by replaying the MSR Cambridge and Harvard NFS traces analyzed in §2.

6.3.1 MSR Cambridge Traces

To limit the experiment duration, we choose 10 of the 36 volumes for evaluating the update and recovery performance. We choose the traces with the number of write requests between $800,000$ and $4,000,000$. Also, to demonstrate that our design does not confine to a specific workload, the traces we select for evaluation all come from different types of servers.

We first pre-process the traces as follows. We adjust the offset of each request accordingly so that the offset maps to the correct location of a chunk. We ensure that the locality of requests to the chunks is preserved. If there are consecutive requests made to a sequence of blocks, they will be combined into one request to preserve the sequential property during replay.

We configure CodFS to use 10 OSDs and split the trace evenly to distribute replay workload among 10 clients. We first write the segments that cover the whole working set size of the trace. Each client then replays the trace by writing to the corresponding offset of the preallocated segments. We use RDP [12] with $(n, k) = (6, 4)$ and 16MB segment size.

Update Performance. Figure 7 shows the aggregate number of writes replayed per second. To perform a stress test, we ignore the original timestamps in the traces and replay the operations as fast as possible. First, we observe that traces with a smaller coverage (as indicated by the percentage of updated WSS in Table 2) in general results in higher IOPS no matter which update scheme is used. For example, the `usr0` trace with 13.08% updated WSS shows more than $3\times$ update performance when compared with the `src22` trace with 99.57% updated WSS. This is due to a more effective client LRU cache when the updates are focused on a small set of chunks. The cache performs write coalescing and reduces the number of round-trips between clients and OSDs. Second, we see that the four schemes exhibit similar behaviour across traces. FL, PL and PLR show comparable update performance. This leads us to the same implication as in §6.2.2 that the dominant factor influencing update performance is the overhead in parity updates. Therefore, the three schemes that use a log-based design for parity chunks all perform significantly better than FO. On average, PLR is 63.1% faster than FO.

Recovery Performance. Figure 8 shows the recovery throughput in a two-node failure scenario. We see that in all traces, FL and PL are slower than FO and PLR in recovery. Also, PLR outperforms FL and PL more significantly in traces where there is a large number of writes and F_{avg} is high. For example, the measured F_{avg} for the `proj0` trace is 45.66 and 182.6 for data and parity chunks, respectively, and PLR achieves a remarkable $10\times$ speedup in recovery throughput over FL. On

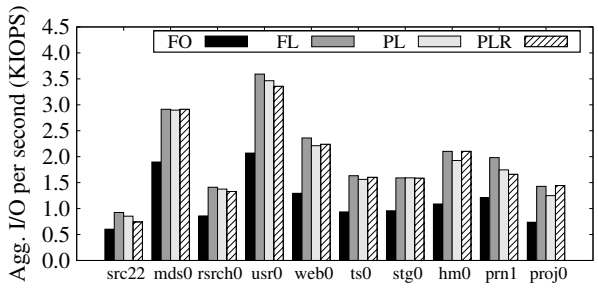


Figure 7: Number of write operations per second replaying the selected MSR Cambridge traces under different update schemes.

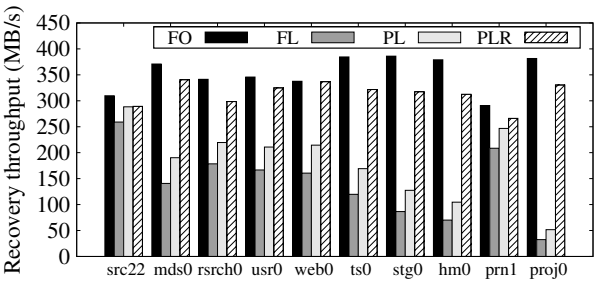


Figure 8: Recovery throughput in a double failure scenario after replaying the selected MSR Cambridge traces under different update schemes.

the other hand, PLR performs the worst in the `src22` trace, where F_{avg} is only 0.73 and 2.82 for data and parity chunks, respectively. Nevertheless, it still manages to give an 11.7% speedup over FL.

6.3.2 Evaluation of Reserved Space

We evaluate our workload-aware approach in managing the reserved space size (see §4.2.2). We use the Harvard NFS traces, whose 41-day span provides long enough duration to examine the effectiveness of shrinking, merging, and prediction. We calculate the *reserved space storage overhead* using the following equation, which is defined as the additional storage space allocated by the reserved space compared with the original working set size without any reserved space:

$$\Gamma = \frac{\sum ReservedSpaceSize}{\sum (DataSize + ParitySize)}.$$

A low Γ means that the reserved space is small compared with the total size of all data and parity chunks.

Using the above metric, we evaluate our workload-aware framework used in PLR by simulating the Harvard NFS traces. We set the segment size to 16MB and use the Cauchy RS code [7] with $(n, k) = (10, 8)$. Here, we compare our workload-aware approach with three baseline approaches, in which we fix the reserved space size to 2MB, 8MB, and 16MB without any adjustment.

We consider two variants of our workload-aware approach. The *shrink+merge* approach executes the shrinking operation at 00:00 and 12:00 on each day, followed by a merge operation on each chunk. The *shrink only* approach is identical to the *shrink+merge* approach in shrinking, but does not perform any merge operation after shrinking (i.e., it does not free the space occupied by the parity deltas). On the first day, we initialize the reserved space to 16MB. We follow the framework described in §4.2.2 and set the smoothing factor $\alpha = 0.3$.

Simulation Results. Figure 9 shows the value of Γ under the three different approaches by simulating the 41-day Harvard traces. The 2MB, 8MB, and 16MB baseline approaches give $\Gamma = 0.2$, 0.8, and 1.6, respectively, throughout the entire trace since they never shrink the reserved space. The values of Γ for both workload-aware variants drop quickly in the first week of trace and then gradually stabilize. At the end of the trace, the *shrink only* approach has Γ of about 0.36. With merging, the *shrink+merge* approach further reduces Γ to 0.12. Γ is lower than that of the 2MB baseline, as around 13% of parity chunks end up with zero reserved space size.

Aggressive shrinking may increase the number of merge operations. We examine such an impact by showing the average number of merges per 1000 writes in Figure 10. A lower value implies lower write latency since fewer writes are stalled by merge operations. We make a few key observations from this figure. First, the 16MB baseline gives the best results among all strategies, since it keeps the largest reserved space than other baselines and workload-aware approaches throughout the whole period. On the contrary, using a fixed reserve space that is too small increases the number of merges significantly. This effect is shown by the 2MB baseline. Second, the performance of the workload-aware approaches matches closely with the 8MB and 16MB baseline approaches most of the time. Day 30-40 is an exception in which the two workload-aware approaches perform significantly more merges than the 16MB baseline approach. This reflects the penalty of inaccurate prediction when the reserved space is not large enough to handle the sudden bursts in updates. Third, although the *shrink+merge* approach has a lower reserved space storage overhead, it incurs more penalty than the *shrink only* approach in case of a misprediction. However, we observe that on average less than 1% of writes are stalled by a merge operation regardless of which approach is used (recall that the merge is performed every 1000 writes). Thus, we expect that there is very little impact of merging on the performance in PLR.

6.3.3 Summary of Results

We show that PLR achieves efficient updates and recovery. It significantly improves the update through-

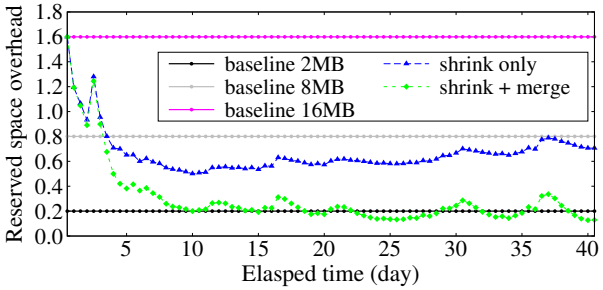


Figure 9: Reserved space overhead under different shrink strategies in the Harvard trace.

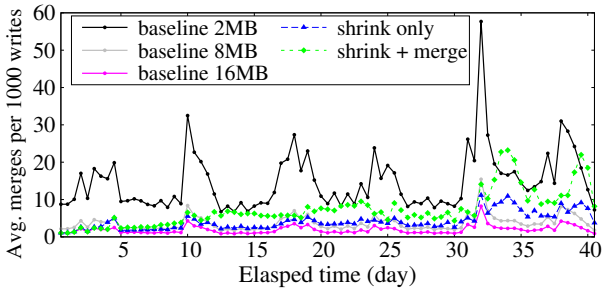


Figure 10: Average number of merges per 1000 writes under different shrink strategies in the Harvard trace.

put of FO and the recovery throughput of FL. We also evaluate our workload-aware approach on reserved space management. We show that the *shrink+merge* approach can reduce the reserved space storage overhead by more than half compared to the 16MB baseline approach, with slight merging penalty to reclaim space.

7 Related Work

Quantitative analysis shows that erasure coding consumes less bandwidth and storage than replication with similar system durability [37, 47]. Several studies adopt erasure coding in distributed storage systems. OceanStore [26, 36] combines replication and erasure coding for wide-area network storage. TotalRecall [6] applies replication or erasure coding to different files dynamically according to the availability level predicted by the system. Ursa Minor [1] focuses on cluster storage and encodes files of heterogeneous types based on the failure models and access patterns. Panasas [49] performs client-side encoding on a per-file basis. TickerTAIP [9], PARAID [48] and Pergamum [44] offload the parity computation to the storage array. Azure [22] and Facebook [39] propose efficient erasure coding schemes to speed up degraded reads. We complement the above studies by improving update efficiency and recovery performance in erasure-coded clustered storage.

Log-structured File System (LFS) [38] first proposes to append updates sequentially to disk to improve write performance. Zebra [19] extends LFS for RAID-like distributed storage systems by striping logs across servers. Self-tuning LFS [27] exploits workload characteristics to improve I/O performance. Clustered storage systems, such as GFS [17] and Azure [8], also adopt the LFS design for the write-once read-many workload. The more recent work Gecko [42] uses a chained-log design to reduce disk I/O contention of LFS in RAID storage. CodFS handles updates differently from LFS, in which it performs in-place updates to data and log-based updates to parity chunks. It also allocates reserve space for parity logging to further mitigate disk seeks. The above studies (including CodFS) focus on disk backends and commodity hardware, while the LFS design is also adopted in other types of emerging storage media, such as SSDs [3] and DRAM storage [31].

Parity logging [11, 43] has been proposed to mitigate the disk seek overhead in parity updates. It accumulates parity updates for each parity region in a log and flushes updates to the parity region when the log is full. The parity and log regions can be distributed across all disks [43]. On the other hand, CodFS reserves log space next to each parity chunk so as to reduce disk seeks due to frequent small writes. It extends the prior parity logging approaches by allowing future shrinking of the reserved space based on the workload.

8 Conclusions

Our key contribution is the parity logging with reserved space (PLR) scheme, which keeps parity updates next to the parity chunk to mitigate disk seeks. We also propose a workload-aware scheme to predict and adjust the reserved space size. To this end, we build CodFS, an erasure-coded clustered storage system that achieves efficient updates and recovery. We evaluate our CodFS prototype using both synthetic and real-world traces and show that PLR improves update and recovery performance over pure in-place and log-based updates. In future work, we plan to (1) evaluate other metrics (e.g., latency) of different parity update schemes, (2) evaluate the impact of the shrinking and merging operations on throughput and latency, and (3) explore a more robust design of reserved space management. The source code of CodFS is available for public-domain use on **http://ansrlab.cse.cuhk.edu.hk/software/codfs**.

Acknowledgments

We would like to thank our shepherd, Ethan L. Miller, and the anonymous reviewers for their valuable comments. This work was supported in part by grants AoE/E-02/08 and ECS CUHK419212 from the University Grants Committee of Hong Kong and ITS/250/11 from the ITF of HKSAR.

References

[1] M. Abd-El-Malek, W. Courtright II, C. Cranor, G. Ganger, J. Hendricks, A. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. Sambasivan, et al. Ursa Minor: Versatile Cluster-based Storage. In *Proc. of USENIX FAST*, Dec 2005.

[2] I. F. Adams, M. W. Storer, and E. L. Miller. Analysis of Workload Behavior in Scientific and Historical Long-Term Data Repositories. *ACM Trans. on Storage*, 8:6:1–6:27, 2012.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. of USENIX ATC*, Jun 2008.

[4] M. K. Aguilera and R. Janakiraman. Using Erasure Codes Efficiently for Storage in a Distributed System. In *Proc. of IEEE DSN*, Jun 2005.

[5] Apache. HDFS Architecture Guide. `http://hadoop.apache.org/docs/stable1/hdfs_design.html`.

[6] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total Recall: System Support for Automated Availability Management. In *Proc. of USENIX NSDI*, Oct 2004.

[7] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based Erasure-resilient Coding Scheme. Technical report, International Computer Science Institute, Berkeley, USA, 1995.

[8] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of ACM SOSP*, Oct 2011.

[9] P. Cao, S. B. Lin, S. Venkataraman, and J. Wilkes. The TickerTAIP Parallel RAID Architecture. *ACM Trans. Comput. Syst.*, 12:236–269, 1994.

[10] P. M. Chen and E. K. Lee. Striping in a RAID Level 5 Disk Array. In *Proc. of ACM SIGMETRICS*, 1995.

[11] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Comput. Surv.*, 26(2):145–185, Jun 1994.

[12] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proc. of USENIX FAST*, Mar 2004.

[13] D. J. Ellard. *Trace-based Analyses and Optimizations for Network Storage Servers*. PhD thesis, Cambridge, MA, USA, 2004. AAI3131831.

[14] 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.

[15] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A Decentralized Algorithm for Erasure-Coded Virtual Disks. In *Proc. of IEEE DSN*, Jun 2004.

[16] FUSE. Filesystem in Userspace. `http://fuse.sourceforge.net/`.

[17] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, Dec 2003.

[18] Google. Google Protocol Buffers. `https://code.google.com/p/protobuf/`.

[19] J. H. Hartman and J. K. Ousterhout. The Zebra Striped Network File System. *ACM Trans. Comput. Syst.*, 13:274–310, 1995.

[20] J. Hendricks, R. R. Sambasivan, S. Sinnamohideen, and G. R. Ganger. Improving Small File Performance in Object-based Storage. Technical Report CMU-PDL-06-104, Carnegie Mellon University, May 2006.

[21] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst.*, 6:51–81, 1988.

[22] 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.

[23] IOzone. IOzone Filesystem Benchmark. `http://www.iozone.org/`.

[24] C. Jin, D. Feng, H. Jiang, and L. Tian. RAID6L: A Log-assisted RAID6 Storage Architecture with Improved Write Performance. In *Proc. of IEEE MSST*, 2011.

[25] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *Proc. of USENIX FAST*, Feb 2012.

[26] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and

B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. of ACM ASPLOS-IX*, Nov 2000.

[27] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-structured File Systems with Adaptive Methods. In *Proc. of ACM SOSP*, Oct 1997.

[28] J. Menon. A Performance Comparison of RAID-5 and Log-structured Arrays. In *Proc. of 4th IEEE International Symposium on High Performance Distributed Computing (HDPC)*, 1995.

[29] MongoDB, Inc. MongoDB. `http://www.mongodb.org/`.

[30] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-loading: Practical Power Management for Enterprise Storage. *ACM Trans. on Storage*, 4:10:1–10:23, 2008.

[31] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proc. of ACM SOSP*, Oct 2011.

[32] J. Plank, J. Luo, C. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proc. of USENIX FAST*, Feb 2009.

[33] J. S. Plank and C. Huang. Erasure Codes for Storage Systems: A Brief Primer. *;login: the Usenix magazine*, 38(6):44–50, Dec 2013.

[34] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, Jun 1960.

[35] J. K. Resch and J. S. Plank. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In *Proc. of USENIX FAST*, Feb 2011.

[36] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore Prototype. In *Proc. of USENIX FAST*, Mar 2003.

[37] R. Rodrigues and B. Liskov. High Availability in DHTs: Erasure Coding vs. Replication. In *Proc. of IPTPS*, Feb 2005.

[38] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.*, 10:26–52, 1992.

[39] M. Sathiamoorthy, M. Asteris, D. S. 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*, Aug 2013.

[40] SearchStorage. RAID Alternatives: Will Erasure Codes Rule? `http://searchstorage.techtarget.com/tip/RAID-alternatives-Will-erasure-codes-rule`.

[41] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File System Logging Versus Clustering: A Performance Comparison. In *Proc. of USENIX 1995 Technical Conference (TCON)*, 1995.

[42] J.-Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon. Gecko: Contention-oblivious Disk Arrays for Cloud Storage. In *Proc. of USENIX FAST*, Feb 2013.

[43] D. Stodolsky, G. Gibson, and M. Holland. Parity Logging Overcoming the Small Write Problem in Redundant Disk Arrays. In *Proc. of the 20th Annual International Symposium on Computer Architecture (ISCA)*, May 1993.

[44] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing Tape with Energy Efficient, Reliable, Disk-based Archival Storage. In *Proc. of USENIX FAST*, Feb 2008.

[45] A. Thomasian. Reconstruct Versus Read-modify Writes in RAID. *Inf. Process. Lett.*, 93(4):163–168, Feb 2005.

[46] Threadpool. `http://threadpool.sf.net/`.

[47] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, Mar 2002.

[48] C. Weddle, M. Oldham, J. Qian, and A. i Andy Wang. PARAID: A Gear-Shifting Power-Aware RAID. In *Proc. of USENIX FAST*, Feb 2007.

[49] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proc. of USENIX FAST*, Feb 2008.

[50] L. Xiang, Y. Xu, J. C. Lui, and Q. Chang. Optimal Recovery of Single Disk Failure in RDP Code Storage Systems. In *Proc. of ACM SIGMETRICS*, Jun 2010.

[51] F. Zhang, J. Huang, and C. Xie. Two Efficient Partial-Updating Schemes for Erasure-Coded Storage Clusters. In *Proc. of IEEE Seventh International Conference on Networking, Architecture, and Storage (NAS)*, Jun 2012.

(Big)Data in a Virtualized World: Volume, Velocity, and Variety in Cloud Datacenters

Robert Birke[+], Mathias Björkqvist[+], Lydia Y. Chen[+], Evgenia Smirni[*], *and* Ton Engbersen[+]
[+]*IBM Research Zurich Lab,* [*]*College of William and Mary*
[+]{*bir, mbj,yic,apj*}*@zurich.ibm.com,* [*]*esmirni@cs.wm.edu*

Abstract

Virtualization is the ubiquitous way to provide computation and storage services to datacenter end-users. Guaranteeing sufficient data storage and efficient data access is central to all datacenter operations, yet little is known of the effects of virtualization on storage workloads. In this study, we collect and analyze field data from production datacenters that operate within the private cloud paradigm, during a period of three years. The datacenters of our study consist of 8,000 physical boxes, hosting over 90,000 VMs, which in turn use over 22 PB of storage. Storage data is analyzed from the perspectives of volume, velocity, and variety of storage demands on virtual machines and of their dependency on other resources. In addition to the growth rate and churn rate of allocated and used storage volume, the trace data illustrates the impact of virtualization and consolidation on the velocity of IO reads and writes, including IO deduplication ratios and peak load analysis of co-located VMs. We focus on a variety of applications which are roughly classified as app, web, database, file, mail, and print, and correlate their storage and IO demands with CPU, memory, and network usage. This study provides critical storage workload characterization by showing usage trends and how application types create storage traffic in large datacenters.

1 Introduction

Datacenters provide a wide spectrum of data related services. They feature powerful computation, reliable data storage, fast data retrieval, and, more importantly, excellent scalability of resources. Virtualization is the key technology to increase the resource sharing in a seamless and secure way, while reducing operational costs without compromising performance of data related operations.

To optimize data storage and IO access in virtualized datacenters, storage and file system caching techniques have been proposed [13, 18, 28], as well as data duplication and deduplication techniques [22]. The central theme is to move the right data to the right storage tier, especially during periods of peak loads of co-located virtual machines (VMs). Therefore, it is crucial to understand the characteristics of IO workloads of individual VMs, as well as the workload seen by the hosting boxes. There are several storage-centric studies that have shed light on file system volume [14, 20, 31] and IO velocity, i.e., read/write data access speeds [15, 17, 28]. Despite these studies, it is unclear how virtualization impacts storage and IO demands at the scale of datacenters, and what their relationship to CPU, memory, and network demands are.

The objective of this paper is to provide a better understanding of storage workloads in datacenters from the following perspectives: storage volume, read/write velocity, and application variety. Using field data from production datacenters that operate within the private cloud paradigm, we analyze traces that correspond to 90,000 VMs hosted on 8,000 physical boxes, and containing over 22 PB of actively used storage, covering a wide range of applications, over a time span of three years, from January 2011 to December 2013. Due to the scale of the available data, we adopt a black-box approach in the statistical characterization of the various performance metrics. Due to the lack of information about the system topologies and the employed file system architectures, this study falls short in analyzing latency, file contents, and data access patterns at storage devices. Our analysis provides a multifaceted view of representative virtual storage workloads and sheds light on the storage management of highly virtualized datacenters.

The collected traces allow us to look at the volume of allocated, used, and free space in virtual disks per VM, with special focus on the yearly growth rate and weekly churn rate. We measure velocity by statistically characterizing the loads of read and write operations in GB/h as well as IO operations per second (IOPS) in multiple time

scales, i.e., hourly, daily, and monthly, focusing on characterization of the time variability and peak load analysis. We deduce the efficiency of storage deduplication in a virtualized environment, by analyzing the IO workload of co-located VMs within boxes. To see how storage and IO workloads are driven by different applications, we perform a per-application analysis that allows us to focus on a few typical applications, such as web, app, mail, file, database, and print applications, highlighting their differences and similarities in IO usage. Finally, we present a detailed multi-resource dependency study that centers on data storage/access and provides insights for the current state-of-the-practice in data management in datacenters.

Our findings can be quickly summarized as follows: VM capacity and used space have annual growth rates of 40% and 95%, respectively. The fullness per VM has a growth rate of 19%, though the distribution of storage fullness remain constant across VMs over the three years of the study. The lower bound of VM storage space churn rate is 17%, which is slightly lower than the churn rate of 21% reported from backup systems [31].

Regarding IO velocity, the IO access rates of boxes scales almost linearly with the number of consolidated VMs, despite the non-negligible overhead from virtualization. Both VMs and boxes are dominated by write workloads, with 11% of boxes experiencing higher virtual IO rates than physical ones. Deduplication ratios grow linearly with the degree of virtualization. Peak loads occur at off-hours and are contributed to a very small number of VMs. VMs with high velocity tend to have higher storage fullness and higher churn rates.

Regarding IO variety, different applications use storage in different ways, with file server applications having the highest volume, fullness, and churn rates. Databases have similar characteristics but low fullness. Overall, we observe that high IO demands strongly and positively correlate with CPU and network activity.

The outline of this work is as follows. Section 2 presents related work. Section 3 provides an overview of the dataset. The volume, velocity and variety analysis are detailed in Sections 4, 5 and 6, respectively. A datacentric multi-resource dependency study is discussed in Section 7, followed by conclusions in Section 8.

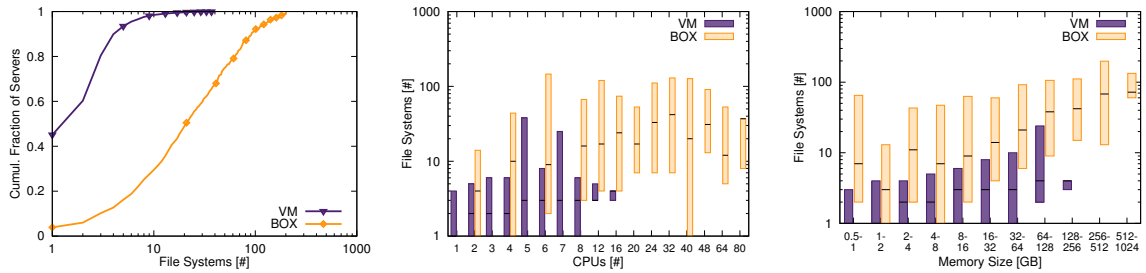
2 Related Work

Managing storage is an expensive business [19]. Coupled with the fact that the cost of storage hardware is several times that of server hardware, efficient use of storage for datacenters becomes critical [29]. Workload characterization studies of storage/IO are pivotal for the development of new techniques to better use systems, but it is difficult to define what is truly a representative system due to the wide variety of workloads. In general, from the various studies on file system workloads, those that stand out are the ones based on academic prototypes and those based on personal computers, in addition to a plethora of lower level storage studies. Virtualization adds additional layers of complexity to any storage media [10, 16]. As virtualization is indeed the standard for datacenter usage, workload studies of virtualized IO are important and relevant. Nonetheless, analyzing all relevant features of all relevant virtualized IO workloads is outside the scope of this work. Here, given the collected trace data, we conduct a statistical analysis with the aim of better understanding how IO occurs in a virtualized environment of a very large scale.

Typically, related work covers aspects of volume [2, 14, 20, 30], velocity [17] and variety, with a focus on file systems. Regarding file system volume, there are several studies that focus on desktop computers [2,14,20]. Using file system metadata during periods of four weeks [20] and five years [2], performance trends and statistics that shed light on fullness, counts of files/directories, file sizes, and file extensions are provided. Recognizing the need to better understand the behavior of backup workloads, Wallace et al. [31] present a broad characterization analysis and point out that the data churn rate is roughly 21% per week. Their study shows that the capacity of physical drives approximately doubles annually while their utilization only drops slightly. The study compares backup storage systems with primary storage ones and finds that their fullness is $60-70\%$ and $30-40\%$, respectively. Characterization of backup systems has been traditionally used to drive the development of deduplication techniques [20, 24].

Most works on IO characterization analysis focus on specific file systems within non-virtualized environments, e.g., NFS [7], CIFS [17], Sprite/VxFs [9], NTFS [25], and the EMC Data Domain backup system [31]. Common characteristics include large and sequential read accesses, increasing read/write ratios, bursty IO, and a small fraction of jobs accounting for a large fraction of file activities. Self-similar behavior [9] is identified and proposed to use to synthesize file system traffic. Backup systems [31] have been observed to have significantly more writes than reads, whereas file systems for primary applications have twice as many reads as writes [17].

Following the advances in virtualization technologies, several recent works focus on optimizing data storage and access performance in virtualized environments with an emphasis on novel shared storage design [11, 13] and data management [15, 18, 28]. To reduce the load on shared storage systems, distributed-like VM storage systems such as Lithium [13] and Capo [28] are proposed. Gulati et al. design and implement the concept of a stor-



(a) CDF (b) Number of CPUs (c) Memory size [GB]

Figure 1: Number of file systems associated with a VM and a box: (a) cumulative distribution, (b) boxplots of file systems as a function of the number of CPUs, and (c) boxplots of file systems as a function of memory size. The boxplots present the 10^{th}, 50^{th}, and 90^{th} percentiles.

age resource pool, shared across a large number of VMs, by considering IO demands of multiple VMs [11]. Systems that aim at improving IO load balancing for virtualized datacenters using performance models have been proposed [10, 23]. Combining intelligent caching, IO deduplication can be achieved by reducing duplicated data across different storage tiers, such as VMs, hosting boxes [18], and disks [15]. Everest [21] addresses the challenges of peak IO loads in datacenters by allowing data written to an overloaded volume to be temporarily off-loaded into a short-term virtual store. Nectar [12] proposes to interchangeably manage computation and data storage in datacenters by automating the process of generating data, thus freeing space of infrequently used data. Workload characterization that focuses on specific server workloads (i.e., application variety) such as web, database, mail, and file server, has been done for the purpose of evaluating energy usage [27]. Till now, only a rather small scale virtual storage workload characterization has been presented [28], pointing out that virtual desktop workloads are defined by their peaks.

The workload study presented here presents a broad overview of virtual machine storage demands at production datacenters, covering IO volume, velocity, and variety, and how these relate to the degree of virtualization as well as usage of other resources. The analysis presented here compliments many existing IO and file system studies by using a very large dataset from production datacenters in highly virtualized environments.

3 Statistics Collection

We surveyed 90,000 VMs, hosted on 8,000 physical servers in different data centers dispersed around the globe, serving over 300 corporate customers from a wide variety of industries, over a three year period and accounting for 22 PB of storage capacity. The servers use several operating systems, including Windows and different UNIX versions. VMware is the prevalent virtualization technology used. For a workload study on current virtualization practices, we direct the interested readers to [5].

The collected trace data is retrieved via `vmstat`, `iostat` and supervisor specific monitoring tools, and is collected for VMs as well as for physical servers, termed *hosting boxes*. Each physical box may host multiple (virtual) file systems, which are the smallest units of storage media considered in this study. To characterize data workloads in virtualized datacenters, we focus on three types of IO-related statistics for VMs.

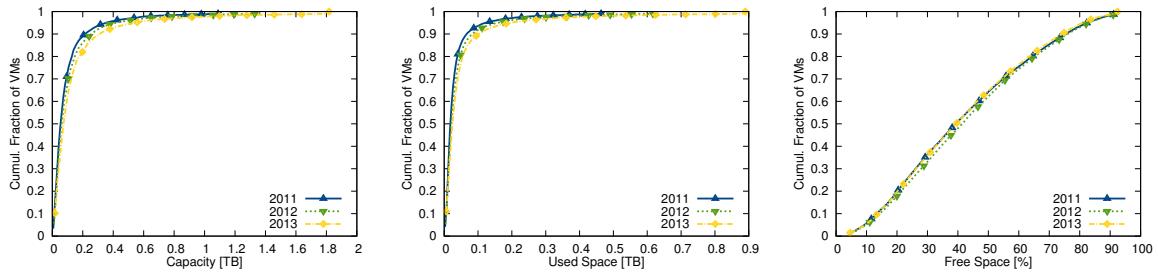
Volume refers to the allocated space, free space, and degree of *fullness*, defined as the ratio between the total used space and the total allocated space, of a VM after aggregating all of its file systems. Here, we focus on long-term trends, i.e., growth rates, and short-term variations, i.e., churn rates.

Velocity refers to read and write speeds measured in number of operations and transfered bytes per time unit, as IOPS and GB/h, respectively. We compare virtual IO velocity, measured at the VMs, with physical IO velocity, measured at the underlying boxes.

Variety refers to volume and velocity of *specific* applications, i.e., app, web, database, file, mail, and print, on specific VMs. To conduct storage-centric multi-resource analysis, we also collect CPU utilization, memory usage, and network traffic for VMs as well as boxes.

The trace data is available in two granularities: (1) in 15-minute/hourly averages from April 2013 and (2) coarse-grain monthly averages from January 2011 to December 2013. When exploring the differences between VMs in a day, we use the detailed traces with 15-minute/hourly granularity from 04/17 and 04/21. Monthly averages are used to derive long-term trends.

We note that the statistics of interest have long tails, therefore we focus on presenting CDFs as well as certain percentiles, i.e., 10^{th}, 50^{th} and 90^{th} percentiles. As the degree of virtualization (i.e., consolidation) on boxes is quite dynamic, we report on daily averages per phys-



(a) Storage Capacity (b) Used Volume (c) % of fullness

Figure 2: CDF of storage volume per VM over three years.

ical box. To facilitate the analysis connecting the per VM storage demands with the per file system storage demands, we present the CDF of the number of file systems across VMs and boxes (see Figure 1 (a)) and also how file system distributions vary across different systems, which we distinguish by the number of CPUs per box and memory (see boxplots in Figure 1 (b) and (c), respectively). Figure 1 (a) shows that boxes typically have a much higher (more than 21) number of virtual file systems than VMs, which have on the average 2 virtual file systems. Such values are very different from desktop studies [2] and underline the uniqueness of our dataset, especially in light of virtualized datacenters. Moreover, looking at the trends of medians in Figure 1 (b) and 1 (c), the number of file systems grows with servers equipped with more CPUs and, particularly, with larger memory.

As our data is obtained by standard utilities at the operating system level, we lack specific information about file systems, such as type, file counts, depth, and extensions. In addition, since the finest-grained granularity of the trace data is for 15-minute/hourly periods, IO peaks within such intervals cannot be observed. For example, the maximum GB/h within a day identified in this study is based on hourly averages, and is much lower than the instantaneous maximum GB/h. The coarseness of the information gathered is contrasted by the huge dataset of this study: 8,000 boxes with high average consolidation levels, i.e., 10 VMs per box, observed over a time-span of three years.

4 Volume

One of the central operations for datacenter management is to dimension storage capacity to handle short term as well as long term fluctuations in storage demand. These fluctuations are further accentuated by data deduplication and backup activities [6, 20]. Surging data demands and data retention periods drive storage decisions; however, existing forecasting studies either adopt a user or a per file system perspective, not necessarily targeting entire datacenters. Here, the aim is to adopt a different perspective and provide an overview on the yearly growth rates and weekly churn rates of storage demand at the VM level. In the following subsections, we focus on the storage demands placed by 90,000 VMs, their used/allocated storage and fullness, followed by statistical analysis of their yearly growth rates and weekly churn rates.

4.1 Data Storage Demands across VMs

Taking yearly averages of the monitored VMs over 2011, 2012, and 2013, we present how storage demands evolve over time and how they are distributed across VMs. Figure 2 (a) and 2 (b) present the CDF of the total sum of allocated and used storage volume per VM over all file systems belonging to each VM. Figure 2 (c) summarizes the resulting fullness. Visual inspection shows that the overall capacity and the used space per VM grow simultaneously, and result in fullness being constant over time. This observation illustrates a similar behavior as the one observed at the file system level [20], and provides information on how to dimension storage systems for datacenters where VMs are the basic compute units.

Via simple statistical fitting, we find that exponential distributions can capture well the VM storage demands in terms of allocated storage capacity and used storage volume. Table 1 summarizes the measured and fitted values, means and 95^{th} percentiles of capacity and used volume are reported. Since there are on average 10 VMs sharing the same physical box [5], a system needs to be equipped with 450 GB of storage space for very aggressive storage multiplexing schemes, i.e., only the used space is taken into account (45×10), or 1120 GB for a more conservative consolidation scheme based on the allocated capacity (112×10). The uniform distribution can approximately model fullness. Since the *relative ratio* of two independently exponential random variables is uniform [26], this further confirms that the exponential distribution is a good fit. Overall, the above analysis gives trends for the entire VM population, which in turn increases over the years, but does not provide any infor-

Table 1: Three year storage volume: measured and fitted data from exponential distribution.

	mean			95^{th}		
Year	2011	2012	2013	2011	2012	2013
Capacity [GB]	122	148	186	365	436	569
Exponential	122	148	186	365	442	556
Used [GB]	47	60	76	128	165	207
Exponential	47	60	76	140	180	228
Fullness [%]	42	44	42	83	83	81

mation on how the storage volume of individual VMs changes. In the following subsections, we focus on computing the yearly growth rate and weekly upper bound of the churn rate for each VM by presenting CDFs for the entire VM population.

On average, a VM has 2.55 file systems with a total capacity of 185 GB, of which roughly 42% is utilized, implying that each VM on average stores 77 GB of data. In general, the allocated capacity and free storage space increases over the years, while storage fullness remains constant.

4.2 Yearly Growth Rate

The data growth rate is predicted to double every two years [1]. Yet, it is still not clear how this value translates into growth at the per VM data volume level, or more importantly, whether the existing storage resources can sustain future data demands. Here, we analyze the long-term volume growth rates from two perspectives: supply, i.e., from the perspective of storage capacity, and demand, i.e., from the perspective of used storage volume.

In Figure 3, we show the CDF of the yearly relative growth rate of allocated capacity, used space, and fullness, across all VMs. We compute the relative yearly growth rate as the difference in used capacity between June 2012 and May 2013, and divide it by the start value. A positive (negative) growth indicates an increasing (decreasing) trend. Overall, the CDF of used space is very close to fullness, meaning that the storage space utilization is highly affected by the data demand, rather than by the supply of the capacity.

One can see that most VMs (roughly 86%) do not upgrade their storage, whereas the remaining 14% VMs increase their storage capacity quite significantly, i.e., up to 200%. Due to this long tail, the mean increase is 40.8%. As for the demand of space, almost all VMs increase their used storage. Only a small amount (below 25%) of VMs decrease their used space and have negative growth rates. On the other hand some VMs have a three-fold increase in used space. As a net result, the mean growth of used space is 95.1%. The smallest growth belongs to fullness: the mean rate is 19.1%. Such a value is higher

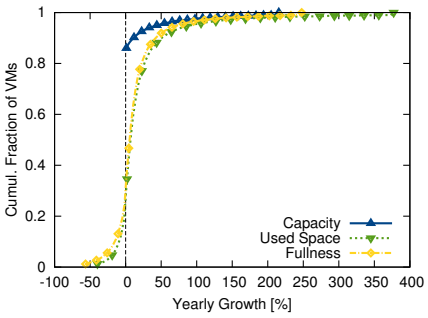


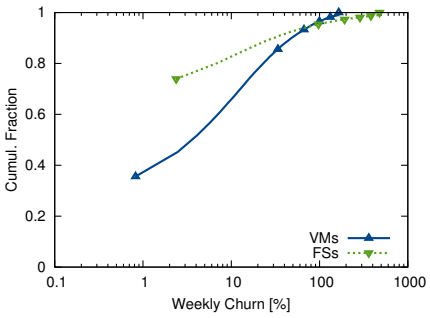
Figure 3: CDF of yearly growth rate of VM storage volume: capacity, used space, and fullness.

than the fullness trend evaluated across the entire VM population in Figure 2(c). Both storage capacity and used space increase over time for each individual VM with a mean yearly growth rate of 40% and 95%, respectively. The resulting fullness also increases by 19% every year.

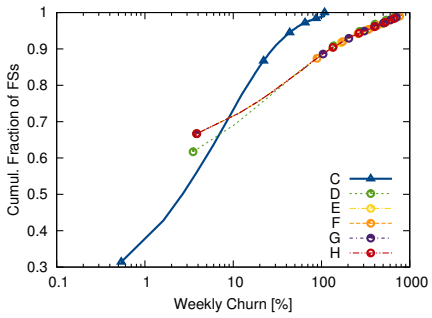
4.3 Weekly Churn Rate: Lower Bound

Here, we study short-term fluctuations of storage volume utilization, defined by the percentage of bytes that have been deleted during a time period of a week with respect to the used space. Note that this value represents a lower bound on the churn rate, since what is available in the trace is total volume in 15-minute intervals, i.e., if a VM writes and deletes the same amount of data within the 15-minute interval, there is no way to know how much is *truly* deleted during that period. We therefore report here a lower bound on the churn rate; the true value may be larger than the one reported here. The inverse of the lower bound of the churn rate reveals the upper bound of the data retention period. For example, a 20% weekly churn rate here means that the data is kept up to 5 weeks before being deleted. We base our computation of the weekly churn rate of VMs on the 15-minute data collected between 04/22/2013 to 04/28/2013. The churn rate is computed as the sum of all relative drops in used space, i.e., all negative differences between two adjacent 15-minute samples. We note that as data is also added over this one week time frame and we consider the sum of all deleted data, this value can go over 100%.

We present CDFs of two types of weekly churn rates in Figure 4 (a): by VMs and by file systems (FSs). The former gives the data volume deleted by VMs and the latter focuses on data volume deleted from an individual file system. Seen from the starting point and long tail of file system's CDF, a high fraction of file systems have a churn rate of zero, while a small fraction of file systems have very high churn rates. Thus a higher variability of churn rates is observed at file systems than at VMs. To



(a) VM and file system



(b) Specific file systems

Figure 4: CDF of the weekly churn rate computed based on single VM and a single file system: the x-axis is the percentage of storage space deleted in a week; the y-axis are the cumulative fraction of VMs, file systems, and specific file systems.

further validate this observation, we compute the churn CDF of the most commonly seen volume labels of file systems, i.e., C, D, E, F, G, and H, from Windows systems, that account for roughly 87% of the entire VM population. Shown in Figure 4 (b), one can clearly see that volume label C has very low churn rate, compared to the other labels. Such an observation matches with the common practice that C drives on Windows systems store program files that are rarely updated and other drives are used to store user data.

Overall, the churn rates of VMs have a mean around 17.9%, whereas churn rates of file systems have a mean around 20.8%. This value, being a lower bound, is on par with previous results in the literature, where a true churn rate, computed from detailed file system traces, is 21% [31]. Most VMs have rather low churn rate lower bounds; from Figure 4 (a) one can see that 75% of VMs have churn rates below 15%. However, 10% of VMs have a churn rate higher than 50%. VMs with high churn rates pose challenges for the storage system, because a large amount of space needs to be reclaimed and written.

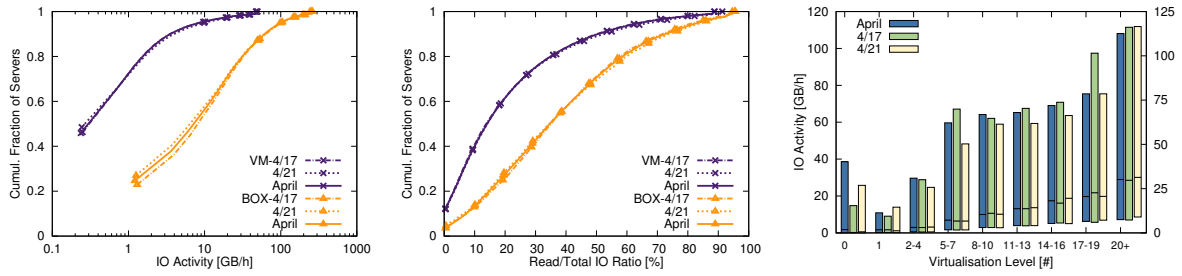
5 Velocity

The most straight-forward performance measure for storage systems is the IO speed, which we term velocity within the context of VMs accessing big data in data centers. The performance at peak loads [21] has long been a target focus for optimization. To expedite IO operations, caching [28] and IO deduplication [15] algorithms are critical. This is especially true within the context of virtualized data centers where the system stack, e.g., the additional hypervisor layer, for IO activities becomes deeper and more complex. The evaluation of caching and IO deduplication schemes in virtualized datacenters is usually done at small scale or lab-like environments [15, 28]. We quantify VM velocity via the speed by which data is placed in and retrieved from datacenter storage, and further pinpoint "hot" or "cold" VMs from the IO perspective. The statistics presented in the following subsections are based on hourly averages from 04/17/2013, which is shown representative for IO velocity in Section 5.1. The focus is on understanding their variability over time and their dependency on the virtualization level (i.e., on the number of simultaneous executing VMs), as well as on peak IO load analysis.

5.1 Overview

We start this section by presenting an overview of the daily velocity of VMs (and their corresponding boxes) in terms of (1) transferred data per hour (GB/h) including both read and write operations; and (2) the percentage of transferred data associated with read operations. Figure 5 depicts the aforementioned information in three types of statistics: the hourly average based on 04/17/2013 (weekday), 04/21/2013 (weekend), and daily average computed over the entire month of April 2013. The aim is to see if the IO velocity of a randomly selected date is sufficiently representative. Overall, the statistics of the daily velocity on 04/17/2013 are very close to those of a weekend day and to the statistics aggregated from the daily average over the entire April, see the almost overlapping lines in all three subfigures of Figure 5. Hence, in the rest of this paper we focus on a specific day 04/17/2013, which we consider as representative.

Shown by a lower CDF in Figure 5 (a), boxes have higher IO velocity than VMs. The average IO velocity for boxes and VMs are 26.7 GB/h and 2.9 GB/h, respectively, i.e., the velocity for boxes is larger roughly by a factor of 9. This factor is in line with the average consolidation level [5], i.e., 10 VMs per box and hints to a linear scaling of IO activity. Regarding the percentage of read operations, boxes have heavier read workloads than VMs do, as shown by the CDF curve in Figure 5 (b)



(a) IO in GB/h (b) Percentage of reads (c) IO by virtualization level

Figure 5: Daily velocity: IO read and write activities per VM and box on 4/17, 4/21, and the entire April.

that corresponds to boxes. There is roughly 12% of VMs having only write workloads, as indicated by the leftmost point of the VM CDF. Meanwhile, less than 1% of VMs have read workloads only. Indeed, the mean read ratio of boxes and VMs are 38% and 21%, respectively. Overall, the velocity of VMs and boxes is dominated by write workloads.

To verify how the virtualization level affects the box IO activity, we group the box IO activity by virtualization level and present the 10^{th}, 50^{th}, and 90^{th} percentiles, see the boxplots in Figure 5 (c). The box IO activity increases almost linearly with the virtualization level, this can be seen by the 50^{th} percentile. When further normalizing the IO velocity of a box by the number of consolidated VMs, the average values per box drop slightly with the virtualization level. This implies that there is a non-negligible fixed overhead associated with virtualization. We omit this graphical presentation due to lack of space.

5.2 Deduplication of Virtual IO

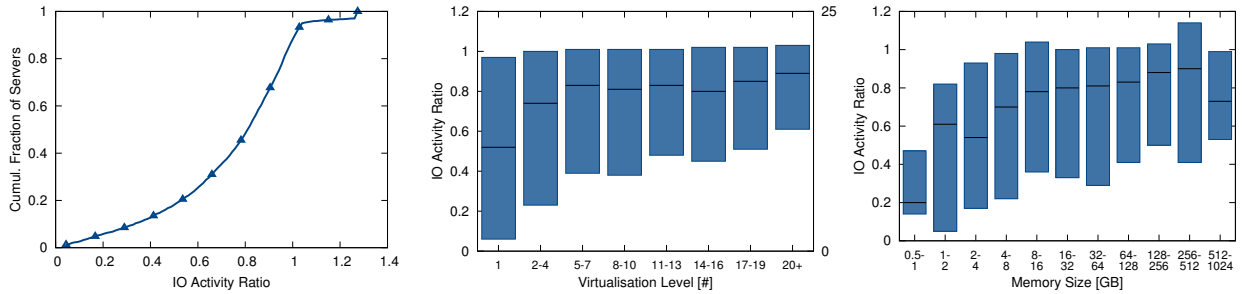
IO deduplication techniques [15] are widely employed to reduce the amount of IO. The discussion in this section is limited to virtual IO since, from the traces, there is no way to distinguish how and where the data is deduplicated and/or cached. We compare the sum of all virtual IO activity aggregated over all consolidated VMs within a box, termed virtual IO, divided by the IO activity measured at the underlying physical box, termed box IO, and call this ratio the *virtual deduplication ratio*. In contrast to the rest of the paper, we here use IOPS as the measurement of velocity, instead of GB/h. When the deduplication ratio is greater (or smaller) than one, the virtual IO is higher (or lower) than the physical box IO, respectively. A deduplication ratio of one is used as the threshold between deduplication and amplification.

We summarize the CDF of the deduplication ratio in Figure 6 (a). Roughly 50% of boxes have a deduplication ratio ranging from 0.8 to 1.2, i.e., close to one, indicating similar IO activities at the physical and virtual levels. Another observation is that most boxes experience amplification, as indicated by deduplication ratios less than one (including close to one), i.e., virtual IO loads are lower than physical IOs. This can be explained by the fact that hypervisors induce IO activities due to VM management, e.g., VM migration.

There is a very small number of boxes (roughly 11%) experiencing IO deduplication, as indicated by the boxes having deduplication ratios greater than one. To understand the cause of such deduplication, we compute the separate deduplication ratio for read and write activities. We see that the observed deduplication stems more from read than write operations, as indicated by a higher fraction of boxes (roughly 18%) having deduplication read ratios greater than one. One can relate this observation to the fact that read caching techniques are more straightforward and effective than write caching techniques.

To see how virtualization affects deduplication ratios, we group the deduplication ratios by their virtualization level and present them using boxplots, see Figure 6 (b). Looking at the lower and middle bars of each boxplot, i.e., the 10^{th} and 50^{th} percentiles, we see that the deduplication ratios increase with the virtualization level. Such an observation can be explained by the fact that IO activities of co-located VMs have certain dependencies that further present opportunities for reducing IO operations for hypervisors. Higher virtualization levels can lead to better IO deduplication. We note that similar observations and conclusions can be deduced by using IO in GB/h, with the deduplication ratios roughly ranging between 0 to 3.

In addition to virtualization, the effectiveness of IO deduplication can be highly dependent on the cache size. Unfortunately, our data set does not contain information about cache sizes, only memory sizes, which in turn are often positively correlated to the cache sizes. Therefore, to infer the dependency between cache size and IO deduplication ratio, we resort to memory size and categorize deduplication ratios by box memory sizes, see Figure 6 (c). The trend is that the IO deduplication ratio increases with increasing memory size, though with a drop for systems having memory greater than 512 GB.



(a) CDF (b) By virtualization level (c) By memory size

Figure 6: Virtual IO deduplication/amplification per box: $\frac{Virtual\_IOPS}{Physical\_IOPS}$

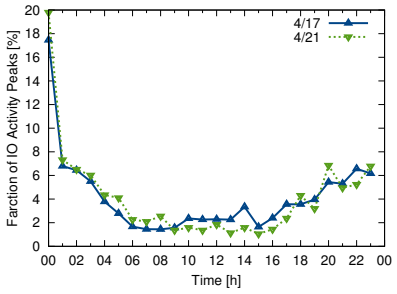


Figure 7: PDF of virtual loads peak times in a day over all consolidated VMs.

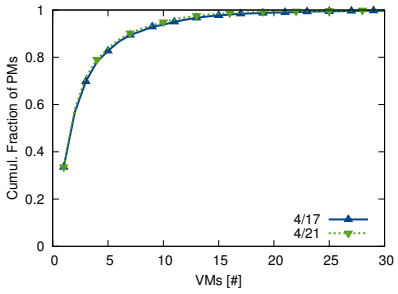


Figure 8: Number of VMs to reach 80% of peak load over all consolidated VMs.

5.3 Peak Velocity of Virtual IO

Virtualization increases the randomness of access patterns due to the general lack of synchronized activity between the VMs and the larger data volume accessed, which in turn imposes several challenges to IO management [8]. The first question is how IO workloads fluctuate over time. To such an end, for each VM and box, we compute their coefficient of variation (CV) of the IO activity in GB/h during a day using the hourly data. The higher the CV value, the higher the variability of the IO workload during the day. Our results show that boxes have rather stable IO velocity with an average CV of around 0.8, while VMs have an average CV of around 1.3.

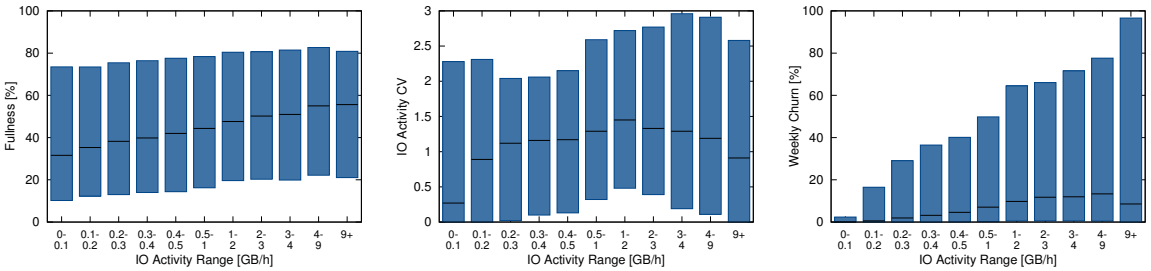
The confirmation of higher time variability of VMs lead us to focus on the characteristics of virtual IO aggregated over all VMs hosted on the same box, in particular their peak loads. We try to capture when the peaks of aggregated velocity happen, and how each VM contributes to the peak. We do this both for a Wednesday (04/17/2013) and a Sunday (04/21/2013) based on the hourly IO activity data.

5.3.1 Peak Timings

Figure 7 presents the empirical frequencies showing which hour of the day the aggregated virtual peak IO loads happen. Clearly, most VMs have peaks during after-hours, i.e., between 6pm to 6am, for both days. This observation matches very well with timings for peak CPU [4] and peak network [3] activities but does not match the belief that IO workloads are driven by the working hours schedule [18]. Indeed, in prior work [5] we have observed that most VM migrations occur during midnight/early morning hours, which is consistent with the activity seen in Figure 7. Clearly, the intensity of virtual IO workloads is affected by background activities such as backup and update operations that are typically run during after-hours.

5.3.2 Top VM Contributors

Another interesting question is how consolidated VMs contribute to peak loads. Information on top VM contributors to peak loads is critical for improving peak load performance via caching [21, 28]. We define as top contributors the co-located VMs having the highest contributions to the peak load in order to reach a certain threshold, i.e., 80% of the peak load in this study. We summarize the distribution of the number of top VM contrib-



(a) Fullness (b) Time variability (c) Weekly churn rate

Figure 9: Cold vs Hot VMs: volume, time variability in a day, and weekly storage space churn. The x-axis is IO in GB/h and y-axes are fullness [%], coefficient of variation (CV), and weekly churn rate.

utors for both days in Figure 8. Interestingly, one can see a clear trend indicating that it is very common that a small number of VMs dominates peak loads for both days. Such a finding is similar to the one reported in [28], where only independent (i.e., not co-located) VMs are considered. These results further show that making a priority the optimization of the IO of a few top VMs may have a large impact on overall performance.

5.4 Characteristics of Cold/Hot VMs

Motivated by the fact that a few number of VMs contribute to peak loads, we try to capture the characteristics of VMs based on their IO activity in GB/h, aiming to classify the VMs as cold/hot. The hotness of the data is very useful to dimension and tier storage systems; e.g., cold data in slow storage media and hot data in flash drives. To this end, we compare the used volume, time variability, and churn rate of VMs grouped by different levels of IO activity, see Figure 9 (a), (b), and (c), respectively. Each box represents a group of VMs having an average activity falling into the IO activity range shown on the x-axis.

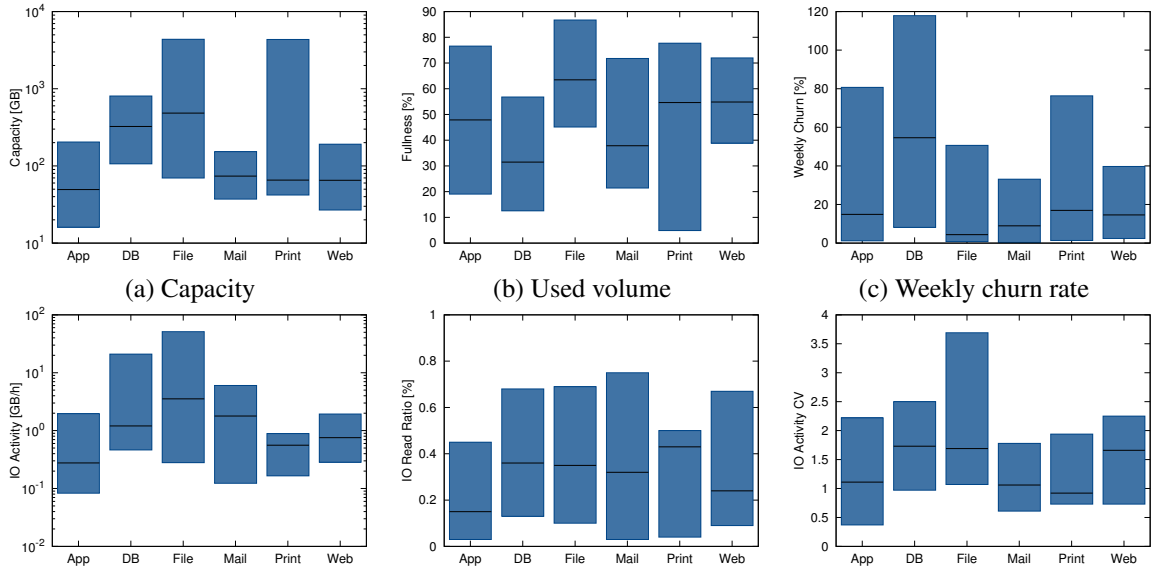
The 50^{th} percentile, i.e., the middle bar in each boxplot, increases with the IO activity level for both the fullness and churn rate. Overall, VMs with high IO activities are also fuller and have higher churn rates, compared to VMs with low IO activities. For fullness, not only the 50^{th} percentile, but also the entire boxes shift with the IO activity level. To see if the reverse is also true, we classify the IO activity level by different levels of used space both in GB and percentage. The data shows that high space usage indeed results in high IO activity, especially when measured in GB. However, VMs with very full storage systems, i.e., 90-100% occupancy, have slightly lower IO activity than VMs with 80-90%. This stems from the fact that most storage systems have optimal performance when they are not completely full. A common rule of thumb is that the best performance is achieved when the used space is up to 80%. Hence, only cold data is placed on disks with a higher percentage of used space. Due to space constraints, we omit the presentation of this set of results.

The time variability shows a different trend, i.e., the CV first increases as IO velocity increases but later decreases, see Figure 9 (b). The hottest VMs, i.e., the ones with IO greater than 9 GB/h, have the second lowest CV, as can be seen from the 50^{th} percentile. We thus conclude that hot VMs have relatively constant, high IO loads across time.

Regarding churn rates, both the 50^{th} and 90^{th} percentiles clearly grow with IO activity levels, indicating strong correlation between IO activity and churn. Such an observation matches very well with common understanding that hot VMs have frequent reads/writes, resulting in frequent data deletion and short data retention periods. This is confirmed by our data showing quantitatively that 50% of hot VMs, i.e., VMs having an IO activity level of 9 GB/h or more, have data retention periods ranging between 11.11 ($1/0.09$) and 1.02 ($1/0.98$) weeks. In summary, hot VMs have higher volume consumption (55%) and churn rates (9%).

6 Variety

The trace data allows to distinguish application types for a subset of VMs. Here, we select the following applications: *app*, *web*, *database (DB)*, *file*, *mail*, and *print*, and characterize their volume and velocity. Our aim here is to provide quantitative as well as qualitative analysis that could be used in application-driven optimization studies for storage systems. The app servers host key applications for clients, such as business analytics. DB servers run different database technologies, such as DB2, Oracle, and MySQL. File servers are used to remotely store files. Due to business confidentiality, it is not possible to provide detailed information about these applications. We summarize the storage capacity, used space, weekly churn rate, IO velocity, percentage of read operations, and time variability using boxplots for each application



(a) Capacity (b) Used volume (c) Weekly churn rate

(d) IO in GB/h (e) Read ratio (f) Time variability (CV)

Figure 10: Application's storage volume and IO velocity.

type, see Figure 10. We mark the 10^{th}, 50^{th}, and 90^{th} percentile of VMs belonging to each application. Most statistics are based on the data collected on 04/17/2013, except for the weekly churn rate that is based on data between 04/22/2013 to 04/28/2013.

Storage Capacity: File VMs have the highest capacities, followed by DB VMs – see the relative values of their respective 50^{th} percentiles. Mail, print, web, and app have similar storage capacities, but print VMs have the highest variance – see the height of the boxplot.

Volume: Fullness shows a slightly different trend from the allocated storage capacity. File VMs are also the fullest, hence they store the largest data volume. Database VMs that have the second highest allocated capacities are now the least full, hinting to large amounts of free space. In terms of variability of fullness across VMs in the same application type, print VMs still have very different storage fullness.

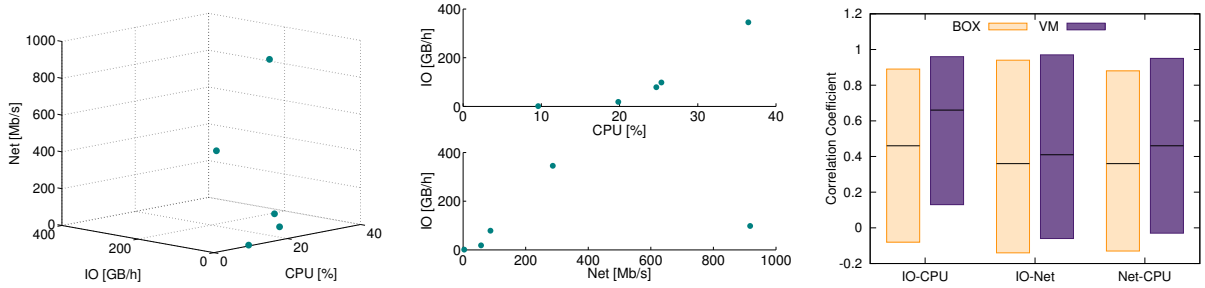
Weekly Churn Rate: DB VMs have the highest weekly churn rate, with some VMs having churn rates greater than 120%, hinting to frequent updates where a lot of storage volume is deleted and reclaimed. Unfortunately, due to the coarseness of the trace data, we cannot confirm whether this is due to the `tmp` space used for large queries, although this is a possible explanation. Such an observation goes hand-in-hand with low fullness of DB. Based on the value of 50^{th} percentile, print VMs have the second highest churn rate, as print VMs store many temporary files, which are deleted after the print jobs are completed. Due to dynamic contents, app and web VMs have high churn rates as well, i.e., similar to the mean churn rate of 17.9% shown in Section 4.3.

IO Velocity: Applying characteristics of hot/cold VMs summarized in Section 5, it is no surprise that file VMs have the highest IO velocity, measured in GB/h. According to the 50^{th} percentile, mail and DB VMs have the second and third highest IO velocity. Print, web, and app VMs experience similar access speeds.

Read/Write Ratio: All application VMs have their 50^{th} percentile of read ratio less than 50%, i.e., all application types have more write intensive operations than read operations. Indeed, as discussed in Section 5, VMs are more write intensive. Among all, app VMs have the lowest read ratio, i.e., lower than 20%. In contrast, print VMs have the highest read ratio close to 50%, which is reasonable as print VMs have rather symmetric read/write operations, i.e., write files to storage and read them for sending to the printers.

Time Variability: To see the IO time variability per application, we use their CV across a day, computed from 24 hourly averages. DB and file show high time variability by their 50^{th} percentile being around 1.8. As web VMs frequently interact with users who have strong time of day patterns, web VMs exhibit time variability as high as file and DB VMs. Mail, print, and app VMs have their CV slightly higher than 1, i.e., IO activities are spread out across the day.

In summary, file VMs have the highest volume, velocity and IO load variability, but with a rather low weekly churn rate around 10%. DB VMs have high volume, velocity, IO load variability and churn rate, but with very low fullness. Mail VMs have moderate volume, and high velocity evenly across the day. All application VMs are write intensive.



(a) VM workload centroids (b) IO-CPU and IO-Net projections (c) Correlation coefficients

Figure 11: Dependency among IO [GB/h], CPU[%], Network [Mb/s].

7 Interdependency of CPU and Network

Since the statistical analysis presented here is based on the perspective of VMs and boxes, it is possible to correlate the storage workloads with those of other resources, in particular CPU and network. Using hourly averages from 04/17/2013, we capture the dependency of VM IO activities on CPU utilization and network traffic measured in megabits per second (Mb/s). We focus on the following two questions: (1) what are the most representative patterns of IO, CPU, and network usage; and (2) what is the degree of dependency among these three resources. For the first question, we use K-means clustering to find the representative VM workloads. For the second question, we use the correlation coefficients for each VM for any pair of IO, CPU, and network, and summarize their distributions.

7.1 Representative VM Workloads

When presenting the VMs' daily average IO, CPU, and network by means of a three dimensional scatter plot, there are roughly 90,000 VM points. Due to the unavoidable over-plotting, there is no obvious pattern that can be identified via visual inspection. To identify representative VM workloads, we resort to K-means clustering. Due to the lack of a priori knowledge on the number of VM clusters, we first vary the target number of clusters from 3 to 20 to observe clustering trends over an increasing number of clusters. Our results show that the overall trajectories of cluster centroids are consistent across different number of clusters. In Figure 11 (a), we present the centroids of 5 clusters. When the cluster number further increases beyond 5, more centroids appear on the line between the first two lowest centroids.

To take an IO-centric perspective, we analyze the representative VM workloads by looking at projections of VM centroids on the IO-CPU and IO-network planes, see Figure 11 (b). When looking at the IO-CPU plane, we see that IO workloads increase with CPU utilization in an exponential manner. The VM centroid with the highest IO (around 342GB/h), i.e., the rightmost point, has the highest CPU utilization (around 36%). In the IO-network plane the trend is less clear. One can observe that the first four VM centroids roughly lie on a line having their network traffic increasing at the same rate as their IO velocity. However, the last VM centroid with the highest network traffic (around 917Mb/s) has a relatively low IO activity (around 97GB/h). Overall, the majority of representative VMs have IO workloads that increase commensurately with CPU loads and network traffic, while very IO intensive VMs tend to heavily utilize the CPU but not the network.

7.1.1 Correlation Coefficients

In Figure 11 (c), we present the 10^{th}, 50^{th}, and 90^{th} percentiles of the correlation coefficients of IO-CPU, IO-network, and CPU-network. To compute correlation coefficients of the aforementioned three pairs, for each VM/box, we use three time series of 24 hourly averages: IO GB/h, CPU Utilization, and network traffic.

Among all three pairs, IO-CPU shows the highest correlation coefficients, especially for VMs. The 50^{th} percentile of the IO-CPU correlation coefficient for VMs and boxes is around 0.65 and 0.45, respectively. This indicates that IO activities closely follow CPU activities. Such an observation is consistent with the clustering results. The correlation coefficients for boxes are slightly lower than for those of VMs. Indeed, there is a certain fraction of boxes and VMs that exhibit negative dependency, and this is observed more prominently between IO and network. As for the network-CPU pair, VMs and boxes demand both resources roughly in a similar manner, supported by that fact that the correlation coefficient values are mostly above zero.

8 Conclusions

We conducted a very large scale study in virtualized, production datacenters that operate under the private cloud

paradigm. We analyze traces that correspond to the activity across three years of 90,000 VMs, hosted on 8,000 physical boxes, and containing more than 22 PB of actively used storage. IO and storage activity is reported from three viewpoints: volume, velocity, and variety, i.e., we take a holistic view of the entire system but also look at individual applications. This workload characterization study differs from others from its sheer size both from observation length and number of traced systems. Yet while some of our findings confirm those reported on smaller studies, some others provide a different perspective. Overall, the degree of virtualization is identified as an important factor in perceived performance, ditto for the per application storage requirements and demand, pointing to directions to focus on for better resource management of virtualized datacenters.

Acknowledgements

We thank the anonymous referees and our shepherd, Garth Gibson, for their feedback that has greatly improved the content of this paper. This work has been partly funded by the EU Commission under the FP7 GENiC project (Grant Agreement No 608826). Evgenia Smirni has been partially supported by NSF grants CCF-0937925 and CCF-1218758, and by a William and Mary Plumeri Award.

References

[1] *Big Data Drives Big Demand for Storage* `http://www.idc.com/getdoc.jsp?containerId=prUS24069113`. 2013.

[2] AGRAWAL, N., BOLOSKY, W., DOUCEUR, J., AND LORCH, J. A five-year study of file-system metadata. In *FAST* (2007), pp. 3–3.

[3] BIRKE, R., CHEN, L. Y., AND MINKENBERG, C. A datacenter network tale from a server's perspective. In *IEEE IWQoS* (2012), pp. 1–10.

[4] BIRKE, R., CHEN, L. Y., AND SMIRNI, E. Data centers in the cloud: A large scale performance study. In *IEEE CLOUD* (2012), pp. 336–343.

[5] BIRKE, R., PODZIMEK, A., CHEN, L. Y., AND SMIRNI, E. State-of-the-practice in data center virtualization: Toward a better understanding of vm usage. In *IEEE/IFIP DSN* (2013), pp. 1–12.

[6] DUBNICKI, C., GRYZ, L., HELDT, L., KACZMARCZYK, M., KILIAN, W., STRZELCZAK, P., SZCZEPKOWSKI, J., UNGUREANU, C., AND WELNICKI, M. Hydrastor: A scalable secondary storage. In *FAST* (2009), pp. 197–210.

[7] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. Passive NFS tracing of email and research workloads. In *FAST* (2003).

[8] EVANS, C. *Dedicated VM storage emerges to meet virtualisation demands.* `http://www.computerweekly.com/feature/Dedicated-VM-storage-emerges-to-meet-virtualisation-demands`. 2013.

[9] GRIBBLE, S. D., MANKU, G. S., ROSELLI, D. S., BREWER, E. A., GIBSON, T. J., AND MILLER, E. L. Self-similarity in file systems. In *SIGMETRICS* (1998), pp. 141–150.

[10] GULATI, A., SHANMUGANATHAN, G., AHMAD, I., WALDSPURGER, C. A., AND UYSAL, M. Pesto: online storage performance management in virtualized datacenters. In *SoCC* (2011), p. 19.

[11] GULATIAND, A., SHANMUGANATHAN, G., ZHANG, X., AND VARMAN, P. Demand based hierarchical qos using storage resource pools. In *USENIX ATC* (2012), pp. 1–14.

[12] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *OSDI* (2010), pp. 75–88.

[13] HANSEN, J. G., AND JUL, E. Lithium: virtual machine storage for the cloud. In *SoCC* (2010), pp. 15–26.

[14] J.DOUCEUR, AND BOLOSKY, W. A large-scale study of file-system contents. In *SIGMETRICS* (1999), pp. 59–70.

[15] KOLLER, R., AND RANGASWAMI, R. I/O deduplication: utilizing content similarity to improve I/O performance. In *FAST 2010*, pp. 16–16.

[16] LE, D., HUANG, H., AND WANG, H. Characterizing datasets for data deduplication in backup applications. In *FAST* (2012), pp. 1–10.

[17] LEUNG, A. W., PASUPATHY, S., GOODSON, G. R., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *USENIX ATC* (2008), pp. 213–226.

[18] LI, M., GAONKAR, S., BUTT, A. R., KENCHAMMANA, D., AND VORUGANTI, K. Cooperative storage-level de-duplication for I/O reduction in virtualized data centers. In *MASCOTS* (2012), pp. 209–218.

[19] MERRILL, D. R. *Storage economics: Four principles for reducing total cost of ownership. http://www.hds.com/assets/pdf/four_principles_for_reducing_total_cost_of_ownership.pdf*. 2009.

[20] MEYER, D., AND BOLOSKY, W. A study of practical deduplication. In *FAST* (2011), pp. 1–1.

[21] NARAYANAN, D., DONNELLY, A., THERESKA, E., ELNIKETY, S., AND ROWSTRON, A. I. T. Everest: Scaling down peak loads through i/o offloading. In *OSDI* (2008), pp. 15–28.

[22] NG, C.-H., MA, M., WONG, T.-Y., LEE, P. P. C., AND LUI, J. C. S. Live deduplication storage of virtual machine images in an open-source cloud. In *Middleware* (2011), pp. 81–100.

[23] PARK, N., AHMAD, I., AND LILJA, D. J. Romano: autonomous storage management using performance prediction in multi-tenant datacenters. In *SoCC* (2012), p. 21.

[24] PARK, N., AND LILJA, D. J. Characterizing datasets for data deduplication in backup applications. In *IISWC* (2010), pp. 1–10.

[25] ROSELLI, D., LORCH, J., AND ANDERSON, T. A comparison of file system workloads. In *USENIX ATC* (2000), pp. 41–54.

[26] ROSS, S. *A First Course in Probability*. 2004.

[27] SEHGAL, P., TARASOV, V., AND ZADOK, E. Evaluating performance and energy in file system server workloads. In *FAST* (2010), pp. 253–266.

[28] SHAMMA, M., MEYER, D., WIRES, J., IVANOVA, M., HUTCHINSON, N., AND WARFIELD, A. Capo: Recapitulating storage for virtual desktops. In *FAST* (2011), pp. 31–45.

[29] SIMPSON, N. *Building a data center cost model. http://www.burtongroup.com/Research/DocumentList.aspx?cid=49*. 2009.

[30] VOGELS, W. File system usage in windows nt 4.0. *SIGOPS Oper. Syst. Rev. 33*, 5 (Dec. 1999), 93–109.

[31] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., MARK, M. C., AND HSU, W. Characteristics of backup workloads in production systems. In *FAST* (2012), pp. 4–4.

From research to practice: experiences engineering a production metadata database for a scale out file system

Charles Johnson[1], Kimberly Keeton[1], Charles B. Morrey III[1], Craig A. N. Soules[2], Alistair Veitch[3], Stephen Bacon, Oskar Batuner, Marcelo Condotta, Hamilton Coutinho, Patrick J. Doyle, Rafael Eichelberger, Hugo Kiehl, Guilherme Magalhaes, James McEvoy, Padmanabhan Nagarajan, Patrick Osborne, Joaquim Souza, Andy Sparkes, Mike Spitzer, Sebastien Tandel, Lincoln Thomas, and Sebastian Zangaro

HP Labs[1] Natero[2] Google[3] HP Storage

Abstract

HP's StoreAll with Express Query is a scalable commercial file archiving product that offers sophisticated file metadata management and search capabilities [3]. A new REST API enables fast, efficient searching to find all files that meet a given set of metadata criteria and the ability to tag files with custom metadata fields. The product brings together two significant systems: a scale out file system and a metadata database based on LazyBase [10]. In designing and building the combined product, we identified several real-world issues in using a pipelined database system in a distributed environment, and overcame several interesting design challenges that were not contemplated by the original research prototype. This paper highlights our experiences.

1 Introduction

Unstructured data, which accounts for more than 90% of the information in the world today [11], creates a number of challenges, including economically storing the data (even as it ages), effectively protecting and managing it, and extracting value from the stored data. To help customers tame their information explosion, HP wanted to provide an archival storage solution that would scale to billions of files and objects and create structure for unstructured data by allowing customers to exploit rich metadata services.

To help with the problem of extracting value, the solution would need to provide fast metadata search to support a variety of usage scenarios. For example, system administrators need to quickly and efficiently find files that match a given criteria to monitor storage operation (e.g., identify files created, modified, or deleted within a given time frame) and enforce compliance (e.g., determine which files are approaching retention expiration, or are on legal hold). Users want to "tag" files with custom metadata attributes and later search using those attributes. Such metadata services would also benefit external applications like backup and enterprise content management, by allowing them to avoid costly file system scans when determining which files have changed and must be backed up or indexed.

Ad hoc solutions in this space couple together an external relational DBMS and a scale out file store. This approach is unable to support the necessary scaling and performance requirements. Additionally, such solutions do not provide integrated search capabilities across system and custom metadata, and are likely to be expensive to maintain. Instead, our goal was to embed the metadata service within the file system to solve these challenges.

StoreAll with Express Query is a file archiving solution that couples a scale-out file system with an embedded database to accelerate metadata queries [3]. Initial releases target archival workloads, where files must be kept for an extended period of time, may be actively searched and may subject to business or regulatory requirements. In these systems, the number of files and aggregate data size can be extremely large, due to the need to retain files for many years.

This paper describes our experiences transforming a research metadata database (LazyBase [10]) into a production-quality metadata database, Express Query. In our work, we discovered several issues prompted by the scalable file archiving use case that we had not considered in the research prototype, and re-evaluated several of our original design decisions.

We begin by providing background on LazyBase and the scale out file system (§2). We highlight some of the challenges we encountered and overcame (§3), as well as the new capabilities we added to improve usability and flexibility (§4). Finally, we overview the related work (§5) and summarize the lessons we learned (§6).

2 Background

This section provides an overview of the original LazyBase [10] design and of the StoreAll file system architecture.

2.1 LazyBase

Express Query is based on LazyBase, a distributed database that provides scalable, high-throughput ingest of updates, while allowing a per-query tradeoff between latency and result freshness [10]. LazyBase provides this tradeoff using an architecture designed around batching and pipelining of updates. Read queries observe a stale, but consistent, version of the data, which is sufficient for many applications; more up-to-date results can be obtained when needed by scanning updates still being processed by later stages of the pipeline.

LazyBase provides a service model that decouples update processing from read-only queries. Updates (e.g., adds, modifies, deletes) are *observational*, meaning that data additions and modifications must provide new or updated values, which will overwrite (or delete) existing data. Because data is batched, uploaded (potentially) out-of-order and processed asynchronously, it may not be possible to read the "current" value of a field to determine the new/updated value; the most recent update may not have been uploaded yet or may still be being processed by the pipeline.

To improve database ingest performance, update clients (also known as *sources*) batch updates together and upload them to LazyBase as a single *self-consistent update (SCU)*, which is the granularity of transactional (e.g., ACID) properties throughout the update pipeline. For read-only queries, LazyBase provides snapshot isolation, where all reads in a query will see a consistent snapshot of the database, as of the time that the query started; in practice, this is the last SCU that was applied at query start time.

LazyBase tables contain an arbitrary number of named and typed columns. Each table has a *primary sort order* and one or more optional *secondary sort orders* (analogous to materialized views), which contain a subset of the columns and rows of the primary sort order. Each sort order is a collection of fixed-size pages, called *extents*, which are stored in compressed form. Additionally, each sort order has an *extent index*, which stores the minimum and maximum value of the key in each extent of the underlying sort order. Because extents are typically large (64KB), and the index only stores min and max values, the index is small enough to fit into memory, even if the table is very large. As a result, LazyBase requires fewer disk I/Os to locate a data extent through the extent index than would be required for a traditional B-tree index. Primary and secondary sort orders, as well as extent indexes, are stored as DataSeries files [9].

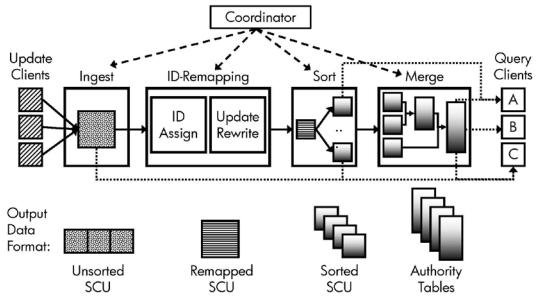


Figure 1: LazyBase prototype architecture [10].

Figure 1 illustrates LazyBase's update processing pipeline. The *ingest* stage accepts client uploads and makes them durable. The *ID-remapping stage* converts SCUs from using their internal temporary IDs to using global IDs common across the system. The *sort* stage sorts each of the SCU's tables for each of its sort orders. The *merge* stage combines multiple SCUs into a single sorted SCU. In addition to these stages, a coordinator tracks and schedules work in the system, maintaining availability and managing recovery.

2.2 StoreAll architecture

StoreAll's shared nothing clustered file system is subdivided into segments (volumes). Each segment contains a portion of the inodes (directories and files) in the file system. A segment is owned by one server, and the file system supports failover to other servers if the owning server fails. Each server handles reads and writes and manages locking for inodes in the segments it owns. A server can access an inode owned by another server in the cluster via internal network handshaking. The system supports NFS, CIFS, HTTP, FTP, and local file system access and scales to more than 16 PB of data in a single name space.

As the file system is updated, the system records metadata state changes (e.g., file creations, deletions, retention operations) into a per-segment *archive journal*. This journal is a transactionally reliable change log of file system metadata updates that each server maintains for the segments that it manages. Every few seconds the *archive journal writer* (`ajwriter`) flushes the archive journal files (`ajfiles`) for the segments owned by that

server; for each segment, the `ajwriter` closes the existing `ajfile` and starts a new one. Once the `ajfiles` are closed, they appear in the StoreAll namespace in a hidden directory and an update notification is sent to the subscribers of the `ajwriter`. This distributed publish/-subscribe event-driven architecture scales out well because changes are recorded locally and immediately. It avoids expensive file system scans for metadata changes and provides a difficult-to-bypass auditing mechanism.

3 Lessons Learned

Incorporating LazyBase into the StoreAll product pushed our initial LazyBase design in interesting new directions. In this section, we highlight several of the lessons learned, including the demands of the file system use case, the limits of our initial design, and how we addressed the challenges. We believe that these lessons and our solutions generalize to using a system like LazyBase in other distributed environments.

3.1 Transaction model complications

The combination of observational updates, out-of-order events and asynchronous processing complicates the transactional model. Here, we describe three aspects of the problem and our solutions: out-of-order event processing, expressing freshness, and enforcing data integrity.

3.1.1 Out-of-order event processing

Depending on the order in which batches are uploaded, events may be processed by the database in a different order than they were generated in the file system. LazyBase's pipeline has built-in support for processing out-of-order updates. It uses both per-field and per-row timestamps, and makes no assumptions about where the timestamps come from, only that the timestamps generated for updates to a particular field must be totally ordered. When merging multiple versions of a given row, LazyBase compares the timestamps of all versions of a field and takes the newest.

In the research prototype for LazyBase, we used the event timestamps in the input data as the field timestamps. We assumed that all updates for a given field could be globally ordered based on their timestamps. In the product, we had to cope with the fact that event timestamps associated with the same file system object could be generated by different servers with skewed clocks. The clock skew issue prompted changes in the way we track event timestamps for StoreAll.

In StoreAll, servers that host client connections are called *entry servers (ESs)*. ESs initiate file system operations on one or more file system objects on behalf of their clients. However, durable modifications caused by these operations are made only at the server that owns the file system object; such servers are called *destination servers (DSs)*. Any ES in the system can initiate an operation that results in durable modifications to a file system object. Operations that do not generate any durable modifications (e.g., `read` and `getattr`) can be supported via caching on the ES, without requiring communication with the DS that owns the object. As in all distributed systems, the clocks on the individual ES and DS nodes will have skew.

Ultimately, we eliminated the clock skew issue by using the DS timestamp for all events that make durable modifications to file system objects. We use the ES timestamp to support read auditing, with the proviso that these timestamps are not comparable to those in non-audit tables and using the knowledge that audit events are never updated after insertion.

3.1.2 Freshness

The LazyBase research prototype expressed freshness as a single number. In contrast, in a distributed system such as StoreAll, where multiple servers upload new data to Express Query, freshness can't be expressed as a single number. As described in § 3.2, updates are batched independently for different segments, meaning that it is not possible to provide a single point-in-time view of the entire file system's metadata. Instead, the freshness provided by Express Query is a range, delimited by the oldest and newest of the freshness levels from individual segments. Segment freshness levels are affected by a number of issues, including events being cached before being flushed to an `ajfile` (as described in §2.2), or a segment going offline for a time and only uploading events once it comes back online.

To simplify the early Express Query design, we disabled freshness queries. Even though database clients cannot request a particular freshness, they still need to know about the achieved freshness of their query results. For example, a periodic backup application that queries for recently updated files and wants to start its next backup where the previous one left off needs to know the freshness for the previous query results to avoid missing modified files. To address this need, Express Query explicitly tracks each segment's freshness, and query results include the minimum (*FreshnessComplete*) and maxi-

mum (*FreshnessPartial*) freshness values across the segments. FreshnessComplete indicates the timestamp before which all events have been observed from all segments. FreshnessPartial indicates the timestamp for the latest event processed for any segment. Thus, in the window between FreshnessComplete and FreshnessPartial, query results include some, but not all, of the events generated in the file system. Database clients can use this information to determine how to use the query results.

3.1.3 Enforcing data integrity

As described in § 2.1, the combination of observational updates, out-of-order event arrival and asynchronous processing means that LazyBase does not support read-modify-write transactions. This property has interesting implications for file system event processing. For example, custom attributes for an old version of a file should no longer be visible once the file has been deleted. However, since StoreAll users need to be able to add an arbitrary number of custom attributes for a file, so we organized the schema to store custom attributes in a different table (with one row per attribute) from the rest of the system attributes (with one row per file system object). This meant that file deletions couldn't automatically delete custom attributes, because there was no way to reliably read and delete the up-to-date set of custom attributes when processing the deletion event.

Instead, we needed to explicitly enforce integrity constraints between the tables. Express Query tracks file creation and deletion times, as well as timestamps for custom metadata operations, and queries must include timestamp comparison logic to check for attribute validity. A lazy cleaning pass periodically gets rid of custom attributes for deleted files as well as file lifetime information for files that were created or deleted sufficiently long ago.

3.2 Batching

As Cipar et al. observed, the choice of batch size causes a tradeoff between ingest throughput and latency [10]. Larger batches lead to greater pipeline processing efficiency (and hence better throughput), but also increase the delay before data can be queried – essentially, this decreases the freshness of the query results. We considered increasing batch size by including updates from multiple sources in the same batch, but quickly realized that this complicates the transactional model: it is more difficult for individual sources to abort, when the other sources in the same batch want to commit. As a result, we elected to create independent batches for different sources.

Express Query treats each file system segment as a source. A user-space tool called the *archive journal scanner*, or `ajscanner`, subscribes to the `ajwriter` notifications (§2.2). For each `ajfile`, the `ajscanner` parses the event data to create a batch of updates to upload to Express Query. The `ajscanner` processes `ajfiles` for each segment in order (determined using the `ajfiles`' `mtimes`), and uploads data from different segments in parallel. From Express Query's perspective, each segment appears as a separate source, uploading a stream of SCUs, one per `ajfile`. We use the fact that `ajfiles` are created regularly every few seconds to strike a balance between pipeline throughput and pipeline latency (freshness).

3.3 Auto-increment IDs

The LazyBase research prototype supported the concept of a 64-bit integer auto-increment ID column, also known as a database surrogate key [7]. IDs can more space-efficiently represent long values (e.g., file pathnames), by substituting the ID wherever the value would have been used in a table. The exact savings depends on a variety of factors, including the length of the strings, the strings' compressibility, and how many string fields are present in a table. The expectation was that by converting long values into integers, the ID-remapping mechanism would improve ingestion performance. Indeed, we found that using IDs sped up merge performance for a simulated file creation benchmark by an average of 54%. However, ID-remapping has both query and ingestion costs that must be considered.

The LazyBase prototype included IDs for a variety of string fields, including pathnames, and used these IDs as the primary key for most tables. Because LazyBase uses in-memory extent indexes to support point and range queries, sorting a table by the ID effectively randomized the data order, requiring a full table scan for what otherwise should be point or range queries. Furthermore, every query that selected or filtered on an ID-remapped attribute (in combination with other attributes) required a join with the ID table. In the file system context, this meant that all pathname-based queries (e.g., "find all files in a directory" or "show pathnames for all files modified in the last day") required a join between the path ID table and the table(s) containing the other metadata; often, these other tables required full table scans. In contrast, if IDs were not used and pathnames were included in the tables containing the other metadata attributes, with a sort order by pathname, path-based lookups could have been satisfied by an indexed lookup to the table(s) con-

Experiment	IDs (sec)	No IDs (sec)
File lookup	55.16 +/- 4.23	0.12 +/- 0.14
Directory lookup (small)	509.83 +/- 12.51	0.44 +/- 0.03
Directory lookup (med)	819.42 +/- 105.11	8.28 +/- 0.10

Table 1: ID vs. no-ID execution time (in seconds) for file and directory lookup queries, for 100M file dataset. Values shown are average +/- standard deviation for ten trials. The small directory lookup examines about 148k files; the medium directory lookup examines about 3.84M files. Directory lookups compute the max file size to eliminate output processing costs.

taining the other attributes. As shown in Table 1[1], the combination of full table scans and joins proved to be unacceptably inefficient.

The ingestion costs proved to be non-trivial, as well. The ID-remap stage must look up each incoming value to determine what global ID to assign, which requires all prior SCUs to be queryable and thus violates the goal of delayed processing for efficiency. Because the preceding individual SCUs may not have been merged into larger SCUs, remapping may require reading input data from many files, with the number of I/Os depending on the distribution of values in the input data. Additionally, the ID-remap stage proved to be a scalability bottleneck: since processing is serialized due to the need to look at all prior SCUs, the stage can only be scaled by partitioning the namespace. Although parallelizing ID-remap would help ingest-time scalability, it still would not address the query-time concerns described above.

Our solution was to eliminate the use of auto-incrementing IDs and the ID-remap stage entirely. This approach improved query performance dramatically and simplified many stages of the pipeline, including the coordinator job scheduling and recovery processing.

3.4 Primary key

Our initial Express Query design used pathname as the primary key for most tables, to transparently support backup/restore and remote replication, which preserve pathnames. This choice worked well for the archival use cases we initially targeted, where files were almost never modified after being created, and were not renamed. However, to support a more general file system use case, the system needed to provide support for renames and hard links. Unfortunately, with pathname as the primary key, this more general use case required reassigning the primary key, a costly operation. As a result,

[1]The equipment used for all experiments is an HP DL380p Gen8 server (2 x Intel Xeon E5-2697v2 CPUs, 2.70 GHz, 12 cores, 24 hyperthreads) with 384GB of DRAM. LazyBase/Express Query data is stored on an HP D2700 disk array with a P822 RAID controller and 25 146GB 15k RPM SAS drives.

	Primary sort order (sec)	Secondary sort order (sec)
Point key	129.08 +/- 4.17	0.05 +/- 0.01
Range (10%)	131.48 +/- 2.94	16.97 +/- 0.17
Range (25%)	136.44 +/- 2.60	39.68 +/- 0.34
Range (50%)	138.52 +/- 4.91	77.60 +/- 0.37
Range (75%)	142.02 +/- 3.67	115.80 +/- 1.00

Table 2: Execution time (in seconds) for point and range queries for primary sort order (table scan) vs. secondary sort order (index lookup), for 100M file dataset. Values shown are average +/- standard deviation for ten trials. The table shows range query results for four different selectivities (fraction of rows used to calculate result). Range queries compute a count to eliminate output processing costs.

the next version of our design chose as its primary key a globally unique file system-internal identifier for all file system objects. Tables continue to store the file system object's pathname and to define a secondary sort order based on the pathname, to avoid the auto-increment ID issues described in §3.3.

3.5 Secondary sort orders

As with any data management system, a universal challenge is how to organize the data to balance between query cost efficiency and data maintenance efficiency. In Express Query, this challenge amounts to which secondary sort orders to maintain, and how many columns each secondary sort order should contain.

For queries that filter on a secondary sort order's search key, the sort order provides efficient indexed lookups. Table 2 compares query execution time for indexed lookups vs. full table scans. If the secondary sort order is populated with a sufficiently large subset of the columns of the primary sort order, then a single secondary sort order can satisfy queries that access multiple attributes. For example, a query to select all pathnames, file sizes and file owners for files that have been recently modified could be efficiently satisfied by a secondary sort order that is sorted according to `mtime` and also contains the pathname, size and owner.

Creating and maintaining secondary sort orders during the update pipeline requires resources, however. The more secondary sort orders and the more columns per secondary sort order, the longer ingesting takes, and hence the freshness of the queryable data suffers. Table 3 quantifies the cost of update pipeline processing for additional fully-populated secondary sort orders.

To reap the potential query-time performance benefits from secondary sort orders, our initial Express Query schema maintained a fully-populated secondary sort order for each of the system attributes in the file objects table. We continue to experiment with reducing the num-

	Primary only	Primary + 15 secondary	Slowdown
Durable	1965 sec	6939 sec	3.53X
Queryable	2379 sec	11157 sec	4.69X

Table 3: Update pipeline processing time (in seconds) for ingesting 100M simulated file creations. "Durable" is time until the data is made durable (i.e., through the ingest pipeline stage). "Queryable" is time until the data is queryable (i.e., through the complete pipeline, including ingest). "Primary only" is a schema with no secondary sort orders for the file object data. "Primary + 15 secondary" is a schema with 15 fully-populated secondary sort orders, one per system attribute.

ber of secondary sort orders and the fraction of columns in various secondary sort orders, to improve ingest resource utilization and query freshness.

4 New Features

The goals for StoreAll's metadata database were to support user-initiated operations, such as assigning custom metadata tags to files, efficiently performing ad hoc file searches (e.g., a fast Unix `find`) and generating file system utilization reports. Additionally, the database needed to support external applications, such as a backup service tracking recently changed files. Finally, it needed to support internal file system operations, such as content validation scans and storage tiering policies. The query API needed to be flexible in the face of schema changes, and to facilitate rapid prototyping and experimentation by developers of the file system services using the database. The end user-visible interface needed to be intuitive and simple.

This section describes two APIs – SQL and REST – that we implemented to improve usability and flexibility for internal and external users of the database, respectively. The system continues to support programmatic queries where flexibility is not required, or performance overrides other considerations.

4.1 SQL API

We added a full SQL front end to Express Query, using the foreign data wrapper (FDW) API from PostgreSQL [5]. We define FDWs on top of the Express Query native tables, using the DataSeries (DS) storage layer and translation logic to access the tables. SQL queries are parsed, optimized, and partially executed by PostgreSQL, using foreign table accesses (table scans and index lookups) at the leaf nodes of the query execution tree, instead of native PostgreSQL table or index scans. Our approach uses multiple components: a Transaction Manager, a DS FDW, a DS row iterator, and a shim layer to translate between the FDW and row iterator. These components cooperate to request data from the Express Query pipeline workers, perform data translation operations, and implement transactional properties.

The *Transaction Manager* keeps track of active transactions and which versions of the Express Query tables they access, to ensure that all table accesses in the same transaction see a consistent view of the underlying database (i.e., per-transaction snapshot isolation). This mapping also informs garbage collection: the Transaction Manager prevents the garbage collector from reclaiming any versions that are still in use by an active transaction.

FDW. The FDW interfaces with the rest of PostgreSQL's query execution engine. It allows query qualifications (e.g., conditions in a SQL SELECT WHERE statement) to be passed to Express Query, to permit filtering of the rows examined to satisfy the query, rather than requiring a full table scan. Only qualifications with $=$, $<$, $<=$, $>$, $>=$, or LIKE operators on search keys are passed through, because they can be used by Express Query's index interface.

Translation shim. For each foreign table involved in a query, this layer communicates with the rest of Express Query to register the foreign table's transaction id with the Transaction Manager and learn which ingest pipeline worker(s) to contact to retrieve the data. The shim layer translates PostgreSQL's generic data types into Express Query data type-specific values, and prepares the DS search keys from the PostgreSQL qualifications. It uses these search key(s) to request data from the Express Query ingest worker(s) for the table.

DS row iterator. This layer applies the appropriate equality or range search key filters, and returns data from the Express Query pipeline worker one row at a time.

With this breakdown, the FDW needs no knowledge of Express Query, and Express Query needs no knowledge of PostgreSQL.

4.2 REST API

Although Express Query's SQL read query front end met the goal of enabling ad hoc queries, it did not isolate end users from the specifics of the database schema. To provide a simpler and more flexible interface, we defined a REST API [6], to permit users to request file and directory attributes, search for all paths matching a set of attribute criteria, and define custom attributes.

File-mode REST requests ("queries" in REST parlance) have three components: the path to be queried, the at-

tributes to be returned, and the query expression itself. In addition, several options specify recursive search, limitations on the number of results returned, and result order. The API supports both system and custom attributes. System attributes include the attributes stored in the file's inode (e.g., `size` and `mode`), as well as attributes particular to StoreAll's retention-enabled file system (e.g., storage tier, retention state). The API also provides attributes that summarize the last activity for a file (e.g., content modifications, custom metadata changes, file creations and deletions); we added these attributes to help database clients like backup providers efficiently discover what files had recent changes, to facilitate their own operations (e.g., choosing which files to back up). Users can also specify their own custom attributes, which are associated with paths as string key-value pairs.

We automatically translate each REST API query into a SQL query to retrieve the relevant metadata; results are presented in JSON.

5 Related work

Spyglass [12] provides an engine customized for file metadata indexing and querying. It leverages the property that files have many common attributes (e.g., owner and path prefix) to optimize index structures. Exploiting these properties achieves very high query performance, but sacrifices flexibility, in that the system does not support arbitrary user-specified attributes. Instead of constantly updating as the file system changes, Spyglass relies on efficient scans of periodic snapshots, which can result in highly variable freshness, depending on how often snapshots are taken. It also prevents the system from offering auditing capabilities, but enables a valuable feature in historical metadata search.

A number of systems (e.g., [4, 8, 1, 2, 13]) offer full file system search capabilities that can include metadata attributes. They typically rely on either some form of inverted index (fast for queries, but expensive to update and rebuild) or rely on a conventional RDBMS, which severely limits their scalability and performance properties. (Our early experiments with using both open-source and commercial RDBMSs for this purpose motivated the original LazyBase research.) By focusing on keyword search, these systems are somewhat orthogonal to our purposes, as they are not customized for metadata-intensive applications; many do not even index file metadata. Many of these systems also do not allow for custom metadata, rely on inefficient file system scans, or are not integrated into the kernel, and thus cannot offer auditing.

6 Conclusions

This paper highlights some of our experiences transforming a research prototype of a pipelined database into a production metadata database in a scale out file system. We summarize these experiences as follows:

Fallacies in our initial design. Despite our initial intuition, auto-incrementing IDs and ID-remapping provided unacceptable query and ingest performance slowdowns; therefore we removed them. We also realized that in a distributed environment, freshness is a window, not a single number; this complexity compelled us to disable freshness queries and report the achieved freshness range as part of query results.

Usability and flexibility sometimes override performance. Although our initial focus was on performance of the update pipeline and a fast programmatic query API, we learned that the flexibility to do ad hoc queries and rapid prototyping merited the inclusion of a SQL query API. Similarly, the desire to provide a simple interface that isolated users from schema changes prompted the development of a REST API.

Issues that we hadn't considered, motivated by our use case. LazyBase's lack of read-modify-write transactions meant that some data integrity constraints (e.g., custom attribute suppression for deleted files) needed to be explicitly enforced. Similarly, our initial choice of pathname as a primary key, while convenient for our initial archive use case, proved to be the wrong choice for a more general file system use case.

Modifications to the environment to ensure LazyBase assumptions hold. For example, we forced batches to contain only updates from a single source to ensure isolation between sources. Additionally, we forced timestamps on a particular field to have a total ordering, to ensure that LazyBase's out-of-order processing worked correctly.

Need to balance ingest-time and query-time processing. We observed tensions between ingest processing efficiency and query performance when selecting batch sizes and choosing which secondary sort orders to include in the schema. As in most data management systems, such design decisions must balance these competing demands.

7 Acknowledgments

We thank Jiri Schindler, our shepherd; Steven Hand; and the anonymous reviewers for constructive comments that have significantly improved the paper.

References

[1] Apache Solr. http://lucene.apache.org/solr/, Jan. 2014.

[2] Autonomy. http://www.autonomy.com/, Jan. 2014.

[3] HP StoreAll with Express Query. http://www.hp.com/go/storeall/, Jan. 2014.

[4] Introduction to Spotlight. https://developer.apple.com/library/mac/documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.html, Jan. 2014.

[5] PostgreSQL. http://www.postgresql.org/, Jan. 2014.

[6] Representational state transfer. http://en.wikipedia.org/wiki/Representational_state_transfer, Jan. 2014.

[7] Surrogate key. http://en.wikipedia.org/wiki/Surrogate_key, Jan. 2014.

[8] Windows search. http://windows.microsoft.com/en-us/windows7/products/features/windows-search, Jan. 2014.

[9] ANDERSON, E., ARLITT, M., MORREY III, C. B., AND VEITCH, A. DataSeries: An efficient, flexible data format for structured serial data. *ACM SIGOPS Operating Systems Review 43*, 1 (January 2009), 70–75.

[10] CIPAR, J., GANGER, G., KEETON, K., MORREY III, C. B., SOULES, C. A. N., AND VEITCH, A. LazyBase: Trading freshness for performance in a scalable database. In *Proc. of European Systems Conference (EuroSys)* (April 2012), pp. 169–182.

[11] GANTZ, J., AND REINSEL, D. Extracting value from chaos. *IDC report* (June 2011).

[12] LEUNG, A. W., SHAO, M., BISSON, T., PASUPATHY, S., AND MILLER, E. L. Spyglass: Fast, scalable metatdata search for large-cale storage systems. In *Proc. 7th USENIX Conf. on File and Storage Technologies FAST* (2009), pp. 153–166.

[13] MANBER, U., AND WU, S. Glimpse: A tool to search through entire file systems. In *Proc. of the Winter 1994 USENIX Conference* (San Francisco, CA, 1994), pp. 23–32.

Analysis of HDFS Under HBase: A Facebook Messages Case Study

Tyler Harter, Dhruba Borthakur†, Siying Dong†, Amitanand Aiyer†,
Liyin Tang†, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin, Madison † Facebook Inc.

Abstract

We present a multilayer study of the Facebook Messages stack, which is based on HBase and HDFS. We collect and analyze HDFS traces to identify potential improvements, which we then evaluate via simulation. Messages represents a new HDFS workload: whereas HDFS was built to store very large files and receive mostly-sequential I/O, 90% of files are smaller than 15MB and I/O is highly random. We find hot data is too large to easily fit in RAM and cold data is too large to easily fit in flash; however, cost simulations show that adding a small flash tier improves performance more than equivalent spending on RAM or disks. HBase's layered design offers simplicity, but at the cost of performance; our simulations show that network I/O can be halved if compaction bypasses the replication layer. Finally, although Messages is read-dominated, several features of the stack (i.e., logging, compaction, replication, and caching) amplify write I/O, causing writes to dominate disk I/O.

1 Introduction

Large-scale distributed storage systems are exceedingly complex and time consuming to design, implement, and operate. As a result, rather than cutting new systems from whole cloth, engineers often opt for *layered* architectures, building new systems upon already-existing ones to ease the burden of development and deployment.

Layering, as is well known, has many advantages [23]. For example, construction of the Frangipani distributed file system [27] was greatly simplified by implementing it atop Petal [19], a distributed and replicated block-level storage system. Because Petal provides scalable, fault-tolerant virtual disks, Frangipani could focus solely on file-system level issues (*e.g.*, locking); the result of this two-layer structure, according to the authors, was that Frangipani was "relatively easy to build" [27].

Unfortunately, layering can also lead to problems, usually in the form of decreased performance, lowered reliability, or other related issues. For example, Denehy *et al.* show how naïve layering of journaling file systems atop software RAIDs can lead to data loss or corruption [5]. Similarly, others have argued about the general inefficiency of the file system atop block devices [10].

In this paper, we focus on one specific, and increasingly common, layered storage architecture: a distributed database (HBase, derived from BigTable [3]) atop a distributed file system (HDFS [24], derived from the Google File System [11]). Our goal is to study the interaction of these important systems, with a particular focus on the lower layer; thus, our highest-level question: is HDFS an effective storage backend for HBase?

To derive insight into this hierarchical system, and thus answer this question, we trace and analyze it under a popular workload: Facebook Messages (FM) [20]. FM is a messaging system that enables Facebook users to send chat and email-like messages to one another; it is quite popular, handling millions of messages each day. FM stores its information within HBase (and thus, HDFS), and hence serves as an excellent case study.

To perform our analysis, we first collect detailed HDFS-level traces over an eight-day period on a subset of machines within a specially-configured *shadow cluster*. FM traffic is mirrored to this shadow cluster for the purpose of testing system changes; here, we utilize the shadow to collect detailed HDFS traces. We then analyze said traces, comparing results to previous studies of HDFS under more traditional workloads [14, 16].

To complement to our analysis, we also perform numerous simulations of various caching, logging, and other architectural enhancements and modifications. Through simulation, we can explore a range of "what if?" scenarios, and thus gain deeper insight into the efficacy of the layered storage system.

Overall, we derive numerous insights, some expected and some surprising, from our combined analysis and simulation study. From our analysis, we find writes represent 21% of I/O to HDFS files; however, further investigation reveals the vast majority of writes are HBase overheads from logging and compaction. Aside from these overheads, FM writes are scarce, representing only 1% of the "true" HDFS I/O. Diving deeper in the stack, simulations show writes become amplified. Beneath HDFS replication (which triples writes) and OS caching (which absorbs reads), 64% of the final disk load is write I/O. This write blowup (from 1% to 64%) emphasizes the importance of optimizing writes in layered systems, even for especially read-heavy workloads like FM.

From our simulations, we further extract the following conclusions. We find that caching at the DataNodes

is still (surprisingly) of great utility; even at the last layer of the storage stack, a reasonable amount of memory per node (*e.g.*, 30GB) significantly reduces read load. We also find that a "no-write allocate" policy generally performs best, and that higher-level hints regarding writes only provide modest gains. Further analysis shows the utility of server-side flash caches (in addition to RAM), *e.g.*, adding a 60GB SSD can reduce latency by 3.5x.

Finally, we evaluate the effectiveness of more substantial HDFS architectural changes, aimed at improving write handling: local compaction and combined logging. Local compaction performs compaction work within each replicated server instead of reading and writing data across the network; the result is a 2.7x reduction in network I/O. Combined logging consolidates logs from multiple HBase RegionServers into a single stream, thus reducing log-write latencies by 6x.

The rest of this paper is organized as follows. First, a background section describes HBase and the Messages storage architecture (§2). Then we describe our methodology for tracing, analysis, and simulation (§3). We present our analysis results (§4), make a case for adding a flash tier (§5), and measure layering costs (§6). Finally, we discuss related work (§7) and conclude (§8).

2 Background

We now describe the HBase sparse-table abstraction (§2.1) and the overall FM storage architecture (§2.2).

2.1 Versioned Sparse Tables

HBase, like BigTable [3], provides a *versioned sparse-table* interface, which is much like an associative array, but with two major differences: (1) keys are ordered, so lexicographically adjacent keys will be stored in the same area of physical storage, and (2) keys have semantic meaning which influences how HBase treats the data. Keys are of the form *row:column:version*. A *row* may be any byte string, while a *column* is of the form *family:name*. While both column families and names may be arbitrary strings, families are typically defined statically by a schema while new column names are often created during runtime. Together, a row and column specify a cell, for which there may be many versions.

A sparse table is sharded along both row and column dimensions. Rows are grouped into *regions*, which are responsible for all the rows within a given row-key range. Data is sharded across different machines with region granularity. Regions may be split and re-assigned to machines with a utility or automatically upon reboots. Columns are grouped into families so that the application may specify different policies for each group (*e.g.*, what compression to use). Families also provide a locality hint: HBase clusters together data of the same family.

2.2 Messages Architecture

Users of FM interact with a web layer, which is backed by an application cluster, which in turn stores data in a separate HBase cluster. The application cluster executes FM-specific logic and caches HBase rows while HBase itself is responsible for persisting most data. Large objects (*e.g.*, message attachments) are an exception; these are stored in Haystack [25] because HBase is inefficient for large data (§4.1). This design applies Lampson's advice to "handle normal and worst case separately" [18].

HBase stores its data in HDFS [24], a distributed file system which resembles GFS [11]. HDFS triply replicates data in order to provide availability and tolerate failures. These properties free HBase to focus on higher-level database logic. Because HBase stores all its data in HDFS, the same machines are typically used to run both HBase and HDFS servers, thus improving locality. These clusters have three main types of machines: an *HBase master*, an *HDFS NameNode*, and many *worker* machines. Each worker runs two servers: an *HBase RegionServer* and an *HDFS DataNode*. HBase clients use the HBase master to map row keys to the *one* RegionServer responsible for that key. Similarly, an HDFS NameNode helps HDFS clients map a pathname and block number to the *three* DataNodes with replicas of that block.

3 Methodology

We now discuss trace collection and analysis (§3.1), simulation (§3.2), validity (§3.3), and confidentiality (§3.4).

3.1 Trace Collection and Analysis

Prior Hadoop trace studies [4, 16] typically analyze default MapReduce or HDFS logs, which record coarse-grained file events (*e.g.*, creates and opens), but lack details about individual requests (*e.g.*, offsets and sizes). For our study, we build a new trace framework, HTFS (Hadoop Trace File System) to collect these details. Some data, though (*e.g.*, the contents of a write), is not recorded; this makes traces smaller and (more importantly) protects user privacy.

HTFS extends the HDFS client library, which supports the arbitrary composition of layers to obtain a desired feature set (*e.g.*, a checksumming layer may be used). FM deployments typically have two layers: one for normal NameNode and DataNode interactions, and one for fast failover [6]. HDFS clients (*e.g.*, RegionServers) can record I/O by composing HTFS with other layers. HTFS can trace over 40 HDFS calls and is publicly available with the Facebook branch of Hadoop.[1]

[1] https://github.com/facebook/hadoop-20/blob/master/src/hdfs/org/apache/hadoop/hdfs/APITraceFileSystem.java

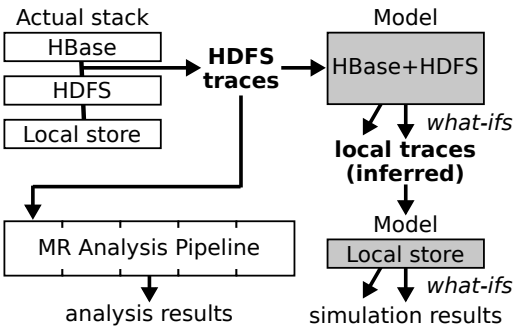


Figure 1: **Tracing, analysis, and simulation.**

We collect our traces on a specially configured *shadow cluster* that receives the same requests as a production FM cluster. Facebook often uses shadow clusters to test new code before broad deployment. By tracing in an HBase/HDFS shadow cluster, we were able to study the real workload without imposing overheads on real users. For our study, we randomly selected nine worker machines, configuring each to use HTFS.

We collected traces for 8.3 days, starting June 7, 2013. We collected 116GB of gzip-compressed traces, representing 5.2 billion recorded events and 71TB of HDFS I/O. The machines each had 32 Xeon(R) CPU cores and 48GB of RAM, 16.4GB of which was allocated for the HBase cache (most memory is left to the file-system cache, as attempts to use larger caches in HBase cause JVM garbage-collection stalls). The HDFS workload is the product of a 60/34/6 get/put/delete ratio for HBase.

As Figure 1 shows, the traces enable both analysis and simulation. We analyzed our traces with a pipeline of 10 MapReduce jobs, each of which transforms the traces, builds an index, shards events, or outputs statistics. Complex dependencies between events require careful sharding for correctness. For instance, a stream-open event and a stream-write event must be in the same compute shard in order to correlate I/O with file type. Furthermore, sharding must address the fact that different paths may refer to the same data (due to renames).

3.2 Modeling and Simulation

We evaluate changes to the storage stack via simulation. Our simulations are based on two models (illustrated in Figure 1): a model which determines how the HDFS I/O translates to local I/O and a model of local storage.

How HDFS I/O translates to local I/O depends on several factors, such as prior state, replication policy, and configurations. Making all these factors match the actual deployment would be difficult, and modeling what happens to be the current configuration is not particularly interesting. Thus, we opt for a model which is easy to understand and plausible (*i.e.*, it reflects a hypothetical policy and state which could reasonably occur).

Our model assumes the HDFS files in our traces are replicated by nine DataNodes which co-reside with the nine RegionServers we traced. The data for each RegionServer is replicated to one co-resident and two remote DataNodes. HDFS file blocks are 256MB in size; thus, when a RegionServer writes a 1GB HDFS file, our model translates that to the creation of twelve 256MB local files (four per replica). Furthermore, 2GB of network reads are counted for the remote replicas. This simplified model of replication could lead to errors for load balancing studies, but we believe little generality is lost for caching simulations and our other experiments. In production, all the replicas of a RegionServer's data may be remote (due to region re-assignment), causing additional network I/O; however, long-running FM-HBase clusters tend to converge over time to the pattern we simulate.

The HDFS+HBase model's output is the input for our local-store simulator. Each local store is assumed to have an HDFS DataNode, a set of disks (each with its own file system and disk scheduler), a RAM cache, and possibly an SSD. When the simulator processes a request, a balancer module representing the DataNode logic directs the request to the appropriate disk. The file system for that disk checks the RAM and flash caches; upon a miss, the request is passed to a disk scheduler for re-ordering.

The scheduler switches between files using a round-robin policy (1MB slice). The C-SCAN policy [1] is then used to choose between multiple requests to the same file. The scheduler dispatches requests to a disk module which determines latency. Requests to different files are assumed to be distant, and so require a 10ms seek. Requests to adjacent offsets of the same file, however, are assumed to be adjacent on disk, so blocks are transferred at 100MB/s. Finally, we assume some locality between requests to non-adjacent offsets in the same file; for these, the seek time is $min\{10ms, distance/(100MB/s)\}$.

3.3 Simulation Validity

We now address three validity questions: *does ignoring network latency skew our results? Did we run our simulations long enough? Are simulation results from a single representative machine meaningful?*

First, we explore our assumption about constant network latency by adding random jitter to the timing of requests and observing how important statistics change. Table 1 shows how much error results by changing request issue times by a uniform-random amount. Errors are very small for 1ms jitter (at most 1.3% error). Even with a 10ms jitter, the worst error is 6.6%. Second, in order to verify that we ran the simulations long enough, we measure how the statistics would have been different if we had finished our simulations 2 or 4 days earlier (in-

		jitter ms			finish day		sample
statistic	**baseline**	**1**	**5**	**10**	**-2**	**-4**	**median**
FS reads MB/min	576	0.0	0.0	0.0	-3.4	-0.6	-4.2
FS writes MB/min	447	0.0	0.0	0.0	-7.7	-11.5	-0.1
RAM reads MB/min	287	-0.0	0.0	0.0	-2.6	-2.4	-6.2
RAM writes MB/min	345	0.0	-0.0	-0.0	-3.9	1.1	-2.4
Disk reads MB/min	345	-0.0	0.0	0.0	-3.9	1.1	-2.4
Disk writes MB/min	616	-0.0	1.3	1.9	-5.3	-8.3	-0.1
Net reads MB/min	305	0.0	0.0	0.0	-8.7	-18.4	-2.8
Disk reqs/min	275.1K	0.0	0.0	0.0	-4.6	-4.7	-0.1
(user-read)	65.8K	0.0	-0.0	-0.0	-2.9	-0.8	-4.3
(log)	104.1K	0.0	0.0	0.0	1.6	1.3	-1.0
(flush)	4.5K	0.0	0.0	0.0	1.2	0.4	-1.3
(compact)	100.6K	-0.0	-0.0	-0.0	-12.2	-13.6	-0.1
Disk queue ms	6.17	-0.4	-0.5	-0.0	-3.2	0.6	-1.8
(user-read)	12.3	0.1	-0.8	-1.8	-0.2	2.7	1.7
(log)	2.47	-1.3	-1.1	0.6	-4.9	-6.4	-6.0
(flush)	5.33	0.3	0.0	-0.3	-2.8	-2.6	-1.0
(compact)	6.0	-0.6	0.0	2.0	-3.5	2.5	-6.4
Disk exec ms	0.39	0.1	1.0	2.5	1.0	2.0	-1.4
(user-read)	0.84	-0.1	-0.5	-0.7	-0.0	-0.1	-1.2
(log)	0.26	0.4	3.3	6.6	-2.1	-1.7	0.0
(flush)	0.15	-0.3	0.7	3.2	-1.1	-0.9	-0.8
(compact)	0.24	-0.0	2.1	5.2	4.0	4.8	-0.3

Table 1: **Statistic Sensitivity.** *The first column group shows important statistics and their values for a representative machine. Other columns show how these values would change (as percentages) if measurements were done differently. Low percentages indicate a statistic is robust.*

stead of using the full 8.3 days of traces). The differences are worse than for jitter, but are still usually small, and are at worst 18.4% for network I/O.

Finally, we evaluate whether it is reasonable to pick a single representative instead of running our experiments for all nine machines in our sample. Running all our experiments for a single machine alone takes about 3 days on a 24-core machine with 72GB of RAM, so basing our results on a representative is desirable. The final column of Table 1 compares the difference between statistics for our representative machine and the median of statistics for all nine machines. Differences are quite small and are never greater than 6.4%, so we use the representative for the remainder of our simulations (trace-analysis results, however, will be based on all nine machines).

3.4 Confidentiality

In order to protect user privacy, our traces only contain the sizes of data (*e.g.*, request and file sizes), but never actual data contents. Our tracing code was carefully reviewed by Facebook employees to ensure compliance with Facebook privacy commitments. We also avoid presenting commercially-sensitive statistics, such as would allow estimation of the number of users of the service. While we do an in-depth analysis of the I/O patterns on a sample of machines, we do not disclose how large the sample is as a fraction of all the FM clusters. Much of the architecture we describe is open source.

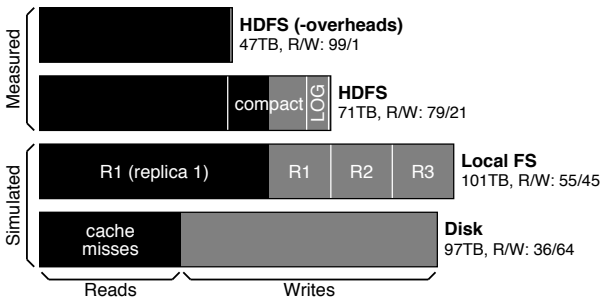


Figure 2: **I/O across layers.** *Black sections represent reads and gray sections represent writes. The top two bars indicate HDFS I/O as measured directly in the traces. The bottom two bars indicate local I/O at the file-system and disk layers as inferred via simulation.*

4 Workload Behavior

We now characterize the FM workload with four questions: *what are the major causes of I/O at each layer of the stack (§4.1)? How much I/O and space is required by different types of data (§4.2)? How large are files, and does file size predict file lifetime (§4.3)?* And *do requests exhibit patterns such as locality or sequentiality (§4.4)?*

4.1 Multilayer Overview

We begin by considering the number of reads and writes at each layer of the stack in Figure 2. At a high level, FM issues `put()` and `get()` requests to HBase. The `put` data accumulates in buffers, which are occasionally flushed to *HFiles* (HDFS files containing sorted key-value pairs and indexing metadata). Thus, `get` requests consult the write buffers as well as the appropriate HFiles in order to retrieve the most up-to-date value for a given key. This core I/O (`put`-flushes and `get`-reads) is shown in the first bar of Figure 2; the 47TB of I/O is 99% reads.

In addition to the core I/O, HBase also does logging (for durability) and compaction (to maintain a read-efficient layout) as shown in the second bar. Writes account for most of these overheads, so the R/W (read/write) ratio decreases to 79/21. Flush data is compressed but log data is not, so logging causes 10x more writes even though the same data is both logged and flushed. Preliminary experiments with log compression [26] have reduced this ratio to 4x. Flushes, which can be compressed in large chunks, have an advantage over logs, which must be written as `puts` arrive. Compaction causes about 17x more writes than flushing does, indicating that a typical piece of data is relocated 17 times. FM stores very large objects (*e.g.*, image attachments) in Haystack [17] for this reason. FM is a very read-heavy HBase workload within Facebook, so it is tuned to compact aggressively. Compaction makes reads faster by merge-sorting many small HFiles into fewer big HFiles,

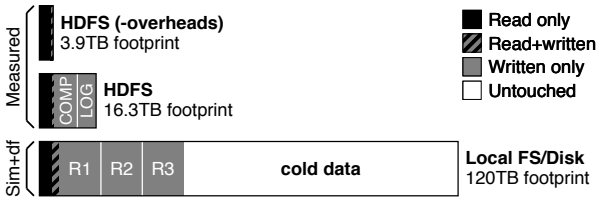


Figure 3: **Data across layers.** *This is the same as Figure 2 but for data instead of I/O. COMP is compaction.*

Family	Description
Actions	Log of user actions and message contents
MessageMeta	Metadata per message (*e.g.*, isRead and subject)
ThreadMeta	Metadata per thread (*e.g.*list of participants)
PrefetchMeta	Privacy settings, contacts, mailbox summary, etc.
Keywords	Word-to-message map for search and typeahead
ThreaderThread	Thread-to-message mapping
ThreadingIdIdx	Map between different types of message IDs
ActionLogIdIdx	Also a message-ID map (like ThreadingIdIdx)

Table 2: **Schema.** *HBase column families are described.*

thus reducing the number of files a `get` must check.

FM tolerates failures by replicating data with HDFS. Thus, writing an HDFS block involves writing three local files and two network transfers. The third bar of Figure 2 shows how this tripling further reduces the R/W ratio to 55/45. Furthermore, OS caching prevents some of these file-system reads from hitting disk. With a 30GB cache, the 56TB of reads at the file-system level cause only 35TB of reads at the disk level, as shown in the fourth bar. Also, very small file-system writes cause 4KB-block disk writes, so writes are increased at the disk level. Because of these factors, writes represent 64% of disk I/O.

Figure 3 gives a similar layered overview, but for data rather than I/O. The first bar shows 3.9TB of HDFS data received some core I/O during tracing (data deleted during tracing is not counted). Nearly all this data was read and a small portion written. The second bar also includes data which was accessed only by non-core I/O; non-core data is several times bigger than core data. The third bar shows how much data is touched at the local level during tracing. This bar also shows *untouched* data; we estimate[2] this by subtracting the amount of data we infer was touched due to HDFS I/O from the disk utilization (measured with `df`). Most of the 120TB of data is very cold; only a third is accessed over the 8-day period.

Conclusion: FM is very read-heavy, but logging, compaction, replication, and caching amplify write I/O, causing writes to dominate disk I/O. We also observe that while the HDFS dataset accessed by core I/O is relatively small, on disk the dataset is very large (120TB) and very cold (two thirds is never touched). Thus, architectures to support this workload should consider its hot/cold nature.

4.2 Data Types

We now study the types of data FM stores. Each user's data is stored in a single HBase row; this prevents the data from being split across different RegionServers. New data for a user is added in new columns within the row. Related columns are grouped into families, which are defined by the FM schema (summarized in Table 2).

[2]the RegionServers in our sample store some data on DataNodes outside our sample (and vice versa), so this is a sample-based estimate rather than a direct correlation of HDFS data to disk data

The *Actions* family is a log built on top of HBase, with different log records stored in different columns; *addMsg* records contain actual message data while other records (*e.g.*, *markAsRead*) record changes to metadata state. Getting the latest state requires reading a number of recent records in the log. To cap this number, a metadata snapshot (a few hundred bytes) is sometimes written to the *MessageMeta* family. Because Facebook chat is built over messages, metadata objects are large relative to many messages (*e.g.*, "hey, whasup?"). Thus, writing a change to *Actions* is generally much cheaper than writing a full metadata object to *MessageMeta*. Other metadata is stored in *ThreadMeta* and *PrefetchMeta* while *Keywords* is a keyword-search index and *ThreaderThread*, *ThreadingIdIdx*, and *ActionLogIdIdx* are other indexes.

Figure 4a shows how much data of each type is accessed at least once during tracing (including later-deleted data); a total (sum of bars) of 26.5TB is accessed. While actual messages (*i.e.*, *Actions*) take significant space, helper data (*e.g.*, metadata, indexes, and logs) takes much more. We also see that little data is both read and written, suggesting that writes should be cached selectively (if at all). Figure 4b reports the I/O done for each type. We observe that some families receive much more I/O per data, *e.g.*, an average data byte of *PrefetchMeta* receives 15 bytes of I/O whereas a byte of *Keywords* receives only 1.1.

Conclusion: FM uses significant space to store messages and does a significant amount of I/O on these messages; however, both space and I/O are dominated by helper data (*i.e.*, metadata, indexes, and logs). Relatively little data is both written and read during tracing; this suggests caching writes is of little value.

4.3 File Size

GFS (the inspiration for HDFS) assumed that "multi-GB files are the common case, and should be handled efficiently" [11]. Other workload studies confirm this, *e.g.*, MapReduce inputs were found to be about 23GB at the 90th percentile (Facebook in 2010) [4]. We now revisit the assumption that HDFS files are large.

Figure 5 shows, for each file type, a distribution of file sizes (about 862 thousand files appear in our traces). Most files are small; for each family, 90% are smaller

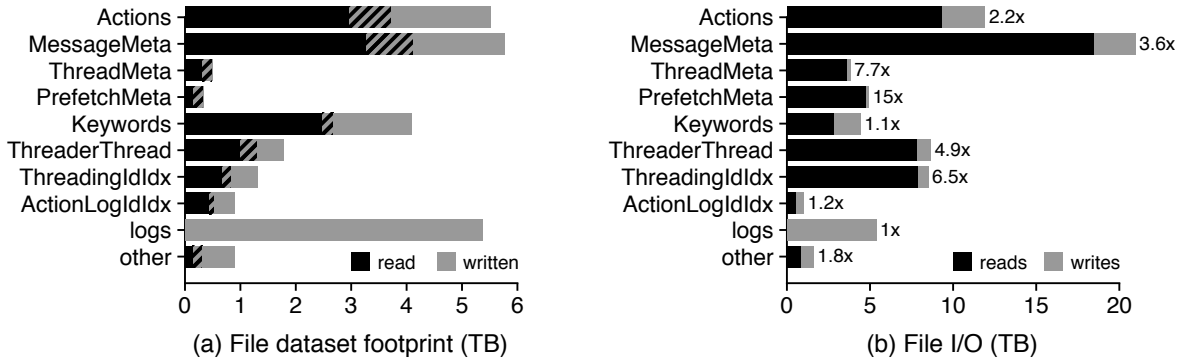


Figure 4: **File types.** *Left: all accessed HDFS file data is broken down by type. Bars further show whether data was read, written, or both. Right: I/O is broken down by file type and read/write. Bar labels indicate the I/O-to-data ratio.*

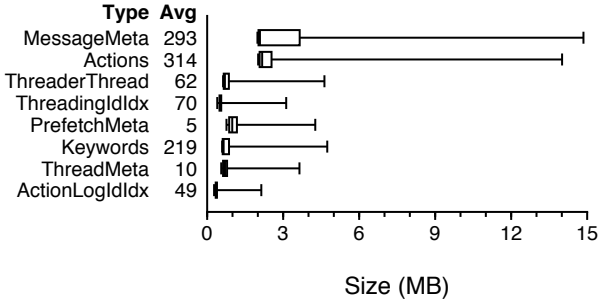


Figure 5: **File-size distribution.** *This shows a box-and-whiskers plot of file sizes. The whiskers indicate the 10th and 90th percentiles. On the left, the type of file and average size is indicated. Log files are not shown, but have an average size of 218MB with extremely little variance.*

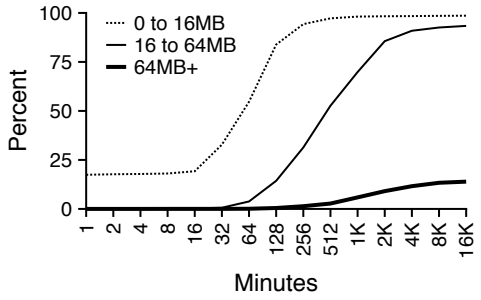


Figure 6: **Size/life correlation.** *Each line is a CDF of lifetime for created files of a particular size. Not all lines reach 100% as some files are not deleted during tracing.*

than 15MB. However, a handful are so large as to skew averages upwards significantly, *e.g.*, the average MessageMeta file is 293MB.

Although most files are very small, compaction should quickly replace these small files with a few large, long-lived files. We divide files created during tracing into small (0 to 16MB), medium (16 to 64MB), and large (64MB+) categories. 94% of files are small, 2% are medium, and 4% are large; however, large files contain 89% of the data. Figure 6 shows the distribution of file lifetimes for each category. 17% of small files are deleted within less than a minute, and very few last more than a few hours; about half of medium files, however, last more than 8 hours. Only 14% of the large files created during tracing were also deleted during tracing.

Conclusion: Traditional HDFS workloads operate on very large files. While most FM data lives in large, long-lived files, most files are small and short-lived. This has metadata-management implications; HDFS manages all file metadata with a single NameNode because the data-to-metadata ratio is assumed to be high. For FM, this assumption does not hold; perhaps distributing HDFS metadata management should be reconsidered.

4.4 I/O Patterns

We explore three relationships between different read requests: temporal locality, spatial locality, and sequentiality. We use a new type of plot, a *locality map*, that describes all three relationships at once. Figure 7 shows a locality map for FM reads. The data shows how often a read was *recently* preceded by a *nearby* read, for various thresholds on "recent" and "nearby". Each line is a hit-ratio curve, with the x-axis indicating how long items are cached. Different lines represent different levels of prefetching, *e.g.*, the 0-line represents no prefetching, whereas the 1MB-line means data 1MB before and 1MB after a read is prefetched.

Line shape describes *temporal locality*, *e.g.*, the 0-line gives a distribution of time intervals between different reads to the same data. Reads are almost never preceded by a prior read to the same data in the past four minutes; however, 26% of reads are preceded within the last 32 minutes. Thus, there is significant temporal locality (*i.e.*, reads are near each other with respect to time), and additional caching should be beneficial. The locality map also shows there is little *sequentiality*. A highly sequen-

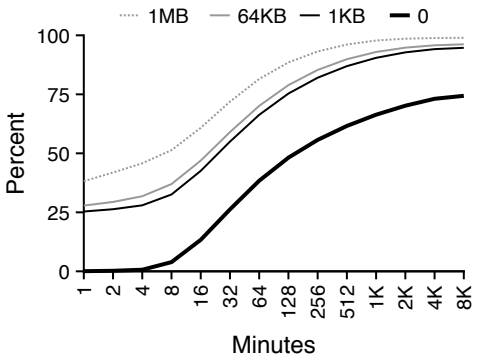


Figure 7: **Reads: locality map.** *This plot shows how often a read was recently preceded by a nearby read, with time-distance represented along the x-axis and offset-distance represented by the four lines.*

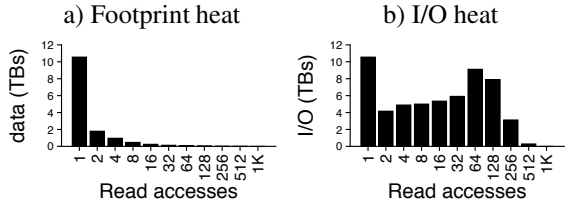


Figure 8: **Read heat.** *In both plots, bars show a distribution across different levels of read heat (i.e., the number of times a byte is read). The left shows a distribution of the dataset (so the bars sum to the dataset size, included deleted data), and the right shows a distribution of I/O to different parts of the dataset (so the bars sum to the total read I/O).*

tial pattern would show that many reads were recently preceded by I/O to nearby offsets; here, however, the 1KB-line shows only 25% of reads were preceded by I/O to very nearby offsets within the last minute. Thus, over 75% of reads are random. The distances between the lines of the locality map describe *spatial locality*. The 1KB-line and 64KB-line are very near each other, indicating that (except for sequential I/O) reads are rarely preceded by other reads to nearby offsets. This indicates very low spatial locality (*i.e.*, reads are far from each other with respect to offset), and additional prefetching is unlikely to be helpful.

To summarize the locality map, the main pattern reads exhibit is temporal locality (there is little sequentiality or spatial locality). High temporal locality implies a significant portion of reads are "repeats" to the same data. We explore this repeated-access pattern further in Figure 8a. The bytes of HDFS file data that are read during tracing are distributed along the x-axis by the number of reads. The figure shows that most data (73.7%) is read only once, but 1.1% of the data is read at least 64 times. Thus, repeated reads are not spread evenly, but are concentrated on a small subset of the data.

Figure 8b shows how many bytes are read for each of the categories of Figure 8a. While 19% of the reads are to bytes which are only read once, most I/O is to data which is accessed many times. Such bias at this level is surprising considering that all HDFS I/O has missed two higher-level caches (an application cache and the HBase cache). Caches are known to lessen I/O to particularly hot data, *e.g.*, a multilayer photo-caching study found caches cause "distributions [to] flatten in a significant way" [15]. The fact that bias remains despite caching suggests the working set may be too large to fit in a small cache; a later section (§5.1) shows this to be the case.

Conclusion: At the HDFS level, FM exhibits relatively little sequentiality, suggesting high-bandwidth, high-latency storage mediums (*e.g.*, disk) are not ideal for serving reads. The workload also shows very little spatial locality, suggesting additional prefetching would not help, possibly because FM already chooses for itself what data to prefetch. However, despite application-level and HBase-level caching, some of the HDFS data is particularly hot; thus, additional caching could help.

5 Tiered Storage: Adding Flash

We now make a case for adding a flash tier to local machines. FM has a very large, mostly cold dataset (§4.1); keeping all this data in flash would be wasteful, costing upwards of $10K/machine[3]. We evaluate the two alternatives: use some flash or no flash. We consider four questions: *how much can we improve performance without flash, by spending more on RAM or disks (§5.1)? What policies utilize a tiered RAM/flash cache best (§5.2)? Is flash better used as a cache to absorb reads or as a buffer to absorb writes (§5.3)?* And ultimately, *is the cost of a flash tier justifiable (§5.4)?*

5.1 Performance without Flash

Can buying faster disks or more disks significantly improve FM performance? Figure 9 presents average disk latency as a function of various disk factors. The first plot shows that for more than 15 disks, adding more disks has quickly diminishing returns. The second shows that higher-bandwidth disks also have relatively little advantage, as anticipated by the highly-random workload observed earlier (§4.4). However, the third plot shows that latency is a major performance factor.

The fact that lower latency helps more than having additional disks suggests the workload has relatively little parallelism, *i.e.*, being able to do a few things quickly is better than being able to do many things at once. Un-

[3] at $0.80/GB, storing 13.3TB (120TB split over 9 machines) in flash would cost $10,895/machine.

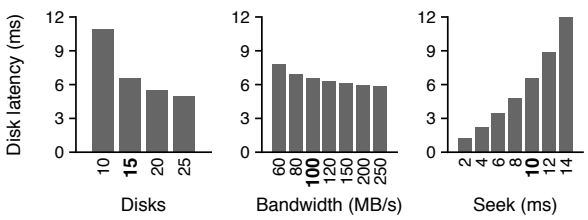


Figure 9: **Disk performance.** *The figure shows the relationship between disk characteristics and the average latency of disk requests. As a default, we use 15 disks with 100MB/s bandwidth and 10ms seek time. Each of the plots varies one of the characteristics, keeping the other two fixed.*

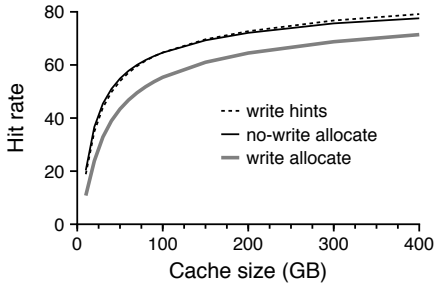


Figure 10: **Cache hit rate.** *The relationship between cache size and hit rate is shown for three policies.*

fortunately, the 2-6ms disks we simulate are unrealistically fast, having no commercial equivalent. Thus, although significant disk capacity is needed to store the large, mostly cold data, reads are better served by a low-latency medium (*e.g.*, RAM or flash).

Thus, we ask, *can the hot data fit comfortably in a pure-RAM cache?* We measure hit rate for cache sizes in the 10-400GB range. We also try three different LRU policies: *write allocate*, *no-write allocate*, and *write hints*. All three are write-through caches, but differ regarding whether written data is cached. Write allocate adds all write data, no-write allocate adds no write data, and the hint-based policy takes suggestions from HBase and HDFS. In particular, a written file is only cached if (a) the local file is a primary replica of the HDFS block, and (b) the file is either flush output (as opposed to compaction output) or is likely to be compacted soon.

Figure 10 shows, for each policy, that the hit rate increases significantly as the cache size increases up until about 200GB, where it starts to level off (but not flatten); this indicates the working set is very large. Earlier (§4.2), we found little overlap between writes and reads and concluded that written data should be cached selectively if at all. Figure 10 confirms: caching all writes is the worst policy. Up until about 100GB, "no-write allocate" and "write hints" perform about equally well. Beyond 100GB, hints help, but only slightly. We use no-write allocate throughout the remainder of the paper because it is simple and provides decent performance.

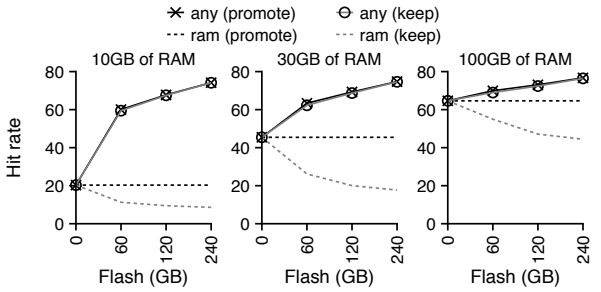


Figure 11: **Tiered hit rates.** *Overall hit rate (any) is shown by the solid lines for the promote and keep policies. The results are shown for varying amounts of RAM (different plots) and varying amounts of flash (x-axis). RAM hit rates are indicated by the dashed lines.*

Conclusion: The FM workload exhibits relatively little sequentiality or parallelism, so adding more disks or higher-bandwidth disks is of limited utility. Fortunately, the same data is often repeatedly read (§4.4), so a very large cache (*i.e.*, a few hundred GBs in size) can service nearly 80% of the reads. The usefulness of a very large cache suggests that storing at least some of the hot data in flash may be most cost effective. We evaluate the cost/performance tradeoff between pure-RAM and hybrid caches in a later section (§5.4).

5.2 Flash as Cache

In this section, we use flash as a second caching tier beneath RAM. Both tiers independently are LRU. Initial inserts are to RAM, and RAM evictions are inserted into flash. We evaluate exclusive cache policies. Thus, upon a flash hit, we have two options: the *promote policy* (PP) repromotes the item to the RAM cache, but the *keep policy* (KP) keeps the item at the flash level. PP gives the combined cache LRU behavior. The idea behind KP is to limit SSD wear by avoiding repeated promotions and evictions of items between RAM and flash.

Figure 11 shows the hit rates for twelve flash/RAM mixes. For example, the middle plot shows what the hit rate is when there is 30GB of RAM: without any flash, 45% of reads hit the cache, but with 60GB of flash, about 63% of reads hit in either RAM or flash (regardless of policy). The plots show that across all amounts of RAM and flash, the number of reads that hit in "any" cache differs very little between policies. However, PP causes significantly more of these hits to go to RAM; thus, PP will be faster because RAM hits are faster than flash hits.

We now test our hypothesis that, in trade for decreasing RAM hits, KP improves flash lifetime. We compute lifetime by measuring flash writes, assuming the FTL provides even wear leveling, and assuming the SSD supports 10K program/erase cycles. Figure 12 reports flash lifetime as the amount of flash varies along the x-axis.

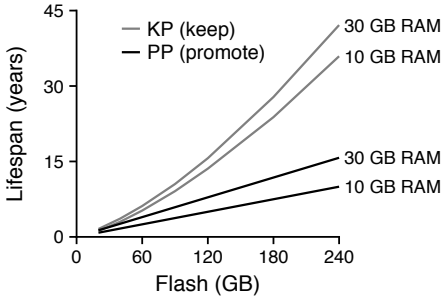


Figure 12: **Flash lifetime.** *The relationship between flash size and flash lifetime is shown for both the keep policy (gray lines) and promote policy (black lines). There are two lines for each policy (10 or 30GB RAM).*

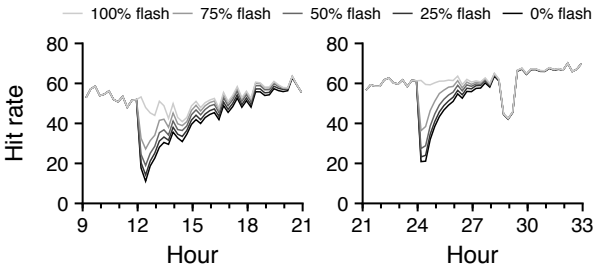


Figure 13: **Crash simulations.** *The plots show two examples of how crashing at different times affects different 100GB tiered caches, some of which are pure flash, pure RAM, or a mix. Hit rates are unaffected when crashing with 100% flash.*

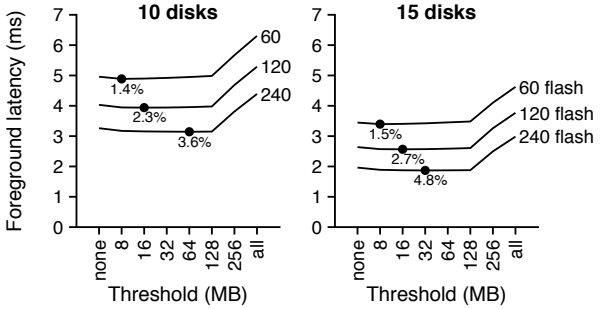


Figure 14: **Flash Buffer.** *We measure how different file-buffering policies impact foreground requests with two plots (for 10 or 15 disks) and three lines (60, 120, or 240GB of flash). Different points on the x-axis represent different policies. The optimum point on each line is marked, showing improvement relative to the latency when no buffering is done.*

The figure shows that having more RAM slightly improves flash lifetime. This is because flash writes occur upon RAM evictions, and evictions will be less frequent with ample RAM. Also, as expected, KP often doubles or triples flash lifetime, *e.g.*, with 10GB of RAM and 60GB of flash, using KP instead of PP increases lifetime from 2.5 to 5.2 years. The figure also shows that flash lifetime increases with the amount of flash. For PP, the relationship is perfectly linear. The number of flash writes equals the number of RAM evictions, which is independent of flash size; thus, if there is twice as much flash, each block of flash will receive exactly half as much wear. For KP, however, the flash lifetime increases superlinearly with size; with 10GB of RAM and 20GB of flash, the years-to-GB ratio is 0.06, but with 240GB of flash, the ratio is 0.15. The relationship is superlinear because additional flash absorbs more reads, causing fewer RAM inserts, causing fewer RAM evictions, and ultimately causing fewer flash writes. Thus, doubling the flash size decreases total flash writes in addition to spreading the writes over twice as many blocks.

Flash caches have an additional advantage: crashes do not cause cache contents to be lost. We quantify this benefit by simulating four crashes at different times and measuring changes to hit rate. Figure 13 shows the results of two of these crashes for 100GB caches with different flash-to-RAM ratios (using PP). Even though the hottest data will be in RAM, keeping some data in flash significantly improves the hit rate after a crash. The examples also show that it can take 4-6 hours to fully recover from a crash. We quantify the total recovery cost in terms of additional disk reads (not shown). Whereas crashing with a pure-RAM cache on average causes 26GB of additional disk I/O, crashing costs only 10GB for a hybrid cache which is 75% flash.

Conclusion: Adding flash to RAM can greatly improve the caching hit rate; furthermore (due to persistence) a hybrid flash/RAM cache can eliminate half of the extra disk reads that usually occur after a crash. However, using flash raises concerns about wear. Shuffling data between flash and RAM to keep the hottest data in RAM improves performance but can easily decrease SSD lifetime by a factor of 2x relative to a wear-aware policy. Fortunately, larger SSDs tend to have long lifetimes for FM, so wear may be a small concern (*e.g.*, 120GB+ SSDs last over 5 years regardless of policy).

5.3 Flash as Buffer

Another advantage of flash is that (due to persistence) it has the potential to reduce disk writes as well as reads. We saw earlier (§4.3) that files tend to be either small and short-lived or big and long-lived, so one strategy would be to store small files in flash and big files on disk.

HDFS writes are considered durable once the data is in memory on every DataNode (but not necessarily on disk), so buffering in flash would not actually improve HDFS write performance. However, decreasing disk writes by buffering the output of *background activities* (*e.g.*, flushes and compaction) indirectly improves *foreground* performance. Foreground activity includes any local requests which could block an HBase request (*e.g.*,

HW	Cost	Failure rate	Performance
HDD	$100/disk	4% AFR [9]	10ms/seek, 100MB/s
RAM	$5.0/GB	4% AFR (8GB)	0 latency
Flash	$0.8/GB	10K P/E cycles	0.5ms latency

Table 3: **Cost Model.** *Our assumptions about hardware costs, failure rates, and performance are presented. For disk and RAM, we state an AFR (annual failure rate), assuming uniform-random failure each year. For flash, we base replacement on wear and state program/erase cycles.*

a `get`). Reducing background I/O means foreground reads will face less competition for disk time. Thus, we measure how buffering files written by background activities affects foreground latencies.

Of course, using flash as a write buffer has a cost, namely less space for caching hot data. We evaluate this tradeoff by measuring performance when using flash to buffer only files which are beneath a certain size. Figure 14 shows how latency corresponds to the policy. At the left of the x-axis, writes are never buffered in flash, and at the right of the x-axis, all writes are buffered. Other x-values represent thresholds; only files smaller than the threshold are buffered. The plots show that buffering all or most of the files results in very poor performance. Below 128MB, though, the choice of how much to buffer makes little difference. The best gain is just a 4.8% reduction in average latency relative to performance when no writes are buffered.

Conclusion: Using flash to buffer all writes results in much worse performance than using flash only as a cache. If flash is used for both caching and buffering, and if policies are tuned to only buffer files of the right size, then performance can be slightly improved. We conclude that these small gains are probably not worth the added complexity, so flash should be for caching only.

5.4 Is Flash worth the Money?

Adding flash to a system can, if used properly, only improve performance, so the interesting question is, given that we want to buy performance with money, *should we buy flash, or something else?* We approach this question by making assumptions about how fast and expensive different storage mediums are, as summarized in Table 3. We also state assumptions about component failure rates, allowing us to estimate operating expenditure.

We evaluate 36 systems, with three levels of RAM (10GB, 30GB, or 100GB), four levels of flash (none, 60GB, 120GB, or 240GB), and three levels of disk (10, 15, or 20 disks). Flash and RAM are used as a hybrid cache with the promote policy (§5.2). For each system, we compute the capex (capital expenditure) to initially purchase the hardware and determine via simulation the foreground latencies (defined in §5.3). Figure 15 shows the cost/performance of each system. 11 of the systems

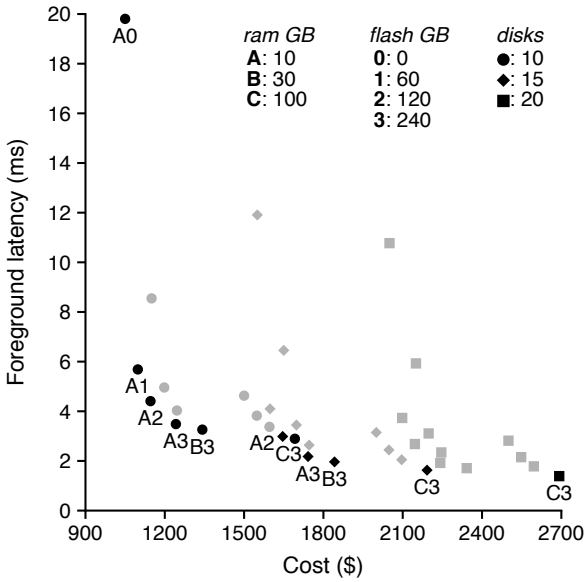


Figure 15: **Capex/latency tradeoff.** *We present the cost and performance of 36 systems, representing every combination of three RAM levels, four flash levels, and three disk levels. Combinations which present unique tradeoffs are black and labeled; unjustifiable systems are gray and unlabeled.*

(31%) are highlighted; these are the only systems that one could justify buying. Each of the other 25 systems is both slower and more expensive than one of these 11 justifiable systems. Over half of the justifiable systems have maximum flash. It is worth noting that the systems with less flash are justified by low cost, not good performance. With one exception (15-disk A2), all systems with less than the maximum flash have the minimum number of disks and RAM. We observe that flash can greatly improve performance at very little cost. For example, A1 has a 60GB SSD but is otherwise the same as A0. With 10 disks, A1 costs only 4.5% more but is 3.5x faster. We conclude that if performance is to be bought, then (within the space we explore) flash should be purchased first.

We also consider expected opex (operating expenditure) for replacing hardware as it fails, and find that replacing hardware is relatively inexpensive compared to capex (not shown). Of the 36 systems, opex is at most $90/year/machine (for the 20-disk C3 system). Furthermore, opex is never more than 5% of capex. For each of the justifiable flash-based systems shown in Figure 15, we also do simulations using KP for flash hits. KP decreased opex by 4-23% for all flash machines while increasing latencies by 2-11%. However, because opex is low in general, the savings are at most $14/year/machine.

Conclusion: Not only does adding a flash tier to the FM stack greatly improve performance, but it is the most cost-effective way of improving performance. In some cases, adding a small SSD can triple performance while only increasing monetary costs by 5%.

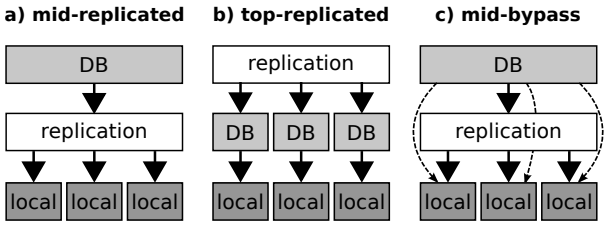


Figure 16: **Layered architectures.** *The HBase architecture (mid-replicated) is shown, as well as two alternatives. Top-replication reduces network I/O by co-locating database computation with database data. The mid-bypass architecture is similar to mid-replication, but provides a mechanism for bypassing the replication layer for efficiency.*

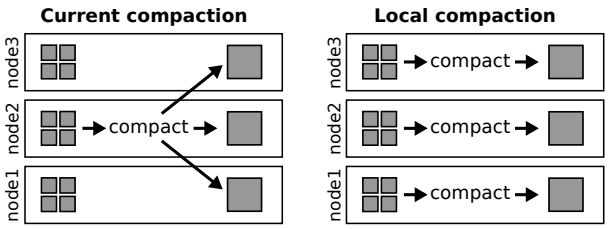


Figure 17: **Local-compaction architecture.** *The HBase architecture (left) shows how compaction currently creates a data flow with significant network I/O, represented by the two lines crossing machine boundaries. An alternative (right) shows how local reads could replace network I/O*

6 Layering: Pitfalls and Solutions

The FM stack, like most storage, is a composition of other systems and subsystems. Some composition is horizontal; for example, FM stores small data in HBase and large data in Haystack (§4.1). In this section, we focus instead on the vertical composition of layers, a pattern commonly used to manage and reduce software complexity. We discuss different ways to organize storage layers (§6.1), how to reduce network I/O by bypassing the replication layer (§6.2), and how to reduce the randomness of disk I/O by adding special HDFS support for HBase logging (§6.3).

6.1 Layering Background

Three important layers are the *local layer* (*e.g.*, disks, local file systems, and a DataNode), the *replication layer* (*e.g.*, HDFS), and the *database layer* (*e.g.*, HBase). FM composes these in a *mid-replicated* pattern (Figure 16a), with the database at the top of the stack and the local stores at the bottom. The merit of this architecture is simplicity. The database can be built with the assumption that underlying storage, because it is replicated, will be available and never lose data. The replication layer is also relatively simple, as it deals with data in its simplest form (*i.e.*, large blocks of opaque data). Unfortunately, mid-replicated architectures separate computation from data. Computation (*e.g.*, database operations such as compaction) can only be co-resident with at most one replica, so all writes involve network transfers.

Top-replication (Figure 16b) is an alternative approach used by the Salus storage system [29]. Salus supports the standard HBase API, but its top-replicated approach provides additional robustness and performance advantages. Salus protects against memory corruption and certain bugs in the database layer by replicating database computation as well as the data itself. Doing replication above the database level also reduces network I/O. If the database wants to reorganize data on disk (*e.g.*, via compaction), each database replica can do so on its local copy. Unfortunately, top-replicated storage is complex. The database layer must handle underlying failures as well as cooperate with other databases; in Salus, this is accomplished with a pipelined-commit protocol and Merkle trees for maintaining consistency.

Mid-bypass (Figure 16c) is a third option proposed by Zaharia *et al.* [30]. This approach (like mid-replication), places the replication layer between the database and the local store, but in order to improve performance, an *RDD* (Resilient Distributed Dataset) API lets the database bypass the replication layer. Network I/O is avoided by shipping computation directly to the data. HBase compaction could be built upon two RDD transformations, *join* and *sort*, and network I/O could thus be avoided.

6.2 Local Compaction

We simulate the mid-bypass approach, with compaction operations shipped directly to all the replicas of compaction inputs. Figure 17 shows how local compaction differs from traditional compaction; network I/O is traded for local I/O, to be served by local caches or disks.

Figure 18 shows the result: a 62% reduction in network reads from 3.5TB to 1.3TB. The figure also shows disk reads, with and without local compaction, and with either write allocate (wa) or no-write allocate (nwa) caching policies (§5.1). We observe disk I/O increases sightly more than network I/O decreases. For example, with a 100GB cache, network I/O is decreased by 2.2GB but disk reads are increased by 2.6GB for no-write allocate. This is unsurprising: HBase uses secondary replicas for fault tolerance rather than for reads, so secondary replicas are written once (by a flush or compaction) and read at most once (by compaction). Thus, local-compaction reads tend to (a) be misses and (b) pollute the cache with data that will not be read again. We see that write allocate still underperforms no-write allo-

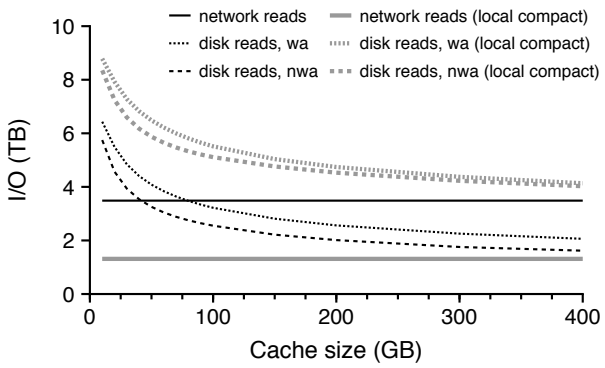


Figure 18: **Local-compaction results.** *The thick gray lines represent HBase with local compaction, and the thin black lines represent HBase currently. The solid lines represent network reads, and the dashed lines represent disk reads; long-dash represents the no-write allocate cache policy and short-dash represents write allocate.*

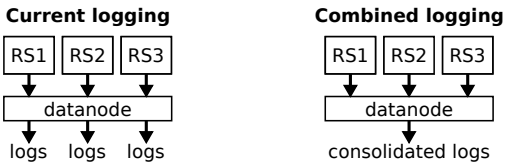


Figure 19: **Combined-logging architecture.** *Currently (left), the average DataNode will receive logs from three HBase RegionServers, and these logs will be written to different locations. An alternative approach (right) would be for HDFS to provide a special logging API which allows all the logs to be combined so that disk seeks are reduced.*

cate (§5.1). However, write allocate is now somewhat more competitive for large cache sizes because it is able to serve some of the data read by local compaction.

Conclusion: Doing local compaction by bypassing the replication layer turns over half the network I/O into disk reads. This is a good tradeoff as network I/O is generally more expensive than sequential disk I/O.

6.3 Combined Logging

We now consider the interaction between replication and HBase logging. Figure 19 shows how (currently) a typical DataNode will receive log writes from three RegionServers (because each RegionServer replicates its logs to three DataNodes). These logs are currently written to three different local files, causing seeks. Such seeking could be reduced if HDFS were to expose a special logging feature that merges all logical logs into a single physical log on a dedicated disk as illustrated.

We simulate combined logging and measure performance for requests which go to disk; we consider latencies for foreground reads (defined in §5.1), compaction, and logging. Figure 20 reports the results for varying numbers of disks. The latency of log writes decreases dramatically with combined logging; for example, with 15 disks, the latency is decreased by a factor of six. Compaction requests also experience modest gains due to less competition for disk seeks. Currently, neither logging nor compaction block the end user, so we also consider the performance of foreground reads. For this metric, the gains are small, *e.g.*, latency only decreases by 3.4% with 15 disks. With just 10 disks, dedicating one disk to logging slightly hurts user reads.

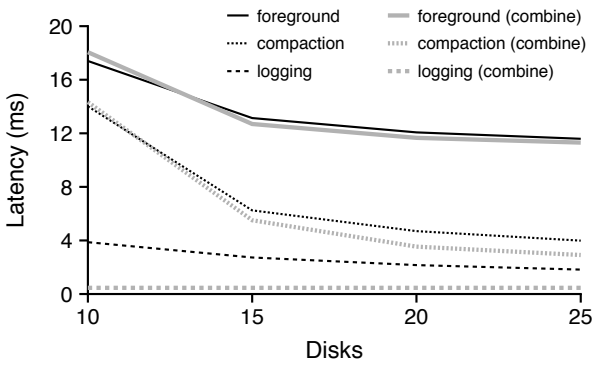


Figure 20: **Combined logging results.** *Disk latencies for various activities are shown, with (gray) and without (black) combined logging.*

Conclusion: Merging multiple HBase logs on a dedicated disk reduces logging latencies by a factor of 6. However, `put` requests do not currently block until data is flushed to disks, and the performance impact on foreground reads is negligible. Thus, the additional complexity of combined logging is likely not worthwhile given the current durability guarantees. However, combined logging could enable HBase, at little performance cost, to give the additional guarantee that data is on disk before a `put` returns. Providing such a guarantee would make logging a foreground activity.

7 Related Work

In this work, we compare the I/O patterns of FM to prior GFS and HDFS workloads. Chen *et al.*[4] provides broad characterizations of a wide variety of MapReduce workloads, making some of the comparisons possible. The MapReduce study is *broad*, analyzing traces of coarse-grained events (*e.g.*, file opens) from over 5000 machines across seven clusters. By contrast, our study is *deep*, analyzing traces of fine-grained events (*e.g.*, reads to a byte) for just nine machines.

Detailed trace analysis has also been done in many non-HDFS contexts, such as the work by Baker *et al.* [2]

in a BSD environment and by Harter *et al.* [13] for Apple desktop applications. Other studies include the work done by Ousterhout *et al.* [21] and Vogels *et al.* [28].

A recent photo-caching study by Huang *et al.* [15] focuses, much like our work, on I/O patterns across multiple layers of the stack. The photo-caching study correlated I/O across levels by tracing at each layer, whereas our approach was to trace at a single layer and infer I/O at each underlying layer via simulation. There is a tradeoff between these two methodologies: tracing multiple levels avoids potential inaccuracies due to simulator oversimplifications, but the simulation approach enables greater experimentation with alternative architectures beneath the traced layer.

Our methodology of trace-driven analysis and simulation is inspired by Kaushik *et al.* [16], a study of Hadoop traces from Yahoo! Both the Yahoo! study and our work involved collecting traces, doing analysis to discover potential improvements, and running simulations to evaluate those improvements.

We are not the first to suggest the methods we evaluated for better HDFS integration (§6); our contribution is to quantify how useful these techniques are for the FM workload. The observation that doing compaction above the replication layer wastes network bandwidth has been made by Wang *et al.* [29], and the approach of local compaction is a specific application of the more general techniques described by Zaharia *et al.* [30]. Combined logging is also commonly used by administrators of traditional databases [8, 22].

8 Conclusions

We have presented a detailed multilayer study of storage I/O for Facebook Messages. Our combined approach of analysis and simulation allowed us to identify potentially useful changes and then evaluate those changes. We have four major conclusions.

First, the special handling received by writes make them surprisingly expensive. At the HDFS level, the read/write ratio is 99/1, excluding HBase compaction and logging overheads. At the disk level, the ratio is write-dominated at 36/64. Logging, compaction, replication, and caching all combine to produce this write blowup. Thus, optimizing writes is very important even for especially read-heavy workloads such as FM.

Second, the GFS-style architecture is based on workload assumptions such as "high sustained bandwidth is more important than low latency" [11]. For FM, many of these assumptions no longer hold. For example, we demonstrate (§5.1) just the opposite is true for FM: because I/O is highly random, bandwidth matters little, but latency is crucial. Similarly, files were assumed to be very large, in the hundreds or thousands of megabytes. This traditional workload implies a high data-to-metadata ratio, justifying the one-NameNode design of GFS and HDFS. By contrast, FM is dominated by small files; perhaps the single-NameNode design should be revisited.

Third, FM storage is built upon layers of independent subsystems. This architecture has the benefit of simplicity; for example, because HBase stores data in a replicated store, it can focus on high-level database logic instead of dealing with dying disks and other types of failure. Layering is also known to improve reliability, *e.g.*, Dijkstra found layering "proved to be vital for the verification and logical soundness" of an OS [7]. Unfortunately, we find that the benefits of simple layering are not free. In particular, we showed (§6) that building a database over a replication layer causes additional network I/O and increases workload randomness at the disk layer. Fortunately, simple mechanisms for sometimes bypassing replication can reduce layering costs.

Fourth, the cost of flash has fallen greatly, prompting Gray's proclamation that "tape is dead, disk is tape, flash is disk" [12]. To the contrary, we find that for FM, flash is not a suitable replacement for disk. In particular, the cold data is too large to fit well in flash (§4.1) and the hot data is too large to fit well in RAM (§5.1). However, our evaluations show that architectures with a small flash tier have a positive cost/performance tradeoff compared to systems built on disk and RAM alone.

In this work, we take a unique view of Facebook Messages, not as a single system, but as a complex composition of systems and subsystems, residing side-by-side and layered one upon another. We believe this perspective is key to deeply understanding modern storage systems. Such understanding, we hope, will help us better integrate layers, thereby maintaining simplicity while achieving new levels of performance.

9 Acknowledgements

We thank the anonymous reviewers and Andrew Warfield (our shepherd) for their tremendous feedback, as well as members of our research group for their thoughts and comments on this work at various stages. We also thank Pritam Damania, Adela Maznikar, and Rishit Shroff for their help in collecting HDFS traces.

This material was supported by funding from NSF grants CNS-1319405 and CNS-1218405 as well as generous donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Sony, and VMware. Tyler Harter is supported by the NSF Fellowship and Facebook Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2014.

[2] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, Seattle, Washington, November 2006.

[4] Chen, Yanpei and Alspaugh, Sara and Katz, Randy. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proc. VLDB Endow.*, August 2012.

[5] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 87–100, San Francisco, California, December 2005.

[6] Dhruba Borthakur and Kannan Muthukkaruppan and Karthik Ranganathan and Samuel Rash and Joydeep Sen Sarma and Nicolas Spiegelberg and Dmytro Molkov and Rodrigo Schmidt and Jonathan Gray and Hairong Kuang and Aravind Menon and Amitanand Aiyer. Apache Hadoop Goes Realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*, Athens, Greece, June 2011.

[7] E. W. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[8] IBM Product Documentation. Notes/domino best practices: Transaction logging. http://www-01.ibm.com/support/docview.wss?uid=swg27009309, 2013.

[9] Ford, Daniel and Labelle, François and Popovici, Florentina I. and Stokely, Murray and Truong, Van-Anh and Barroso, Luiz and Grimes, Carrie and Quinlan, Sean. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[10] Gregory R. Ganger. Blurring the Line Between Oses and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, New York, October 2003.

[12] Jim Gray. Tape is Dead. Disk is Tape. Flash is Disk, RAM Locality is King, 2006.

[13] Tyler Harter, Charlotte Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Sys-tems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[14] Joseph L. Hellerstein. Google cluster data. Google research blog, January 2010. Posted at http://googleresearch.blogspot.com/2010/01/google-cluster-data.html.

[15] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An Analysis of Facebook Photo Caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 167–181, Farmington, Pennsylvania, November 2013.

[16] Rini T. Kaushik and Milind A Bhandarkar. GreenHDFS: Towards an Energy-Conserving, Storage-Efficient, Hybrid Hadoop Compute Cluster. In *The 2010 Workshop on Power Aware Computing and Systems (HotPower '10)*, Vancouver, Canada, October 2010.

[17] Niall Kennedy. Facebook's Photo Storage Rewrite. http://www.niallkennedy.com/blog/2009/04/facebook-haystack.html, April 2009.

[18] Butler W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP '83)*, pages 33–48, Bretton Woods, New Hampshire, October 1983.

[19] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, Massachusetts, October 1996.

[20] Kannan Muthukkaruppan. Storage Infrastructure Behind Facebook Messages. In *Proceedings of International Workshop on High Performance Transaction Systems (HPTS '11)*, Pacific Grove, California, October 2011.

[21] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP '85)*, pages 15–24, Orcas Island, Washington, December 1985.

[22] Matt Perdeck. Speeding up database access. http://www.codeproject.com/Articles/296523/Speeding-up-database-access-part-8-Fixing-memory-d, 2011.

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

[24] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Incline Village, Nevada, May 2010.

[25] Jason Sobel. Needle in a haystack: Efficient storage of billions of photos. http://www.flowgram.com/p/2qi3k8eicrfgkv, June 2008.

[26] Nicolas Spiegelberg. Allow record compression for hlogs. https://issues.apache.org/jira/browse/HBASE-8155, 2013.

[27] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237, Saint-Malo, France, October 1997.

[28] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 93–109, Kiawah Island Resort, South Carolina, December 1999.

[29] Yang Wang and Manos Kapritsos and Zuocheng Ren and Prince Mahajan and Jeevitha Kirubanandam and Lorenzo Alvisi and Mike Dahlin. Robustness in the Salus Scalable Block Store. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation*, Lombard, Illinois, April 2013.

[30] Zaharia, Matei and Chowdhury, Mosharaf and Das, Tathagata and Dave, Ankur and Ma, Justin and McCauley, Murphy and Franklin, Michael J. and Shenker, Scott and Stoica, Ion. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, San Jose, California, April 2010.

Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces

Yang Liu*, Raghul Gunasekaran†, Xiaosong Ma◇,* and Sudharshan S. Vazhkudai†

*North Carolina State University, yliu43@ncsu.edu
◇Qatar Computing Research Institute, xma@qf.org.qa
†Oak Ridge National Laboratory, {gunasekaranr, vazhkudaiss}@ornl.gov

Abstract

Competing workloads on a shared storage system cause I/O resource contention and application performance vagaries. This problem is already evident in today's HPC storage systems and is likely to become acute at exascale. We need more interaction between application I/O requirements and system software tools to help alleviate the I/O bottleneck, moving towards I/O-aware job scheduling. However, this requires rich techniques to capture application I/O characteristics, which remain evasive in production systems.

Traditionally, I/O characteristics have been obtained using client-side tracing tools, with drawbacks such as non-trivial instrumentation/development costs, large trace traffic, and inconsistent adoption. We present a novel approach, I/O Signature Identifier (IOSI), to characterize the I/O behavior of data-intensive applications. IOSI extracts signatures from *noisy, zero-overhead* server-side I/O throughput logs that are already collected on today's supercomputers, without interfering with the compiling/execution of applications. We evaluated IOSI using the Spider storage system at Oak Ridge National Laboratory, the S3D turbulence application (running on 18,000 Titan nodes), and benchmark-based pseudo-applications. Through our experiments we confirmed that IOSI effectively extracts an application's I/O signature despite significant server-side noise. Compared to client-side tracing tools, IOSI is transparent, interface-agnostic, and incurs no overhead. Compared to alternative data alignment techniques (e.g., dynamic time warping), it offers higher signature accuracy and shorter processing time.

1 Introduction

High-performance computing (HPC) systems cater to a diverse mix of scientific applications that run concurrently. While individual compute nodes are usually dedicated to a single parallel job at a time, the interconnection network and the storage subsystem are often shared among jobs. Network topology-aware job placement attempts to allocate larger groups of contiguous compute nodes to each application, in order to provide more stable message-passing performance for inter-process communication. I/O resource contention, however, continues to cause significant performance vagaries in applications [16, 59]. For example, the indispensable task of checkpointing is becoming increasingly cumbersome. The CHIMERA [13] astrophysics application produces 160TB of data per checkpoint, taking around an hour to write [36] on Oak Ridge National Laboratory's Titan [3] (currently the world's No. 2 supercomputer [58]).

This already bottleneck-prone I/O operation is further stymied by resource contention due to concurrent applications, as there is no I/O-aware scheduling or inter-job coordination on supercomputers. As hard disks remain the dominant parallel file system storage media, I/O contention leads to excessive seeks, significantly degrading the overall I/O throughput.

This problem is expected to exacerbate on future extreme-scale machines (hundreds of petaflops). Future systems demand a sophisticated interplay between application requirements and system software tools that is lacking in today's systems. The aforementioned I/O performance variance problem makes an excellent candidate for such synergistic efforts. For example, knowledge of application-specific I/O behavior potentially allows a scheduler to stagger I/O-intensive jobs, improving *both* the stability of individual applications' I/O performance and the overall resource utilization. However, I/O-aware scheduling requires detailed information on application I/O characteristics. In this paper, we explore the techniques needed to capture such information in an automatic and non-intrusive way.

Cross-layer communication regarding I/O characteristics, requirements or system status has remained a challenge. Traditionally, these I/O characteristics have been captured using client-side tracing tools [5, 7], running on the compute nodes. Unfortunately, the information provided by client-side tracing is not enough for inter-job coordination due to the following reasons.

*Part of this work was conducted at North Carolina State University.

First, client-side tracing requires the use of I/O tracing libraries and/or application code instrumentation, often requiring non-trivial development/porting effort. Second, such tracing effort is entirely elective, rendering any job coordination ineffective when only a small portion of jobs perform (and release) I/O characteristics. Third, many users who do enable I/O tracing choose to turn it on for shorter debug runs and off for production runs, due to the considerable performance overhead (typically between 2% and 8% [44]). Fourth, different jobs may use different tracing tools, generating traces with different formats and content, requiring tremendous knowledge and integration. Finally, unique to I/O performance analysis, detailed tracing often generates large trace files themselves, creating additional I/O activities that perturb the file system and distort the original application I/O behavior. Even with reduced compute overhead and minimal information collection, in a system like Titan, collecting traces for individual applications from over 18,000 compute nodes will significantly stress the interconnect and I/O subsystems. These factors limit the usage of client-side tracing tools for development purposes [26, 37], as opposed to routine adoption in production runs or for daily operations.

Similarly, very limited server-side I/O tracing can be performed on large-scale systems, where the bookkeeping overhead may bring even more visible performance degradations. Centers usually deploy only rudimentary monitoring schemes that collect *aggregate* workload information regarding combined I/O traffic from concurrently running applications.

In this paper, we present IOSI (I/O Signature Identifier), a novel approach to characterizing per-application I/O behavior from *noisy, zero-overhead server-side I/O throughput logs*, collected without interfering with the target application's execution. IOSI leverages the existing infrastructure in HPC centers for periodically logging high-level, server-side I/O throughput. E.g., the throughput on the I/O controllers of Titan's Spider file system [48] is recorded once every 2 seconds. Collecting this information has no performance impact on the compute nodes, does not require any user effort, and has minimal overhead on the storage servers. Further, the log collection traffic flows through the storage servers' Ethernet management network, without interfering with the application I/O. Hence, we refer to our log collection as zero-overhead.

Figure 1 shows sample server-side log data from a typical day on Spider. The logs are composite data, reflecting multiple applications' I/O workload. Each instance of an application's execution will be recorded in the server-side I/O throughput log (referred to as a *sample* in the rest of this paper). Often, an I/O-intensive application's samples show certain repeated I/O patterns,

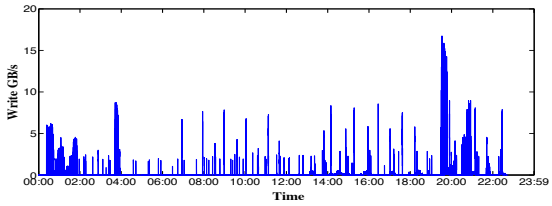


Figure 1: Average server-side, write throughput on Titan's Spider storage (a day in November 2011).

as can be seen from Figure 1. Therefore, the main idea of this work is to *collect and correlate multiple samples, filter out the "background noise", and finally identify the target application's native I/O traffic common across them*. Here, "background noise" refers to the traffic generated by other concurrent applications and system maintenance tasks. Note that IOSI is not intended to record fine-grained, per-application I/O operations. Instead, it derives an estimate of their bandwidth needs along the execution timeline to support future I/O-aware smart decision systems.

Contributions: (1) We propose to extract per-application I/O workload information from existing, zero-overhead, server-side I/O measurements and job scheduling history. Further, we obtain such knowledge of a target application without interfering with its computation/communication, or requiring developers/users' intervention. (2) We have implemented a suite of techniques to identify an application's I/O signature, from noisy server-side throughput measurements. These include i) *data preprocessing*, ii) *per-sample wavelet transform (WT) for isolating I/O bursts*, and iii) *cross-sample I/O burst identification*. (3) We evaluated IOSI with real-world server-side I/O throughput logs from the Spider storage system at the Oak Ridge Leadership Computing Facility (OLCF). Our experiments used several pseudo-applications, constructed with the expressive IOR benchmarking tool [1], and S3D [56], a large-scale turbulent combustion code. Our results show that IOSI effectively extracts an application's I/O signature despite significant server-side noise.

2 Background

We first describe the features of typical I/O-intensive parallel applications and the existing server-side monitoring infrastructure on supercomputers – two enabling trends for IOSI. Next, we define the per-application I/O signature extraction problem.

2.1 I/O Patterns of Parallel Applications

The majority of applications on today's supercomputers are parallel numerical simulations that perform iterative, timestep-based computations. These applications are write-heavy, periodically writing out intermediate re-

sults and checkpoints for analysis and resilience, respectively. For instance, applications compute for a fixed number of timesteps and then perform I/O, repeating this sequence multiple times. This process creates regular, predictable I/O patterns, as noted by many existing studies [25, 49, 61]. More specifically, parallel applications' dominant I/O behavior exhibits several distinct features that enable I/O signature extraction:

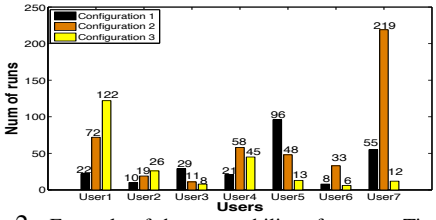


Figure 2: Example of the repeatability of runs on Titan, showing the number of runs using identical job configurations for seven users issuing the largest jobs, between July and September 2013.

Burstiness: Scientific applications have distinct compute and I/O phases. Most applications are designed to perform I/O in short bursts [61], as seen in Figure 1.

Periodicity: Most I/O-intensive applications write data periodically, often in a highly regular manner [25, 49] (both in terms of interval between bursts and the output volume per burst). Such regularity and burstiness suggests the existence of steady, wavelike I/O signatures. Note that although a number of studies have been proposed to optimize the checkpoint interval/volume [19, 20, 39], regular, content-oblivious checkpointing is still the standard practice in large-scale applications [51, 66]. IOSI does not depend on such periodic I/O patterns and handles irregular patterns, as long as the application I/O behavior stays consistent across multiple job runs.

Repeatability: Applications on extreme-scale systems typically run many times. Driven by their science needs, users run the same application with different input data sets and model parameters, which results in repetitive compute and I/O behavior. Therefore, applications tend to have a consistent, identifiable workload signature [16]. To substantiate our claim, we have studied three years worth of Spider server-side I/O throughput logs and Titan job traces for the same time period, and verified that applications have a recurring I/O pattern in terms of frequency and I/O volume. Figure 2 plots statistics of per-user jobs using identical job configurations, which is highly indicative of executions of the same application. We see that certain users, especially those issuing large-scale runs, tend to reuse the same job configuration for many executions.

Overall, the above supercomputing I/O features motivate IOSI to find commonality between multiple noisy server-side log samples. Each sample documents the server-side aggregate I/O traffic during an execution of the same target application, containing different and unknown noise signals. The intuition is that *with a reasonable number of samples, the invariant behavior is likely to belong to the target application.*

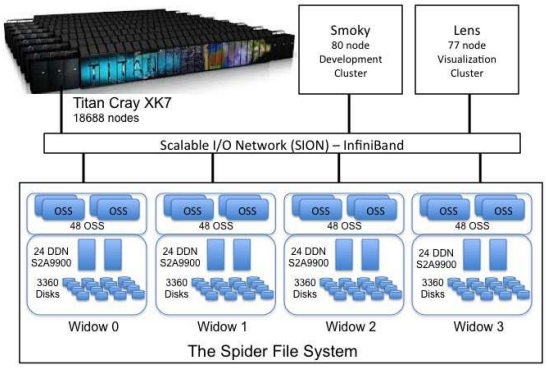


Figure 3: Spider storage system architecture at OLCF.

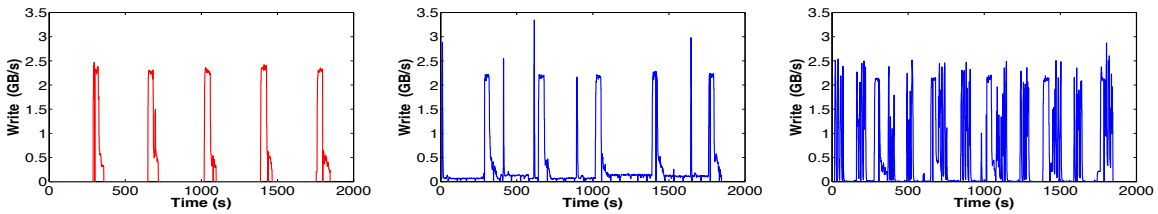
2.2 Titan's Spider Storage Infrastructure

Our prototype development and evaluation use the storage server statistics collected from the Spider center-wide storage system [55] at OLCF, a Lustre-based parallel file system. Spider currently serves the world's No. 2 machine, the 27 petaflop Titan, in addition to other smaller development and visualization clusters. Figure 3 shows the Spider architecture, which comprises of 96 Data Direct Networks (DDN) S2A9900 RAID controllers, with an aggregate bandwidth of 240 GB/s and over 10 PBs of storage from 13,440 1-TB SATA drives. Access is through the object storage servers (OSSs), connected to the RAID controllers in a fail-over configuration. The compute platforms connect to the storage infrastructure over a multistage InfiniBand network, SION (Scalable I/O Network). Spider has four partitions, widow$[0-3]$, with identical setup and capacity. Users can choose any partition(s) for their jobs.

Spider has been collecting server-side I/O statistics from the DDN RAID controllers since 2009. These controllers provide a custom API for querying performance and status information over the management Ethernet network. A custom daemon utility [43] polls the controllers for bandwidth and IOPS at 2-second intervals and stores the results in a MySQL database. Bandwidth data are automatically reported from individual DDN RAID controllers and aggregated across all widow partitions to obtain the overall file system bandwidth usage.

2.3 Problem Definition: Parallel Application I/O Signature Identification

As mentioned earlier, IOSI aims to identify the I/O signature of a parallel application, from zero-overhead, aggregate, server-side I/O throughput logs that are *al-*



(a) IOR_A target signature (b) Sample IOR_AS1 (c) Sample IOR_AS6

Figure 4: I/O signature of IOR_A and two samples

ready being collected. IOSI's input includes (1) the start and end times of the target application's multiple executions in the past, and (2) server-side logs that contain the I/O throughput generated by those runs (as well as unknown I/O loads from concurrent activities). The output is the extracted I/O signature of the target application.

We define an application's *I/O signature* as the I/O throughput it generates at the server-side storage of a given parallel platform, for the duration of its execution. In other words, if this application runs alone on the target platform without any noise from other concurrent jobs or interactive/maintenance workloads, the server-side throughput log during its execution will be its signature. It is virtually impossible to find such "quiet time" once a supercomputer enters the production phase. Therefore, IOSI needs to "mine" the true signature of the application from server-side throughput logs, collected from its multiple executions. Each execution instance, however, will likely contain different noise signals. We refer to each segment of such a noisy server-side throughput log, punctuated by the start and end times of the execution instance, a *"sample"*. Based on our experience, generally 5 to 10 samples are required for getting the expected results. Note that there are long-running applications (potentially several days for each execution). It is possible for IOSI to extract a signature from even partial samples (e.g., from one tenth of an execution time period), considering the self-repetitive I/O behavior of large-scale simulations.

Figure 4 illustrates the signature extraction problem using a pseudo-application, IOR_A, generated by IOR [1], a widely used benchmark for parallel I/O performance evaluation. IOR supports most major HPC I/O interfaces (e.g., POSIX, MPIIO, HDF5), provides a rich set of user-specified parameters for I/O operations (e.g., file size, file sharing setting, I/O request size), and allows users to configure iterative I/O cycles. IOR_A exhibits a periodic I/O pattern typical in scientific applications, with 5 distinct *I/O bursts*. Figure 4(a) shows its I/O signature, obtained from a quiet Spider storage system partition during Titan's maintenance window. Figures 4(b) and 4(c) show its two server-side I/O log samples when executed alongside other real applications and interactive I/O activities. These samples clearly demonstrate the existence of varying levels of noise. Thus, IOSI's purpose is to find the common features from multiple samples (e.g., Figures 4(b) and 4(c)), to obtain an I/O signature that approximates the original (Figure 4(a)).

3 Related Work

I/O Access Patterns and I/O Signatures: Miller and Katz observed that scientific I/O has highly sequential and regular accesses, with a period of CPU processing followed by an intense, bursty I/O phase [25]. Carns et al. noted that HPC I/O patterns tend to be repetitive across different runs, suggesting that I/O logs from prior runs can be a useful resource for predicting future I/O behavior [16]. Similar claims have been made by other studies on the I/O access patterns of scientific applications [28, 47, 53]. Such studies strongly motivate IOSI's attempt to identify common and distinct I/O bursts of an application from multiple noisy, server-side logs.

Prior work has also examined the identification and use of I/O signatures. For example, the aforementioned work by Carns et al. proposed a methodology for continuous and scalable characterization of I/O activities [16]. Byna and Chen also proposed an I/O prefetching method with runtime and post-run analysis of applications' I/O signatures [15]. A significant difference is that IOSI is designed to automatically extract I/O signatures from existing coarse-grained server-side logs, while prior approaches for HPC rely on client-side tracing (such as MPI-IO instrumentation). For more generic application workload characterization, a few studies [52, 57, 64] have successfully extracted signatures from various server-side logs.

Client-side I/O Tracing Tools: A number of tools have been developed for general-purpose client-side instrumentation, profiling, and tracing of generic MPI and CPU activity, such as mpiP [60], LANL-Trace [2], HPCT-IO [54], and TRACE [42]. The most closely related to IOSI is probably Darshan [17]. It performs low-overhead, detailed I/O tracing and provides powerful post-processing of log files. It outputs a large collection of aggregate I/O characteristics such as operation counts and request size histograms. However, existing client-side tracing approaches suffer from the limitations mentioned in Section 1, such as installation/linking require-

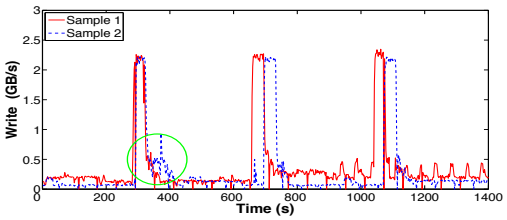


Figure 5: Drift and scaling of I/O bursts across samples

ments, voluntary participation, and producing additional client I/O traffic. IOSI's server-side approach allows it to handle applications using any I/O interface.

Time-series Data Alignment There have been many studies in this area [6, 9, 10, 27, 38, 46]. Among them, dynamic time warping (DTW) [10, 46] is a well-known approach for comparing and averaging a set of sequences. Originally, this technique was widely used in the speech recognition community for automatic speech pattern matching [23]. Recently, it has been successfully adopted in other areas, such as data mining and information retrieval, for automatically addressing time deformations and aligning time-series data [18, 30, 33, 67]. Due to its maturity and existing adoption, we choose DTW for comparison against the IOSI algorithms.

4 Approach Overview

Thematic to IOSI is the realization that the noisy, server-side samples contain common, periodic I/O bursts of the target application. It exploits this fact to extract the I/O signature, using a rich set of statistical techniques. Simply correlating the samples is not effective in extracting per-application I/O signatures, due to a set of challenges detailed below.

First, the server-side logs do not distinguish between different workloads. They contain I/O traffic generated by many parallel jobs that run concurrently, as well as interactive I/O activities (e.g., migrating data to and from remote sites using tools like FTP). Second, I/O contention not only generates "noise" that is superimposed on the true I/O throughput generated by the target application, but also *distorts* it by slowing down its I/O operations. In particular, I/O contention produces *drift* and *scaling* effects on the target application's I/O bursts. The degree of drift and scaling varies from one sample to another. Figure 5 illustrates this effect by showing two samples (solid and dashed) of a target application performing periodic writes. It shows that I/O contention can cause shifts in I/O burst timing (particularly with the last two bursts in this case), as well as changes in burst duration (first burst, marked with oval). Finally, the noise level and the runtime variance caused by background I/O further create the following dilemma in processing the I/O signals: IOSI has to rely on the application's I/O bursts to properly *align* the noisy samples as they are

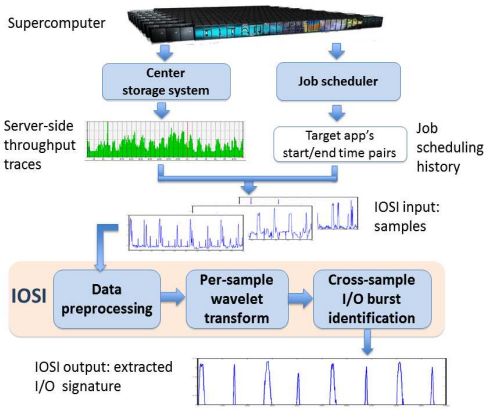


Figure 6: IOSI overview

the only common features; at the same time, it needs the samples to be reasonably aligned to *identify* the common I/O bursts as belonging to the target application.

Recognizing these challenges, IOSI leverages an array of signal processing and data mining tools to discover the target application's I/O signature using a black-box approach, unlike prior work based on white-box models [17, 59]. Recall that IOSI's purpose is to render a *reliable estimate of user-applications' bandwidth needs*, instead of to optimize individual applications' I/O operations. Black-box analysis is better suited here for generic and non-intrusive pattern collection.

The overall context and architecture of IOSI are illustrated in Figure 6. Given a target application, multiple samples from prior runs are collected from the server-side logs. Using such a sample set as input, IOSI outputs the extracted I/O signature by *mining* the common characteristics hidden in the sample set. Our design comprises of three phases:

1. **Data preprocessing**: This phase consists of four key steps: outlier elimination, sample granularity refinement, runtime correction, and noise reduction. The purpose is to prepare the samples for alignment and I/O burst identification.
2. **Per-sample wavelet transform**: To utilize "I/O bursts" as common features, we employ wavelet transform to distinguish and isolate individual bursts from the noisy background.
3. **Cross-sample I/O burst identification**: This phase identifies the common bursts from multiple samples, using a grid-based clustering algorithm.

5 IOSI Design and Algorithms

In this section, we describe IOSI's workflow, step by step, using the aforementioned IOR_A pseudo-application (Figure 4) as a running example.

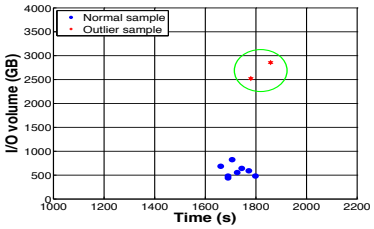
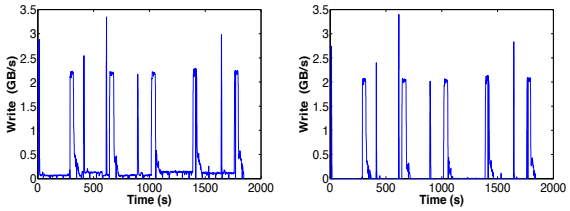


Figure 7: Example of outlier elimination



(a) Before noise reduction (b) After noise reduction

Figure 8: IOR_A samples after noise reduction

5.1 Data Preprocessing

Given a target application, we first compare the job log with the I/O throughput log, to obtain I/O samples from the application's multiple executions, particularly *by the same user* and with *the same job size* (in term of node counts). As described in Section 2, HPC users tend to run their applications repeatedly.

From this set, we then eliminate outliers – samples with significantly heavier noise signals or longer/shorter execution time.[1] Our observation from Spider is that despite unpredictable noise, the majority of the samples (from the same application) bear considerable similarity. Intuitively, including the samples that are apparently significantly skewed by heavy noise is counter-productive. We perform *outlier elimination* by examining (1) the application execution time and (2) the volume of data written within the sample (the "area" under the server-side throughput curve). Within this 2-D space, we apply the *Local Outlier Factor (LOF)* algorithm [12], which identifies observations beyond certain threshold as outliers. Here we set the threshold μ as the mean of the sample set. Figure 7 illustrates the distribution of execution times and I/O volumes among 10 IOR_A samples collected on Spider, where two of the samples (dots within the circle) are identified by LOF as outliers.

Next, we perform sample granularity refinement, by decreasing the data point interval from 2 seconds to 1 using simple linear interpolation [22]. Thus, we insert an extra data point between two adjacent ones, which turns out to be quite helpful in identifying short bursts that last for only a few seconds. The value of each extra data point is the average value of its adjacent data points. It is particularly effective in retaining the amplitude of narrow bursts during the subsequent WT stage.

In the third step, we perform *duration correction* on the remaining sample data set. This is based on the observation that noise can only prolong application execution, hence the sample with the *shortest* duration received the *least* interference, and is consequently closest in duration to the target signature. We apply a simple trimming process to correct the drift effect mentioned in Section 4, preparing the samples for subsequent correlation and alignment. This procedure discards data points at regular intervals to shrink each longer sample to match the shortest one. For example, if a sample is 4% longer than the shortest one, then we remove from it the 1st, 26th, 51st, ..., data points. We found that after outlier elimination, the deviation in sample duration is typically less than 10%. Therefore such trimming is not expected to significantly affect the sample data quality.

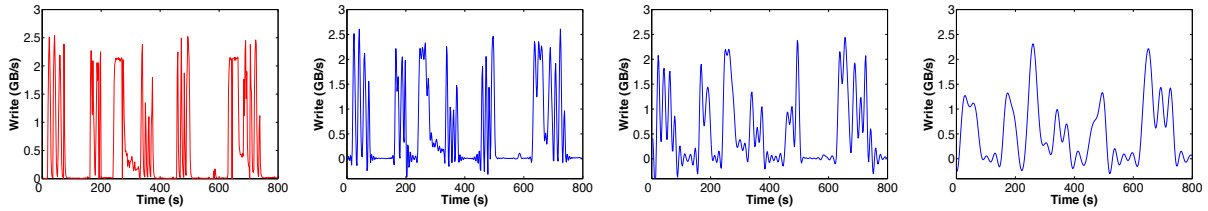
Finally, we perform preliminary *noise reduction* to remove *background noise*. While I/O-intensive applications produce heavy I/O bursts, the server-side log also reports I/O traffic from interactive user activities and maintenance tasks (such as disk rebuilds or data scrubbing by the RAID controllers). Removing this type of persistent background noise significantly helps signature extraction. In addition, although such noise does not significantly distort the shape of application I/O bursts, having it embedded (and duplicated) in multiple application's I/O signatures will cause inaccuracies in I/O-aware job scheduling. To remove background noise, IOSI (1) aggregates data points from all samples, (2) collects those with a value lower than the overall average throughput, (3) calculates the *average background noise level* as the mean throughput from these selected data points, and (4) lowers each sample data point by this average background noise level, producing zero if the result is negative. Figure 8(b) shows the result of such preprocessing, and compared to the original sample in Figure 8(a), the I/O bursts are more pronounced. The I/O volume of IOR_AS1 was trimmed by 26%, while the background noise level was measured at 0.11 GB/s.

5.2 Per-Sample Wavelet Transform

As stated earlier, scientific applications tend to have a bursty I/O behavior, justifying the use of *I/O burst* as the basic unit of signature identification. An I/O burst indicates a phase of high I/O activity, distinguishable from the background noise over a certain duration.

With less noisy samples, the burst boundaries can be easily found using simple methods such as *first difference* [50] or *moving average* [62]. However, with noisy samples identifying such bursts becomes challenging, as there are too many ups and downs close to each other. In particular, it is difficult to do so without knowing the cutoff threshold for a "bump" to be considered a candidate I/O burst. Having too many or too few candidates

[1] Note that shorter execution time can happen with restart runs resuming from a prior checkpoint.



(a) Preprocessed IOR_AS6 segment (b) After WT (Decomposition level 1) (c) After WT (Decomposition level 2) (d) After WT (Decomposition level 3)

Figure 9: *dmey* WT results on a segment of IOR_AS6

can severely hurt our sample alignment in the next step.

To this end, we use a WT [21, 41, 63] to smooth samples. WT has been widely applied to problems such as filter design [14], noise reduction [35], and pattern recognition [24]. With WT, a time-domain signal can be decomposed into low-frequency and high-frequency components. The approximation information remains in the low-frequency component, while the detail information remains in the high-frequency one. By carefully selecting the *wavelet function* and *decomposition level* we can observe the major bursts from the low-frequency component. They contain the most energy of the signal and are isolated from the background noise.

By retaining the temporal characteristics of the time-series data, WT brings an important feature not offered by widely-used alternative techniques such as Fourier transform [11]. We use WT to clarify individual bursts from their surrounding data, without losing the temporal characteristics of the time-series sample.

WT can use quite a few wavelet families [4, 45], such as *Haar*, *Daubechies*, and *Coiflets*. Each provides a transform highlighting different frequency and temporal characteristics. For IOSI, we choose *discrete Meyer (dmey)* [40] as the mother wavelet. Due to its smooth profile, the approximation part of the resulting signal consists of a series of smooth waves. Its output consists of a series of waves where the center of the wave troughs can be easily identified as wave boundaries.

Figures 9(a) and Figures 9(b), 9(c), 9(d) illustrate a segment of IOR_AS6 and its dmey WT results, respectively. With a WT, the high-frequency signals in the input sample are smoothed, producing low-frequency components that correlate better with the target application's periodic I/O. However, here the waves cannot be directly identified as I/O bursts, as a single I/O burst from the application's point of view may appear to have many "sub-crests", separated by minor troughs. This is due to throughput variance caused by either application behavior (such as adjacent I/O calls separated by short computation/communication) or noise, or both. To prevent creating many such "sub-bursts", we use the mean height of all wave crests for filtering – only the troughs lower than this threshold are used for separating bursts.

Another WT parameter to consider is the *decomposition level*, which determines the level of detailed information in the results. The higher the decomposition level, the fewer details are shown in the low-frequency component, as can be seen from Figures 9(b), 9(c) and 9(d). With a decomposition level of 1 (e.g. Figures 9(b)), the wavelet smoothing is not sufficient for isolating burst boundaries. With a higher decomposition level of 3 the narrow bursts fade out rapidly, potentially missing target bursts. IOSI uses a decomposition level of 2 to better retain the bursty nature of the I/O signature.

5.3 Cross-Sample I/O Burst Identification

Next, IOSI correlates all the pre-processed, and wavelet transformed samples to identify common I/O bursts. To address the circular dependency challenge mentioned earlier between alignment and common feature identification, we adapt a grid-based clustering approach called CLIQUE [8]. It performs multi-dimensional data clustering by identifying grids (called *units*) where there is higher *density* (number of data points within the unit). CLIQUE treats each such *dense unit* as a "seed cluster" and grows it by including neighboring dense units.

CLIQUE brings several advantages to IOSI. First, its model fits well with our context: an I/O burst from a given sample is mapped to a 2-D data point, based on its *time* and *shape* attributes. Therefore, data points from different samples close to each other in the 2-D space naturally indicate common I/O bursts. Second, with controllable grid width and height, IOSI can better handle burst drifts (more details given below). Third, CLIQUE performs well for scenarios with far-apart clusters, where inter-cluster distances significantly exceed those between points within a cluster. As parallel applications typically limit their "I/O budget" (fraction of runtime allowed for periodic I/O) to 5%-10%, individual I/O bursts normally last seconds to minutes, with dozens of minutes between adjacent bursts. Therefore, CLIQUE is not only effective for IOSI, but also efficient, as we do not need to examine too far around the burst-intensive areas. Our results (Section 6) show that it outperforms the widely used DTW time-series alignment algorithm [10], while incurring significantly lower overhead.

We make two adjustments to the original CLIQUE

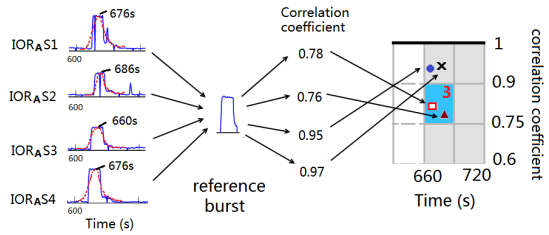


Figure 10: Mapping IOR_A I/O bursts to 2-D points

algorithm. Considering the I/O bursts are sufficiently spaced from each other in a target application's execution, we limit the growth of the cluster to the immediate *neighborhood* of a dense unit: the units that are adjacent to it. Also, we have modified the density calculation to focus not on the sheer number of data points in a unit, but on the number of *different samples* with bursts there. The intuition is that a common burst from the target application should have most (if not all) samples agree on its existence. Below, we illustrate with IOR_A the process of IOSI's common burst identification.

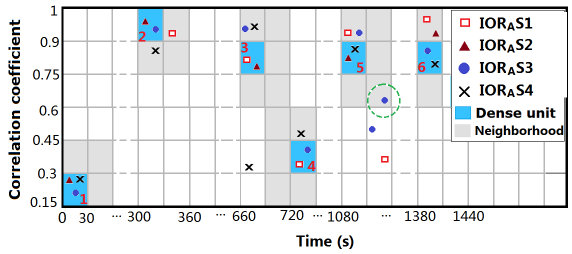


Figure 11: CLIQUE 2-D grid containing IOR_A bursts

To use our adapted CLIQUE, we need to first discretize every sample s_i into a group of 2-D points, each representing one I/O burst identified after a WT. Given its jth I/O burst $b_{i,j}$, we map it to point $\langle t_{i,j}, c_{i,j} \rangle$. Here $t_{i,j}$ is the time of the wave crest of $b_{i,j}$, obtained after a WT, while $c_{i,j}$ is the correlation coefficient between $b_{i,j}$ and a *reference burst*. To retain the details describing the shape of the I/O burst, we choose to use the pre-WT burst in calculating $c_{i,j}$, though the burst itself was identified using a WT. Note that we rely on the transitive nature of correlation ("bursts with similar correlation coefficient to the common reference burst tend to be similar to each other"), so the reference burst selection does not have a significant impact on common burst identification. In our implementation, we use the "average burst" identified by WT across all samples.

Figure 10 shows how we mapped 4 I/O bursts, each from a different IOR_A sample. Recall that WT identifies each burst's start, peak, and end points. The x coordinate for each burst shows "when it peaked," derived using the post-WT wave (dotted line). The y coordinate shows "how similar it is to a reference burst shape," calculated using the pre-WT sample (solid lines).

Therefore, our CLIQUE 2-D data space has an x range of $[0, t]$ (where t is the adjusted sample duration after preprocessing) and a y range of $[0, 1]$. It is partitioned into uniform 2-D grids (units). Defining the unit width and height is critical for CLIQUE, as overly small or large grids will obviously render the density index less useful. Moreover, even with carefully selected width and height values, there is still a chance that a cluster of nodes are separated into different grids, causing CLIQUE to miss a dense unit.

To this end, instead of using only one pair of width-height values, IOSI tries out multiple grid size configurations, each producing an extracted signature. For width, it sets the lower bound as the average I/O burst duration across all samples and upper bound as the average time distance between bursts. For a unit height, it empirically adopts the range between 0.05 and 0.25. Considering the low cost of CLIQUE processing with our sample sets, IOSI uniformly samples this 2-D parameter space (e.g., with 3-5 settings per dimension), and takes the result that identified the most data points as belonging to common I/O bursts. Due to the strict requirement of identifying common bursts, we have found in our experiments that missing target bursts is much more likely to happen than including fake bursts in the output signature. Figure 11 shows the resulting 2-D grid, containing points mapped from bursts in four IOR_A samples.

We have modified the original *dense unit* definition as follows. Given s samples, we calculate the *density* of a unit as "the number of samples that have points within this unit". If a unit meets a certain density threshold $\lceil \gamma * s \rceil$, where γ is a controllable parameter between 0 and 1, the unit is considered dense. Our experiments used a γ value of 0.5, requiring each dense unit to have points from at least 2 out of the 4 samples. All dense units are marked with a dark shade in Figure 11.

Due to the time drift and shape distortion caused by noise, nodes from different samples representing the same I/O burst could be partitioned by grid lines. As mentioned earlier, for each dense unit, we only check its immediate neighborhood (shown in a lighter shade in Figure 11) for data points potentially from a common burst. We identify *dense neighborhoods* (including the central dense unit) as those meeting a density threshold of $\lceil \kappa * s \rceil$, where κ is another configurable parameter with value larger than γ (e.g., 0.9).

Note that it is possible for the neighborhood (or even a single dense unit) to contain multiple points from the same sample. IOSI further identifies points from the common I/O burst using a *voting* scheme. It allows up to one point to be included from each sample, based on the total normalized Euclidean distance from a candidate point to peers within the neighborhood. From each sample, the data point with the lowest total distance is

selected. In Figure 11, the neighborhood of dense unit 5 contains two bursts from IOR_AS3 (represented by dots). The burst in the neighborhood unit (identified by the circle) is discarded using our voting algorithm. As the only "redundant point" within the neighborhood, it is highly likely to be a "fake burst" from other concurrently running I/O-intensive applications. This can be confirmed by viewing the original sample IOR_AS3 in Figure12(b), where a tall spike not from IOR_A shows up around the 1200th second.

5.4 I/O Signature Generation

Given the common bursts from dense neighborhoods, we proceed to sample alignment. This is done by aligning all the data points in a common burst to the average of their x coordinate values. Thereafter, we generate the actual I/O signature by sweeping along the x (time) dimension of the CLIQUE 2-D grid. For each dense neighborhood identified, we generate a corresponding I/O burst at the aligned time interval, by averaging the bursts mapped to the selected data points in this neighborhood. Here we used the bursts after preprocessing, but before WT.

6 Experimental Evaluation

We have implemented the proof-of-concept IOSI prototype using Matlab and Python. To validate IOSI, we used IOR to generate multiple pseudo-applications with different I/O write patterns, emulating write-intensive scientific applications. In addition, we used S3D [31, 56], a massively parallel direct numerical simulation solver developed at Sandia National Laboratory for studying turbulent reacting flows.

Figure 13(a), 13(e) and 13(i) are the true I/O signatures of the three IOR pseudo-applications, IOR_A, IOR_B, and IOR_C. These pseudo-applications were run on the Smoky cluster using 256 processes, writing to the Spider center-wide parallel file system. Each process was configured to write sequentially to a separate file (stripe size of 1MB, stripe count of 4) using MPI-IO. We were able to obtain "clean" signatures (with little noise) for these applications by running our jobs when Titan was not in production use (under maintenance) and one of the file system partitions was idle. Among them, IOR_A represents simple periodic checkpointing, writing the same volume of data at regular intervals (128GB every 300s). IOR_B also writes periodically, but alternates between two levels of output volume (64GB and 16GB every 120s), which is typical of applications with different frequencies in checkpointing and results writing (e.g., writing intermediate results every 10 minutes but checkpointing every 30 minutes). IOR_C has more complex I/O patterns, with three different write cycles repeated periodically (one output phase every 120s, with output size cycling through 64GB, 32GB, and 16GB).

6.1 IOR Pseudo-Application Results

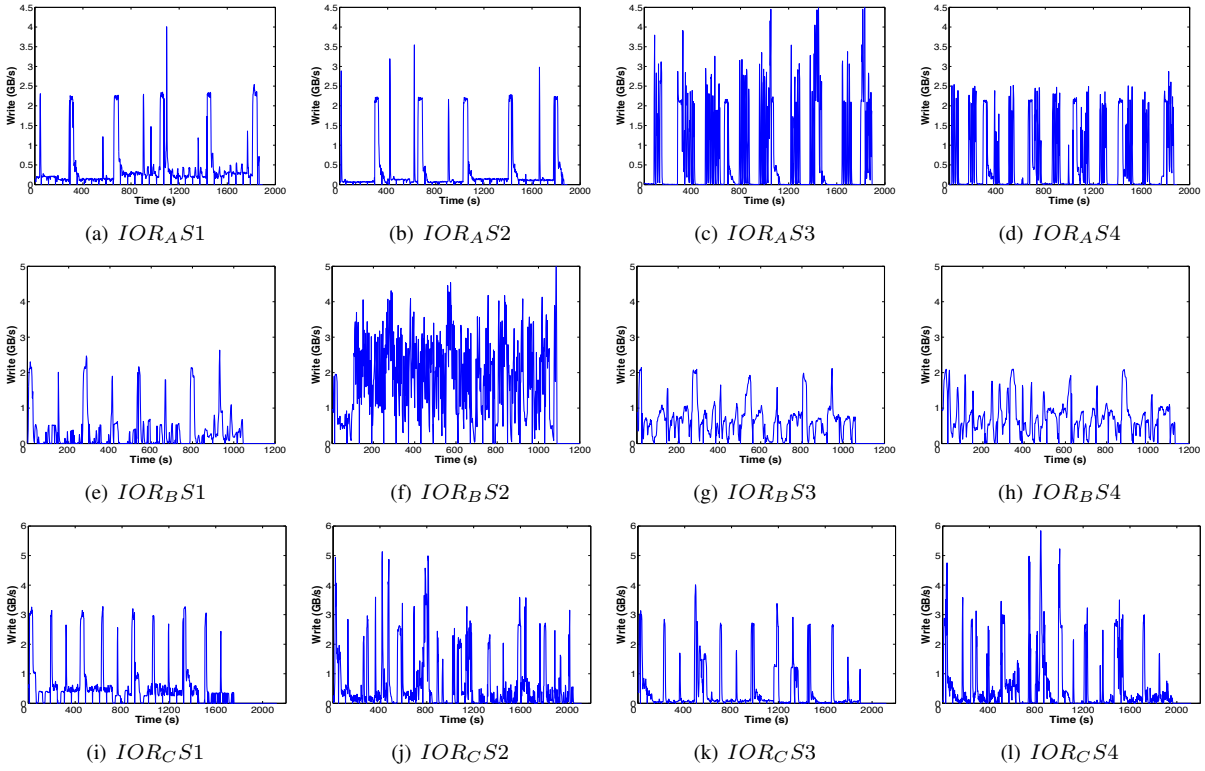
To validate IOSI, the IOR pseudo-applications were were run at different times of the day, over a two-week period. Each application was run at least 10 times. During this period, the file system was actively used by Titan and other clusters. The I/O activity captured during this time is the noisy server-side throughput logs. From the scheduler's log, we identified the execution time intervals for the IOR runs, which were then intersected with the I/O throughput log to obtain per-application samples.

It is worth noting that the I/O throughput range of all of the IOR test cases is designed to be 2-3GB/s. After analyzing several months of Spider log data, we observed that it is this low-bandwidth range that is highly impacted by background noise. If the bandwidth of the application is much higher (say 20GB/s), the problem becomes much easier, since there is less background noise that can achieve that bandwidth level to interfere.

Due to the space limit, we only show four samples for each pseudo-app in Figure 12. We observe that most of them show human-visible repeated patterns that overlap with the target signatures. There is, however, significant difference between the target signature and any individual sample. The samples show a non-trivial amount of "random" noise, sometimes (e.g., IOR_AS1) with distinct "foreign" repetitive pattern, most likely from another application's periodic I/O. Finally, a small number of samples are noisy enough to make the target signature appear overwhelmed (which should be identified as outliers and discarded from signature extraction).

Figure 13 presents the original signatures and the extracted signatures using three approaches: IOSI with and w/o data preprocessing, plus DTW with data preprocessing. As introduced in Section 3, DTW is a widely used approach for finding the similarity between two data sequences. In our problem setting, similarity means a match in I/O bursts from two samples. We used sample preprocessing to make a fair comparison between DTW and IOSI. Note that IOSI without data preprocessing utilizes samples after granularity refinement, to obtain extracted I/O signatures with similar length across all three methods tested.

Since DTW performs pair-wise alignment, it is unable to perform effective global data alignment across multiple samples. In our evaluation, we apply DTW as follows. We initially assign a pair of samples as input to DTW, and feed the result along with another sample to DTW again. This process is repeated until all samples are exhausted. We have verified that this approach performs better (in terms of both alignment accuracy and processing overhead) than the alternative of averaging all pair-wise DTW results, since it implicitly carries

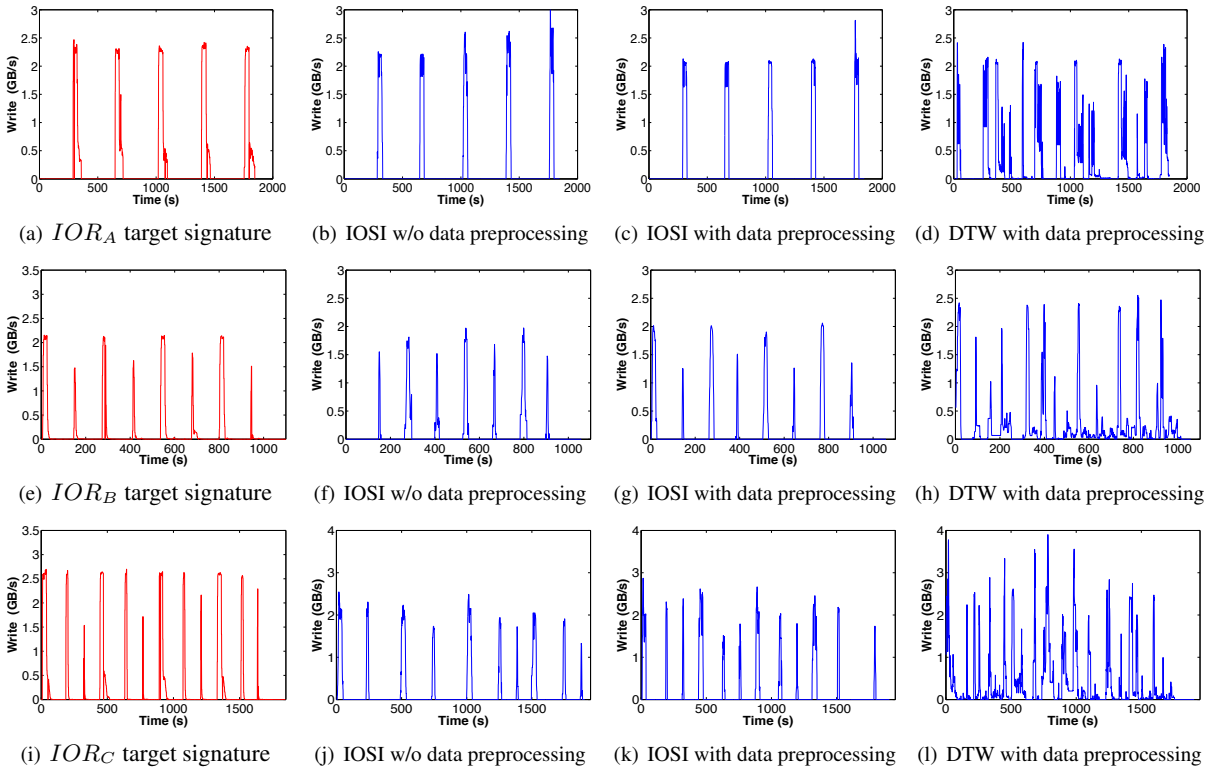


(a) IOR_AS1 (b) IOR_AS2 (c) IOR_AS3 (d) IOR_AS4

(e) IOR_BS1 (f) IOR_BS2 (g) IOR_BS3 (h) IOR_BS4

(i) IOR_CS1 (j) IOR_CS2 (k) IOR_CS3 (l) IOR_CS4

Figure 12: Samples from IOR test cases

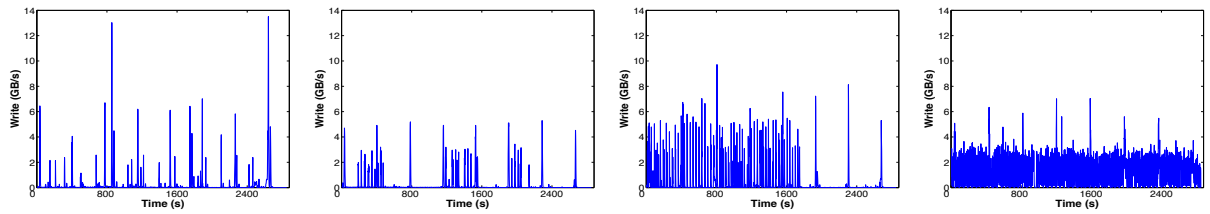


(a) IOR_A target signature (b) IOSI w/o data preprocessing (c) IOSI with data preprocessing (d) DTW with data preprocessing

(e) IOR_B target signature (f) IOSI w/o data preprocessing (g) IOSI with data preprocessing (h) DTW with data preprocessing

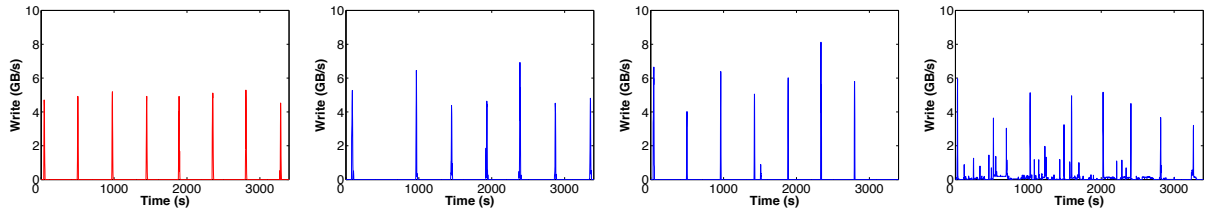
(i) IOR_C target signature (j) IOSI w/o data preprocessing (k) IOSI with data preprocessing (l) DTW with data preprocessing

Figure 13: Target and extracted I/O signatures of IOR test cases



(a) $S3DS1$ (b) $S3DS2$ (c) $S3DS3$ (d) $S3DS4$

Figure 14: S3D samples



(a) S3D target I/O signature (b) IOSI w/o data preprocessing (c) IOSI with data preprocessing (d) DTW with data preprocessing

Figure 15: S3D target I/O signature and extracted I/O signature by IOSI and DTW

out global data alignment. Still, DTW generated significantly lower-quality signatures, especially with more complex I/O patterns, due to its inability to reduce noise. For example, DTW's IOR_C (Figure 13(l)) signature appears to be dominated by noise.

In contrast, IOSI (with or w/o data preprocessing) generated output signatures with much higher fidelity. In both cases, IOSI is highly successful in capturing I/O bursts in the time dimension (with small, yet visible errors in the vertical height of the bursts). Without preprocessing, IOSI missed 3 out of the 25 I/O bursts from all pseudo-applications. With preprocessing, however, the symptom is much improved (no burst missed).

6.2 S3D Results

Next, we present results with the aforementioned large-scale scientific application, S3D. S3D was run on Titan and the Spider file system. S3D performs periodic checkpointing I/O, with each process generating 3.4 MB of data. Figure 14 shows selected samples from multiple S3D runs, where we see a lot of I/O interference since both Titan and Spider were being used in production mode. Unlike IOR, we were not able to run S3D on a quiescent Spider file system partition to obtain its "clean" signature to validate IOSI. Instead, we had to use client-side I/O tracing, to produce the target I/O signature (Figure 15(a)). The I/O signature also displays variance in peak bandwidth, common in real-world, large job runs. Again, we extracted the I/O signature from the samples using IOSI (with and without data preprocessing), plus DTW with preprocessing (Figure 15).

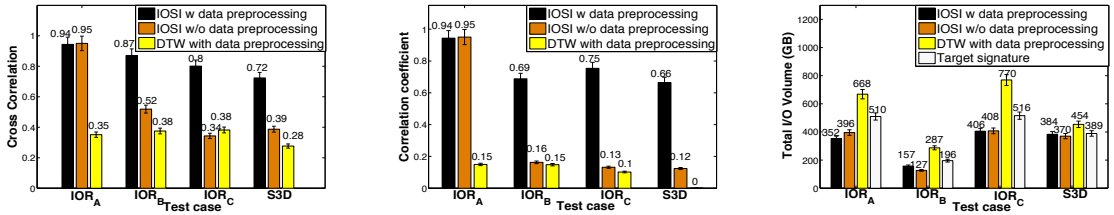
As in the case of IOR, IOSI with data preprocessing performs better than IOSI without data preprocessing and DTW. This result suggests that IOSI is able to the extract I/O signatures of real-world scientific applications from noisy throughput logs, collected from a very busy supercomputing center. While both versions of IOSI missed an I/O burst, the data preprocessing helps deliver better alignment accuracy (discussed in Figures 16(a) and 16(b)). The presence of heavy noise in the samples likely caused DTW's poor performance.

6.3 Accuracy and Efficiency Analysis

We quantitatively compare the accuracy of IOSI and DTW using two commonly used similarity metrics, cross correlation (Figure 16(a)) and correlation coefficient (Figure 16(b)). Correlation coefficient measures the strength of the linear relationship between two samples. Cross correlation [65] is a similarity measurement that factors in the drift in a time series data set. Figure 16 portraits these two metrics, as well as the total I/O volume comparison, between the extracted and the original application signature.

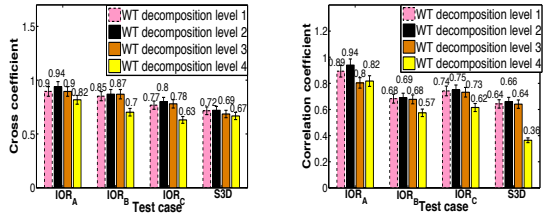
Note that correlation coefficient is inadequate to characterize the relationship between the two time series when they are not properly aligned. For example, with IOR_B, the number of bursts in the extracted signatures by IOSI with and without data preprocessing is very close. However, the one without preprocessing suffers more burst drift compared to the original signature. Cross correlation appears more tolerant to IOSI without preprocessing compared to correlation coefficient. Also, IOSI significantly outperforms DTW (both with preprocessing), by a factor of 2.1-2.6 in cross correlation, and 4.8-66.0 in correlation coefficient.

Note that the DTW correlation coefficient for S3D is too small to show. Overall, IOSI with preprocessing achieves a cross correlation between 0.72 and 0.95, and



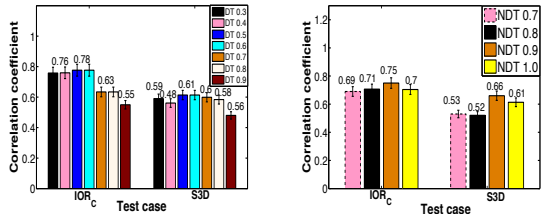
(a) Cross correlation (b) Correlation coefficient (c) Total I/O volume

Figure 16: Result I/O signature accuracy evaluation



(a) Cross correlation (b) Correlation coefficient

Figure 17: IOSI - WT sensitivity analysis



(a) Density threshold (DT) (b) Neighborhood density threshold (NDT)

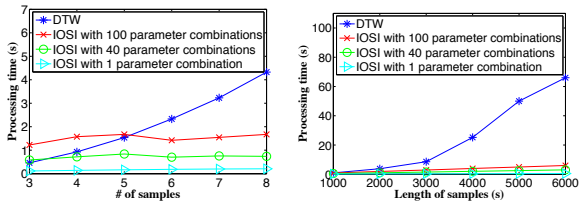
Figure 18: IOSI - Clustering sensitivity analysis

a correlation coefficient between 0.66 and 0.94.

We also compared the total volume of I/O traffic (i.e., the "total area" below the signature curve), shown in Figure 16(c). IOSI generates I/O signatures with a total I/O volume closer to the original signature than DTW does. It is interesting that without exception, IOSI and DTW err on the lower and higher side, respectively. The reason is that DTW tends to include foreign I/O bursts, while IOSI's WT process may "trim" the I/O bursts in its threshold-based burst boundary identification.

Next, we performed sensitivity analysis on the tunable parameters of IOSI, namely the *WT decomposition level*, and *density threshold/neighborhood density threshold in CLIQUE clustering*. As discussed in Section 5, we used a WT decomposition level of 2 in IOSI. In Figures 17(a) and 17(b), we compare the impact of WT decomposition levels using both cross correlation and correlation coefficient. Figure 17(a) shows that IOSI does better with a decomposition level of 2, compared to levels 1, 3 and 4. Similarly, Figure 17(b) shows that the correlation coefficient is the best at the WT decomposition level of 2.

In Figure 18(a), we tested IOSI with different density thresholds $\lceil \gamma * s \rceil$ in CLIQUE clustering, where γ is the controllable factor and s is the number of samples. In IOSI, the default γ value is 50%. From Figure 18(a) we noticed a peak correlation coefficient at γ value of around 50%. There is significant performance degradation at over 70%, as adjacent bursts may be grouped to form a single burst. In Figure 18(b), we tested IOSI with different neighborhood density thresholds $\lceil \kappa * s \rceil$, where κ is another configurable factor with value larger than γ. IOSI used 90% as the default value of κ. Figure 18(b) suggests that lower thresholds perform poorly, as more neighboring data points deteriorates the quality of identified I/O bursts. With a threshold of 100%, we expect bursts from all samples to be closely aligned, which is impractical.

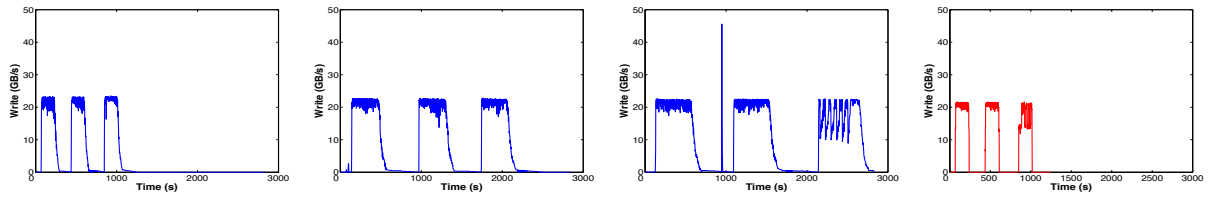


(a) Scalability in # of samples (b) Scalability in sample duration

Figure 19: Processing time analysis

Finally, we analyze the processing time overhead of these methods. IOSI involves mainly two computation tasks: wavelet transform and CLIQUE clustering. The complexity of WT (discrete) is *O(n)* [29] and CLIQUE clustering is $O(C^k + nk)$ [32], where k is the highest dimensionality, n the number of input points, and C the number of clusters. In our CLIQUE implementation, k is set to 2 and C is also a small number. Therefore we assume C^k as a constant, resulting in a complexity of *O(n)*, leading to the overall linear complexity of IOSI. DTW, on the other hand, has a complexity of $O(mn)$ [34], where m and n are the lengths of the two input arrays.

Experimental results confirm the above analysis. In Figure 19(a), we measure the processing time with different sample set sizes (each sample containing around 2000 data points). For IOSI, the processing time appears to stay flat as more samples are used. This is because the CLIQUE clustering time, which is rather independent of the number of samples and depends more on the number



(a) 160-node job (b) 320-node job (c) 640-node job (d) IOSI Result

Figure 20: Weak scaling sample and IOSI extracted I/O signature

of grids, dominates IOSI's overhead. Even with 100 2-D IOSI parameter settings (for the CLIQUE grid size), DTW's cost catches up with 5 samples and grows much faster beyond this point. Figure 19(b) shows results with 8 samples, at varied sample lengths. We see that IOSI processing time increases linearly while DTW displays a much faster growth. To give an idea of its feasibility, IOSI finishes processing three months of Spider logs (containing 80,815 job entries) in 72 minutes.

6.4 Discussion

I/O signatures and application configurations Scientific users tend to have a pattern of I/O behavior. However, they do scale applications with respect to the number of nodes, resulting in similar I/O characteristics. In Figure 20, we show the I/O pattern of a real Titan user, running jobs with three different node counts (160, 320, and 640). From Figures 20(a)-20(c), we observe that the total I/O volume increases linearly with the node count (weak scaling), but the peak bandwidth remains almost constant. As a result, the time spent on I/O also increases linearly. IOSI can discern such patterns and extract the I/O signature, as shown in Figure 20(d). As described earlier, in the data preprocessing step we perform runtime correction and the samples are normalized to the sample with the shortest runtime. In this case, IOSI normalizes the data sets to that of the shortest job (i.e., the job with the smallest node count), and provides the I/O signature of the application for the smallest job size.

Identifying different user workloads Our tests used a predominant scientific I/O pattern, where applications perform periodic I/O. However, as long as an application exhibits similar I/O behavior across multiple runs, the common I/O pattern can be captured by IOSI as its algorithms make no assumption on periodic behavior.

False-positives and missing bursts False-positives are highly unlikely as it is very difficult to have highly correlated noise behavior across multiple samples. IOSI could miscalculate small-scale I/O bursts if they happen to be dominated by noise in most samples. Increasing the number of samples can help here.

IOSI for resource allocation and scheduling The IOSI generated signature can be used to influence both resource allocation as well as scheduling. Large-scale file systems are typically made available as multiple partitions, with users choosing one or more for their runs. A simple partition allocation strategy would be to let the users choose a set of under-utilized partitions. However, when all partitions are being actively used by multiple users, the challenge is in identifying a set of partitions that will have the least interference on the target application. The IOSI extracted signature can be correlated with the I/O logs of the partitions to identify those that will have a minimal impact on the application. If we are unable to find an optimal partition for an application, the scheduler can even stagger such jobs, preferring others in the queue. The premise here is that finding a partition that better suits the job's I/O needs can help amortize the I/O costs over the entire run. These benefits could outweigh the cost of waiting longer in the queue.

7 Conclusion

We have presented IOSI, a zero-overhead scheme for automatically identifying the I/O signature of data-intensive parallel applications. IOSI utilizes *existing* throughput logs on the storage servers to identify the signature. It uses a suite of statistical techniques to extract the I/O signature from noisy throughput measurements. Our results show that an entirely *data-driven* approach, exploring existing monitoring and job scheduling history can extract substantial application behavior, potentially useful for resource management optimization. In particular, such information gathering does not require any developer effort or internal application knowledge. Such a black-box method may be even more appealing as systems/applications grow larger and more complex.

Acknowledgement

We thank the reviewers and our shepherd, Kimberly Keeton, for constructive comments that have significantly improved the paper. This work was supported in part by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725). This work was also supported in part by the NSF grants CCF-0937690, CCF-0937908, and a NetApp Faculty Fellowship.

References

[1] IOR HPC Benchmark, https://asc.llnl.gov/sequoia/benchmarks/#ior.

[2] Los Alamos National Laboratory open-source LANL-Trace, http://institute.lanl.gov/data/tdata.

[3] Titan, http://www.olcf.ornl.gov/titan/.

[4] Wavelet, http://users.rowan.edu/~polikar/WAVELETS/WTtutorial.html.

[5] Using Cray Performance Analysis Tools. *Document S-2474-51, Cray User Documents, http://docs.cray.com*, 2009.

[6] J. Aach and G. M. Church. Aligning gene expression time series with time warping algorithms. *Bioinformatics*, 17:495–508, 2001.

[7] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[8] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '98)*, 1998.

[9] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval (SPIRE'00)*, 2000.

[10] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *Working Notes of the Knowledge Discovery in Databases Workshop*, 1994.

[11] R. N. Bracewell. *The Fourier transform and its applications*. McGraw-Hill New York, 1986.

[12] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. LOF: identifying density-based local outliers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*, 2000.

[13] S. Bruenn, A. Mezzacappa, W. Hix, J. Blondin, P. Marronetti, O. Messer, C. Dirk, and S. Yoshida. 2d and 3d core-collapse supernovae simulation results obtained with the chimera code. *Journal of Physics: Conference Series*, 180(2009)012018, 2009.

[14] C. S. Burrus, R. A. Gopinath, H. Guo, J. E. Odegard, and I. W. Selesnick. *Introduction to wavelets and wavelet transforms: a primer*, volume 23. Prentice Hall Upper Saddle River, 1998.

[15] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '08)*, 2008.

[16] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and improving computational science storage access through continuous characterization. In *Proceedings of the IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST'11)*, 2011.

[17] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 Characterization of petascale I/O workloads. In *Proceedings of the First Workshop on Interfaces and Abstractions for Scientic Data Storage (IASDS'09)*, 2009.

[18] S. Chu, E. J. Keogh, D. Hart, and M. J. Pazzani. Iterative Deepening Dynamic Time Warping for Time Series. In *Proceedings of the 2nd SIAM International Conference on Data Mining (SDM'02)*, 2002.

[19] J. Daly. A model for predicting the optimum checkpoint interval for restart dumps. In *Proceedings of the 1st International Conference on Computational Science (ICCS'03)*, 2003.

[20] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[21] I. Daubechies. Orthonormal bases of compactly supported wavelets II: Variations on a theme. *SIAM Journal on Mathematical Analysis*, 24:499–519, 1993.

[22] C. de Boor. *A practical guide to splines*. Springer-Verlag New York, 1978.

[23] J. R. Deller, J. G. Proakis, and J. H. Hansen. *Discrete-time processing of speech signals*. IEEE New York, NY, USA, 2000.

[24] P. Du, W. A. Kibbe, and S. M. Lin. Improved peak detection in mass spectrum by incorporating continuous wavelet transform-based pattern matching. *Bioinformatics*, 22(17):2059–2065, 2006.

[25] E.L.Miller and R.H.Katz. Input/output behavior of supercomputing applications. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'91)*, 1991.

[26] G. R. Ganger. Generating Representative Synthetic Workloads: An Unsolved Problem. In *Proceedings of the Computer Measurement Group (CMG'95)*, 1995.

[27] Z. Gong and X. Gu. PAC: Pattern-driven Application Consolidation for Efficient Cloud Computing. In *Proceedings of the 18th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer Telecommunications Systems (MASCOTS'10)*, 2010.

[28] R. Gunasekaran, D. Dillow, G. Shipman, R. Vuduc, and E. Chow. Characterizing Application Runtime Behavior from System Logs and Metrics. In *Proceedings of the Characterizing Applications for Heterogeneous Exascale Systems (CACHES'11)*, 2011.

[29] H. Guo and C. S. Burrus. Fast approximate Fourier transform via wavelets transform. In *Proceedings of the International Symposium on Optical Science, Engineering, and Instrumentation*, 1996.

[30] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer. Automating vertical profiling. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, 2005.

[31] E. R. Hawkes, R. Sankaran, J. C. Sutherland, and J. H. Chen. Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models. *Journal of Physics: Conference Series*, 16(1):65, 2005.

[32] M. Ilango and V. Mohan. A Survey of Grid Based Clustering Algorithms. *International Journal of Engineering Science and Technology*, 2(8):3441–3446, 2010.

[33] E. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and Information Systems*, 7(3):358–386, 2005.

[34] E. J. Keogh and M. J. Pazzani. Scaling up dynamic time warping for datamining applications. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Dining*, 2000.

[35] N. G. Kingsbury. The dual-tree complex wavelet transform: a new technique for shift invariance and directional filters. In *Proceedings of the 8th IEEE Digital Signal Processing (DSP) Workshop*, 1998.

[36] D. Kothe and R. Kendall. Computational science requirements for leadership computing. *Oak Ridge National Laboratory, Technical Report*, 2007.

[37] Z. Kurmas and K. Keeton. Synthesizing Representative I/O Workloads Using Iterative Distillation. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer Telecommunications Systems (MASCOTS'03)*, 2003.

[38] C. Lipowsky, E. Dranischnikow, H. Göttler, T. Gottron, M. Kemeter, and E. Schömer. Alignment of noisy and uniformly scaled time series. In *Proceedings of the Database and Expert Systems Applications (DEXA'09)*, 2009.

[39] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. L. Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *Proceedings of the International Parallel Distributed Processing Symposium (IPDPS'08)*, 2008.

[40] Y. Long, L. Gang, and G. Jun. Selection of the best wavelet base for speech signal. In *Proceedings of the International Symposium on Intelligent Multimedia, Video and Speech Processing*, 2004.

[41] S. Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *Pattern Analysis and Machine Intelligence*, 11(7):674–693, 1989.

[42] M. P. Mesnier, M. Wachs, R. R. Simbasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. R. O'Hallaron. //trace: Parallel trace replay with approximate causal events. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, 2007.

[43] R. Miller, J. Hill, D. A. Dillow, R. Gunasekaran, S. Galen, and D. Maxwell. Monitoring Tools For Large Scale Systems. In *Proceedings of the Cray User Group (CUG'10)*, 2010.

[44] K. Mohror and K. L. Karavanic. An Investigation of Tracing Overheads on High End Systems. Technical report, PSU, 2006.

[45] W. G. Morsi and M. El-Hawary. The most suitable mother wavelet for steady-state power system distorted waveforms. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, 2008.

[46] M. Müller. Dynamic time warping. *Information Retrieval for Music and Motion*, pages 69–84, 2007.

[47] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Sclatter Ellis, and M. Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, 1996.

[48] S. Oral, F. Wang, D. Dillow, G. Shipman, R. Miller, and O. Drokin. Efficient object storage journaling

in a distributed parallel file system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, 2010.

[49] B. Pasquale and G. Polyzos. A static analysis of I/O characteristics of scientific applications in a production workload. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'93)*, 1993.

[50] T. C. Peterson, T. R. Karl, P. F. Jamason, R. Knight, and D. R. Easterling. First difference method: Maximizing station density for the calculation of long-term global temperature change. *Journal of Geophysical Research: Atmospheres (1984–2012)*, 103(D20):25967–25974, 1998.

[51] J. S. Plank and M. G. Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. *Journal of Parallel and distributed Computing*, 61(11):1570–1590, 2001.

[52] A. Povzner, K. Keeton, A. Merchant, C. B. Morrey III, M. Uysal, and M. K. Aguilera. Autograph: automatically extracting workflow file signatures. *ACM SIGOPS Operating Systems Review*, 43(1):76–83, 2009.

[53] P. C. Roth. Characterizing the I/O behavior of scientific applications on the Cray XT. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: held in conjunction with SC'07*, 2007.

[54] S. Seelam, I.-H. Chung, D.-Y. Hong, H.-F. Wen, and H. Yu. Early experiences in application level I/O tracing on Blue Gene systems. In *Proceedings of the International Parallel Distributed Processing Symposium (IPDPS'08)*, 2008.

[55] G. Shipman, D. Dillow, S. Oral, and F. Wang. The Spider Center Wide File System: From Concept to Reality. In *Proceedings of the Cray User Group (CUG'09)*, 2009.

[56] K. Spafford, J. Meredith, J. Vetter, J. Chen, R. Grout, and R. Sankaran. Accelerating S3D: a GPGPU case study. In *Euro-Par 2009 Parallel Processing Workshops*, 2010.

[57] V. Tarasov, S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[58] TOP500 Supercomputer Sites, `http://www.top500.org/`.

[59] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker. Parallel I/O performance: From events to ensembles. In *Proceedings of the International Parallel Distributed Processing Symposium (IPDPS'10)*, 2010.

[60] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proceedings of the 8th ACM SIGPLAN symposium on Principles and Practices of Parallel Programming (PPoPP'01)*, 2001.

[61] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. Mclarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the IEEE 21th Symposium on Mass Storage Systems and Technologies (MSST'04)*, 2004.

[62] W. W.-S. Wei. *Time series analysis*. Addison-Wesley Redwood City, California, 1994.

[63] Y. Xu, J. B. Weaver, D. M. Healy, and J. Lu. Wavelet transform domain filters: a spatially selective noise filtration technique. *IEEE Transactions on Image Processing*, 3(6):747–758, 1994.

[64] N. J. Yadwadkar, C. Bhattacharyya, K. Gopinath, T. Niranjan, and S. Susarla. Discovery of application workloads from network file traces. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, 2010.

[65] J.-C. Yoo and T. H. Han. Fast normalized cross-correlation. *Circuits, Systems and Signal Processing*, 28(6):819–843, 2009.

[66] J. W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[67] D. Yu, X. Yu, Q. Hu, J. Liu, and A. Wu. Dynamic time warping constraint learning for large margin nearest neighbor classification. *Information Sciences*, 181(13):2787–2796, 2011.

Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation

Hui Wang
Rice University

Peter Varman
Rice University

Abstract

Multi-tiered storage made up of heterogeneous devices are raising new challenges in allocating throughput fairly among concurrent clients. The fundamental problem is finding an appropriate balance between fairness to the clients and maximizing system utilization.

In this paper we cast the problem within the broader framework of fair allocation for multiple resources. We present a new allocation model BAA based on the notion of per-device bottleneck sets. Clients bottlenecked on the same device receive throughputs in proportion to their fair shares, while allocation ratios between clients in different bottleneck sets are chosen to maximize system utilization. We show formally that BAA satisfies fairness properties of Envy Freedom and Sharing Incentive. We evaluated the performance of our method using both simulation and implementation on a Linux platform. The experimental results show that our method can provide both high efficiency and fairness.

1 Introduction

The growing popularity of virtualized data centers hosted on shared physical resources has raised the importance of resource allocation issues in such environments. In addition, the widespread adoption of multi-tiered storage systems [2, 3], made up of solid-state drives (SSDs) and traditional hard disks (HDs), has made the already challenging problem of providing QoS and fair resource allocation considerably more difficult.

Multi-tiered storage has several advantages over traditional flat storage in the data center: improved performance for data access and potential operating cost reductions. However, this architecture also raises many challenges for providing performance isolation and QoS guarantees. The large speed gap between SSDs and HDs means that it is not viable to simply treat the storage system as a black box with a certain aggregate IOPS capacity. The system throughput is intrinsically linked to the relative frequencies with which applications access the different types of devices. In addition, the throughput depends on how the device capacities are actually divvied up among the applications. System efficiency is a major concern for data center operators since consolidation ratios are intimately connected to their competitive advantage. The operator also needs to ensure fairness, so that the increased system utilization is not obtained at the cost of treating some users unfairly.

This brings to focus a fundamental tension between fairness and resource utilization in a system with heterogeneous resources. Maintaining high overall system utilization may require favoring some clients disproportionately while starving some others, thereby compromising fairness. Conversely, allocations based on a rigid notion of fairness can result in reduced system utilization as some clients are unnecessarily throttled to maintain parity in client allocations.

The most-widely used concept of fairness is proportional sharing (PS), which provides allocations to clients in proportion to client-specific weights reflecting their priority or importance. Adaptations of the classic algorithms for network bandwidth multiplexing [49, 9, 40] have been proposed for providing proportional fairness for storage systems [48, 45, 21]. Extended proportional-share schedulers which provide reservations and limit guarantees in addition to proportional allocation have also been proposed for storage systems [46, 19, 22, 23]. However, the vast majority of resource allocation schemes have been designed to multiplex a *single resource*, and have no natural extension to divide up multiple resources.

The question of fair division of multiple resources in computer systems was raised in a fundamental paper by Ghodsi et al [17], who advocated a model called Dominant Resource Fairness (DRF) to guide the allocation (see Section 2). A number of related allocation approaches [16, 36, 11, 25, 28, 15, 14, 13, 12, 30, 38] have

since been proposed; these will be discussed in Section 6. These models deal mainly with defining the meaning of fairness in a multi-resource context. For example, DRF and its extensions consider fairness in terms of a client's dominant resource: the resource most heavily used (as a fraction of its capacity) by a client. The DRF policy is to equalize the shares of each client's dominant resource. In [12], fairness is defined in terms of proportional sharing of the empirically-measured global system bottleneck. A theoretical framework called Bottleneck-based fairness [11] proves constructively the existence of an allocation giving each client its entitlement on some global system-wide bottleneck resource. While these models and algorithms make significant advances to the problem of defining multi-resource fairness, they do not deal with the dual problem of their effect on system utilization. In general these solutions tend to over constrain the system with fairness requirements, resulting in allocations with low system utilization.

In this paper we propose a model called Bottleneck Aware Allocation (BAA) based on the notion of *local bottleneck sets*. We present a new allocation policy to maximize system utilization while providing fairness in the allocations of the competing clients. The allocations of BAA enjoy all of the fairness properties of DRF [17], like Sharing Incentive, Envy Freedom [26, 7, 6], and Pareto Optimality. However, within this space of "fair" solutions that includes DRF, it searches for alternative solutions with higher system efficiency. We prove formally that BAA satisfies these fairness properties. We use BAA as part of a two-tier allocate and schedule mechanism: one module uses a standard weighted fair-scheduler to dispatch requests to the storage system; the other module monitors the workload characteristics and dynamically recomputes the weights using BAA for use by the dispatcher, based on the mix of workloads and their access characteristics. We evaluate the performance of our method using simulation and a Linux platform. The results show that our method can provide both high efficiency and fairness for heterogeneous storage and dynamic workloads.

The rest of the paper is organized as follows. In Section 2 we discuss the difficulties of achieving both fairness and efficiency in a heterogeneous storage system. In Section 3 we describe our model and approach to balance needs of fairness and system efficiency. Formal proofs are presented in Section 4. We present some empirical results in Section 5. Related work is summarized in Section 6, and we conclude in Section 7.

2 Overview

The storage system is composed of SSDs and HD arrays, as shown in Figure 1. SSDs and HDs are independent

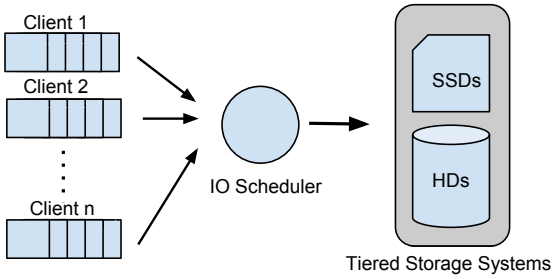


Figure 1: Tiered storage model

devices without frequent data migrations between them. A client makes a sequence of IO requests; the target of each request is either the SSD or the HD, and is known to the scheduler. An access to the SSD is referred to as a *hit* and access to the HD is a *miss*. The hit (miss) ratio of client i is the fraction of its IO requests to the SSD (HD), and is denoted by h_i (respectively m_i). The hit ratio of different applications will generally be different. It may also change in different application phases, but is assumed to be relatively stable within an application phase.

The requests of different clients are held in client-specific queues from where they are dispatched to the storage array by an IO scheduler. The storage array keeps a fixed number of outstanding requests in its internal queues to maximize its internal concurrency. The IO scheduler is aware of the target device (SSD or HD) of a request. Its central component is a module to dynamically assign *weights* to clients based on the measured miss ratios. These weights are used by the dispatcher to choose the order of requests to send to the array; the number of serviced requests of a client is in proportion to its weight. The weights are computed in accordance with the fairness policies of BAA so as to maximize system utilization based on recently measured miss ratios.

We illustrate the difficulties in achieving these goals in the next section, followed by a description of our approach in Section 3.

2.1 Motivating Examples

In traditional proportional fairness a single resource is divided among multiple clients in the ratio of their assigned weights. For instance, if a single disk of 100 IOPS capacity is shared among two backlogged clients of equal weight, then each client will receive 50 IOPS. A work-conserving scheduler like Weighted Fair Queuing [9] will provide fine-grained, weight-proportional bandwidth allocation to the backlogged clients; the system will be 100% utilized as long as there are requests in the system. When the IOs are from heterogeneous

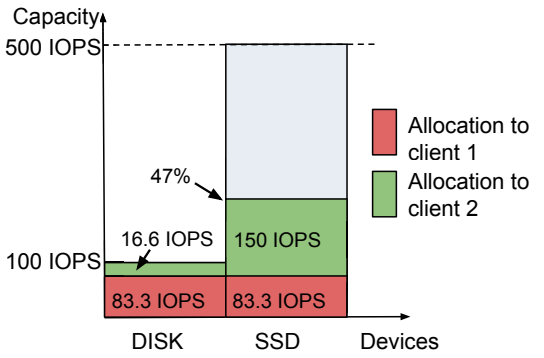
devices like HDs and SSDs, the situation is considerably more complicated. The device load is determined by both the allocation ratios (the relative fraction of the bandwidth assigned to clients), as well as their hit ratios. If the clients with high SSD loads have small allocation ratio, there may be insufficient requests to keep the SSD busy. Conversely, maintaining high utilization of both the HD and the SSD may require adjusting the allocations in a way that starves some clients, resulting in unfairness in the allocation.

Example I Suppose the HD and SSD have capacities of 100 IOPS and 500 IOPS respectively. The system is shared by backlogged clients 1 and 2 with equal weights and hit ratios of $h_1 = 0.5$ and $h_2 = 0.9$ respectively. Under proportional-share allocation, both clients should get an equal number of IOPS. The *unique* allocation in this case is for each client to get 166.6 IOPS. Client 1 will get 83.3 IOPS from each device, while client 2 will get 16.6 IOPS from the HD and 150 IOPS from the SSD. The HD has 100% utilization but the SSD is only 47% utilized (Figure 2(a)). In order to increase the system utilization, the relative allocation of the clients needs to be changed (Figure 2(b)). In fact, both devices can be fully utilized if the scheduler allocates 100 IOPS to client 1 (50 IOPS from the HD and 50 from the SSD), and 500 IOPS to client 2 (50 from the HD and 450 from the SSD). This is again the *unique* allocation that maximizes the system utilization for the given set of hit ratios. Note that increasing the utilization to 100% requires a $1:5$ allocation ratio, and reduces client 1's throughput from 167 to 100 IOPS while increasing client 2's throughput from 167 to 500 IOPS.

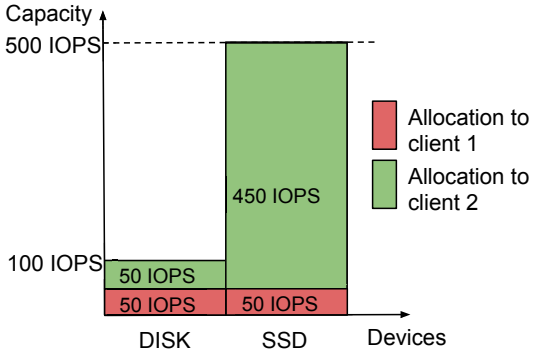
The example above illustrates a general principle. For a given set of workload hit ratios, it is not possible to precisely dictate both the relative allocations (*fairness*) and the system utilization (*efficiency*). How then should we define the allocations to both provide fairness and achieve good system utilization?

Consider next how the DRF policy will allocate the bandwidth. The dominant resource for client 1 is the HD; for client 2 it is the SSD. Suppose DRF allocates n IOPS to client 1 and m IOPS to client 2. Equalizing the dominant shares means that $0.5n/100 = 0.9m/500$ or $n:m = 9:25$. This results in an SSD utilization of approximately 77% (Figure 2(c)). The point to be noted is that none of these earlier approaches considers the issue of resource efficiency when deciding on an allocation. The policies deal with the question of how to set client allocation ratios to achieve some measure of fairness, but do not address how the choice affects system utilization.

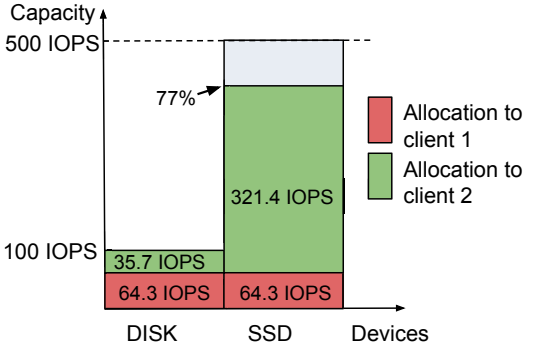
Example II In the example above suppose we add a third backlogged client, also with hit ratio 0.5. In this case, proportional sharing in a $1:1:1$ ratio would result in al-



(a) Allocations in 1:1 ratio. Utilization of the SSD is only 47%



(b) Allocations with 100% utilization. Allocations are in the ratio $1:5$.



(c) Allocations using DRF. Allocations are in the ratio $9:25$. Utilization of the SSD is 77%.

Figure 2: Allocations in multi-tiered storage. HD capacity: 100 IOPS and SSD Capacity: 500 IOPS. The hit ratios of the clients are 0.5 and 0.9 respectively.

locations of 90.9 IOPS each; the client with hit ratio 0.9 would be severely throttled by the allocation ratio, and the SSD utilization will be only 34.5%. If the weights were changed to 1 : 1 : 10 instead, then the HD-bound clients (hit ratio 0.5) would receive 50 IOPS each, and the SSD-bound client (hit ratio 0.9) would receive 500 IOPS. The utilization of both devices is 100%. The DRF policy will result in an allocation ratio of 9 : 9 : 25, allocations of 78, 78 and 217 IOPS respectively, and SSD utilization of 55% (further reduced from the 77% of example 1). This shows that the system utilization is highly dependent on the competing workloads, and stresses how relative allocations (fairness) and system utilization (efficiency) are intertwined.

2.2 Fairness and Efficiency in Multi-tiered Storage

As discussed in the previous section, the ratio of allocation and the system utilization cannot be independently controlled. In [17], DRF allocations are shown to possess desirable fairness properties, namely:

- *Sharing Incentive*: Each client gets at least the throughput it would get from statically partitioning each resource equally among the clients[1]. This throughput will be referred to as the *fair share* of the client. In this paper we will use fair share defined by equal partition of the resources[2].
- *Envy-Freedom*: A client cannot increase its throughput by swapping its allocation with any other client. That is, clients prefer their own allocation over the allocation of any other client.
- *Pareto Efficiency*: A client's throughput cannot be increased without decreasing another's throughput.

We propose a bottleneck-aware allocation policy (BAA), to provide both fairness and high efficiency in multi-tiered storage systems. BAA will preserve the desirable fairness features listed above. However, we add a fundamentally new requirement, namely maximizing the system utilization.

The clients are partitioned into *bottleneck sets* depending on the device on which they have a higher *load*. This is a local property of a client and does not depend on the system bottleneck as in [11, 12].

In our model, clients in the same bottleneck set will receive allocations that are in the ratio of their *fair share*. However, there is *no predefined ratio* between the allocations of clients in different bottleneck sets. Instead, the system is free to set them in a way that maximizes utilization as long as Envy Freedom, Sharing Incentive and Pareto Optimality properties are preserved by the resulting allocation.

[1] [36] extends the definition to weighted clients and weighted partition sizes.

[2] We can apply the framework of [36] to BAA to handle the case of unequal weights as well.

3 Bottleneck-Aware Allocation

In this section, we will discuss our resource allocation model called *Bottleneck-Aware Allocation* (BAA) and the corresponding scheduling algorithm.

3.1 Allocation Model

We begin by precisely defining several terms used in the model. The *capacity* of a device is defined as its throughput (IOPS) when continually busy. As is usually the case, this definition abstracts the fine-grained variations in access times of different types of requests (read or write, sequential or random etc), and uses a representative (e.g. random 4KB reads) or average IOPS number to characterize the device. Denote the capacity (in IOPS) of the disk as C_d and that of the SSD as C_s.

Consider a client i that is receiving a total throughput of T IOPS. This implies it is receiving $T \times h_i$ IOPS from the SSD. The fraction of the capacity of the SSD that it uses is $(T \times h_i)/C_s$. Similarly, the fraction of the capacity of the HD that it uses is $(T \times m_i)/C_d$.

Definitions

1. The *load* of a client i on the SSD is h_i/C_s and on the HD is m_i/C_d. It represents the average fraction of the device capacity that is utilized per IO of a client.

2. Define the system load-balancing point as $h_{bal} = C_s/(C_s + C_d)$. A workload with hit ratio equal to h_{bal} will have equal load on both devices. If the hit ratio of a client is less than or equal to h_{bal} it is said to be *bottlenecked* on the HD; if the hit ratio is higher than h_{bal} it is bottlenecked on the SSD.

3. Partition the clients into two sets D and S based on their *hit ratios*. $D = \{i : h_i \leq h_{bal}\}$ and $S = \{i : h_i > h_{bal}\}$ are the sets of clients that are bottlenecked on the HD and SSD respectively.

4. Define the *fair share* of a client to be the throughput (IOPS) it gets if each of the resources are partitioned equally among all the clients. Denote the fair share of client i by f_i.

5. Let A_i denote the allocation of (total IOPS done by) client i under some resource partitioning. The total throughput of the system is $\sum_i A_i$.

Example III Consider a system with $C_d = 200$ IOPS, $C_s = 1000$ IOPS and four clients p, q, r, s with hit ratios $h_p = 0.75, h_q = 0.5, h_r = 0.90, h_s = 0.95$. In this case,

$h_{bal} = 1000/1200 = 0.83$. Hence, p and q are bottlenecked on the HD, while r and s are bottlenecked on the SSD: $D = \{p,q\}$ and $S = \{r,s\}$.

Suppose the resources are divided equally among the clients, so that each client sees a virtual disk of 50 IOPS and a virtual SSD of 250 IOPS. What are the throughputs of the clients with this static resource partitioning?

Since p and q are HD-bottlenecked, they would use their entire HD allocation of 50 IOPS, and an additional amount on the SSD depending on the hit ratios. Since p's hit ratio is $3/4$, it would get 150 IOPS on the SSD for a total of 200 IOPS, while q ($h_q = 0.5$) would get 50 SSD IOPS for a total of 100 IOPS. Thus the *fair share*s of p and q are 200 and 100 IOPS respectively. In a similar manner, r and s would completely use their SSD allocation of 250 IOPS and an additional amount on the disk. The *fair share*s of r and s in this example are 277.8 and 263.2 IOPS respectively.

Our *fairness policy* is specified by the rules below. The rules (1) and (2) state that the allocations between any two clients that are bottlenecked on the same device are in *proportion to their fair share*. Condition (3) states that clients backlogged on different devices should be envy free. The condition asserts that if client A receives a higher throughput on some device than client B it must get an equal or lesser throughput on the other. We will show in Section 4 that with just rules (1) and (2), the envy-free property is satisfied between any pair of clients that belong both in D or both in S. However, envy-freedom between clients in different sets is explicitly enforced by the third constraint.

Fairness Policy

1. **Fairness between clients in D**:
   $\forall i,j \in D,\ \frac{A_i}{A_j} = \frac{f_i}{f_j}$. Define $\rho_d = \frac{A_i}{f_i}$ to be the ratio of the allocation of client i to its fair share, $i \in D$.
2. **Fairness between clients in S**:
   $\forall i,j \in S,\ \frac{A_i}{A_j} = \frac{f_i}{f_j}$. Define $\rho_s = \frac{A_j}{f_j}$ to be the ratio of the allocation of client j to its fair share, $j \in S$.
3. **Fairness between a client in D and a client in S**:
   $\forall i \in D, j \in S$: $\frac{h_j}{h_i} \geq \frac{A_i}{A_j} \geq \frac{m_j}{m_i}$. Note that if $h_i = 0$ then only the constraint $\frac{A_i}{A_j} \geq \frac{m_j}{m_i}$ is needed.

Example IV What do the fairness policy constraints mean for the system of Example III? Rule 1 means that HD-bound clients p and q should receive allocations in the ratio 2 : 1 (ratio of their fair shares), *i.e.* $A_p/A_q = 2$. Similarly, rule 2 means that SSD-bound clients r and s should receive allocations in the ratio 277.8 : 263.2 = 1.06 : 1, *i.e.* $A_r/A_s = 1.06$. Rule 3 implies a constraint for each of the pairs of clients backlogged on different devices: (p, r), (p, s), (q, r) and (q, s):

i $h_r/h_p = 1.2 \geq A_p/A_r \geq m_r/m_p = 0.4$

ii $h_s/h_p = 1.27 \geq A_p/A_s \geq m_s/m_p = 0.2$

iii $h_r/h_q = 1.8 \geq A_q/A_r \geq m_r/m_q = 0.2$

iv $h_s/h_q = 1.9 \geq A_q/A_s \geq m_s/m_q = 0.1$

These linear constraints will be included in a linear programming optimization model in the next section.

3.2 Optimization Model Formulation

The aim of the resource allocator is to find a suitable allocation A_i for each of the clients. The allocator will maximize the system utilization while satisfying the fairness constraints described in Section 3.1, together with constraints based on the capacity of the HD and the SSD. A direct linear programming (LP) formulation will result in an optimization problem with n unknowns representing the allocations of the n clients, and $O(n^2)$ constraints specifying the rules of the fairness policy.

The search space can be drastically reduced using the auxiliary variables ρ_d and ρ_s (called *amplification factors*) defined in Section 3.1. Rules 1 and 2 require that $A_i = \rho_d f_i$ and $A_j = \rho_s f_j$, for clients $i \in D$ and $j \in S$.

We now formulate the objective function and constraints in terms of the auxiliary quantities ρ_d and ρ_s. The total allocation is:

$$\sum_{\forall k} A_k = (\sum_{i \in D} A_i + \sum_{j \in S} A_j) = (\rho_d \sum_{i \in D} f_i + \rho_s \sum_{j \in S} f_j).$$

The total number of IOPS made to the HD is:

$$\rho_d \sum_{i \in D} f_i m_i \ + \ \rho_s \sum_{j \in S} f_j m_j.$$

The total number of IOPS made to the SSD is:

$$\rho_d \sum_{i \in D} f_i h_i + \rho_s \sum_{j \in S} f_j h_j.$$

Fairness rule 3 states that: $\forall i \in D, j \in S$,

$$\frac{h_j}{h_i} \geq \frac{\rho_d f_i}{\rho_s f_j} \geq \frac{m_j}{m_i}$$

$$\frac{h_j f_j}{h_i f_i} \geq \frac{\rho_d}{\rho_s} \geq \frac{m_j f_j}{m_i f_i}.$$

$$\beta \geq \frac{\rho_d}{\rho_s} \geq \alpha.$$

where

$$\alpha = max_{i,j} \left\{ \frac{m_j f_j}{m_i f_i} \right\} \qquad \beta = min_{i,j} \left\{ \frac{h_j f_j}{h_i f_i} \right\}$$

The final problem formulation is shown below. It is expressed as a 2-variable linear program with unknowns

ρ_d and ρ_s, and four linear constraints between them. Equations 2 and 3 ensure that the total throughputs from the HD and the SSD respectively do not exceed their capacities. Equation 4 ensures that any pair of clients, which are bottlenecked on the HD and SSD respectively, are envy free. As mentioned earlier, we will show that clients which are bottlenecked on the same device will automatically be envy free.

Optimization for Allocation

$$\textbf{Maximize} \quad \rho_d \sum_{i \in D} f_i \; + \; \rho_s \sum_{j \in S} f_j \tag{1}$$

subject to:

$$\rho_d \sum_{i \in D} f_i m_i \; + \; \rho_s \sum_{j \in S} f_j m_j \leq C_d \tag{2}$$

$$\rho_d \sum_{i \in D} f_i h_i \; + \; \rho_s \sum_{j \in S} f_j h_j \leq C_s \tag{3}$$

$$\beta \geq \frac{\rho_d}{\rho_s} \geq \alpha \tag{4}$$

Example V We show the steps of the optimization for the scenario of Example III. $D = \{p,q\}$, $S = \{r,s\}$, and the fair shares $f_p = 200$, $f_q = 100$, $f_r = 277.8$ and $f_s = 263.2$. $\sum_{i \in D} f_i = 200 + 100 = 300$, $\sum_{j \in S} f_j = 277.8 + 263.2 = 541$, $\sum_{i \in D} f_i m_i = 50 + 50 = 100$, $\sum_{j \in S} f_j m_j = 27.78 + 13.2 = 41$, $\sum_{i \in D} f_i h_i = 150 + 50 = 200$, and $\sum_{j \in S} f_j h_j = 250 + 250 = 500$. Also it can be verified that $\alpha = 0.55$ and $\beta = 1.67$. Hence, we get the following optimization problem:

$$\textbf{Maximize}: \quad 300\rho_d \; + \; 541\rho_s \tag{5}$$

subject to:

$$100\rho_d \; + \; 41\rho_s \; \leq \; 200 \tag{6}$$

$$200\rho_d + \; 500\rho_s \; \leq \; 1000 \tag{7}$$

$$1.67 \geq \frac{\rho_d}{\rho_s} \geq 0.55 \tag{8}$$

Solving the linear program gives $\rho_d = 1.41$, $\rho_s = 1.44$, which result in allocations $A_p = 282.5$, $A_q = 141.3$, $A_r = 398.6$, $A_s = 377.6$, and HD and SSD utilizations of 100% and 100%.

We end the section by stating precisely the properties of BAA with respect to fairness and utilization. The properties are proved in Section 4.

- **P1**: *Clients in the same bottleneck set receive allocations proportional to their fair shares.*
- **P2**: *Any pair of clients bottlenecked on the same device will not envy each other. Combined with fairness policy (3) which enforces envy freedom between clients bottlenecked on different devices, we can assert that the allocations are envy free.*
- **P3**: *Every client will receive at least its fair share. In other words, no client receives less throughput than it would if the resources had been hard-partitioned equally among them. Usually, clients will receive more than their fair share by using capacity on the other device that would be otherwise unused.*
- **P4**: *The allocation maximizes the system throughput subject to these fairness criteria.*

3.3 Scheduling Framework

The LP described in Section 3.2 calculates the throughput that each client is allocated based on the mix of hit ratios and the system capacities. The ratios of these allocations make up the weights to a proportional-share scheduler like WFQ [9], which dispatches requests from the client queues.

When a new client enters or leaves the system, the allocations (*i.e.* the weights to the proportional scheduler) need to be updated. Similarly, if a change in a workload's characteristics results in a significant change in its hit ratio, the allocations should be recomputed to prevent the system utilization from falling too low. Hence, periodically (or triggered by an alarm based on device utilizations) the allocation algorithm is invoked to compute the new set of weights for the proportional scheduler. We also include a module to monitor the hit ratios of the clients over a moving window of requests. The hit ratio statistics are used by the allocation algorithm.

Algorithm 1: Bottleneck-Aware Scheduling

Step 1. For each client maintain statistics of its hit ratio over a configurable request-window W.

Step 2. Periodically invoke the BAA optimizer of Section 3.2 to compute the allocation of each client that maximizes utilization subject to fairness constraints.

Step 3. Use the allocations computed in Step 2 as relative weights to a proportional-share scheduler that dispatches requests to the array in the ratio of their weights.

The allocation algorithm is relatively fast since it requires solving only a small 2-variable LP problem, so it can be run quite frequently. Nonetheless, it would be desirable to have a single-level scheme in which the

be desirable to have a single-level scheme in which the scheduler continually adapts to the workload characteristics rather than at discrete steps. In future work we will investigate the possibility of such a single-level scheme.

4 Formal Results

In this section we formally establish the fairness claims of BAA. The two main properties are summarized in Lemma 3 and Lemma 7, which state that the allocations made by BAA are envy free (EF) and satisfy the sharing incentive (SI) property. Table 1 summarizes the meanings of different symbols.

Symbol	Meaning
C_s, C_d	Capacity in IOPS of SSD (HD)
S, D	Set of clients bottlenecked on the SSD (HD)
ρ_s, ρ_d	Proportionality constants of *fairness policy*
f_i	Fair Share for client i
h_i, m_i	Hit (Miss) ratio for client i
h_{bal}	Load Balance Hit Ratio: $C_s/(C_s+C_d)$
n	Total number of clients

Table 1: List of Symbols

Lemma 1 finds expressions for fair shares. The fair share of a client is its throughput if it is given a virtual HD of capacity C_d/n and a virtual SSD of capacity C_s/n. A client in D will use all the capacity of the virtual HD, and hence have a fair share of $C_d/(n \times m_i)$. A client in S uses all the capacity of the virtual SSD, and its fair share is $C_s/(n \times h_i)$.

Lemma 1. *Let n be the number of clients. Then $f_i = min\{C_d/(n \times m_i), C_s/(n \times h_i)\}$. If $i \in D$, then $f_i = C_d/(n \times m_i)$; else if $i \in S$, then $f_i = C_s/(n \times h_i)$.*

Proof. The fair share is the total throughput when a client uses one of its virtual resources completely. For $i \in D$, $h_i \leq h_{bal} = C_s/(C_s+C_d)$ and $m_i \geq 1 - h_{bal} = C_d/(C_s+C_d)$. In this case, $C_d/(n \times m_i) \leq (C_s+C_d)/n$ and $C_s/(n \times h_i) \geq (C_s+C_d)/n$. Hence, the first term is the smaller one, whence the result follows. A similar argument holds for $i \in S$. □

Lemma 2 states a basic property of BAA allocations: all clients in a bottleneck set receive equal throughputs on the device on which they are bottlenecked. This is simply a consequence of *fairness policy* which requires that clients in the same bottleneck set receive throughput in the ratio of their fair shares.

Lemma 2. *All clients in a bottleneck set receive equal throughputs on the bottleneck device. Specifically, all clients in D receive $\rho_d C_d/n$ IOPS from the HD; and all clients in S receive $\rho_s C_s/n$ IOPS from the SSD.*

Proof. Let $i \in D$. From *fairness policy* (1) and lemma 1, $A_i = \rho_d f_i = \rho_d(C_d/(n \times m_i))$. The number of IOPS from the HD is therefore $A_i m_i = \rho_d C_d/n$. Similarly, for $i \in S$, $A_i = \rho_s f_i = \rho_s(C_s/(n \times h_i))$, and the number of IOPS from the SSD is $A_i h_i = \rho_s C_s/n$. □

To prove EF between two clients, we need to show that no client receives more throughput on *both* the resources (HD and SSD). If the two clients are in the same bottleneck set then this follows from Lemma 2, which states that both clients will get *equal* throughputs on their bottleneck device. When the clients are in different bottleneck sets then the condition is explicitly enforced by *fairness policy* (3).

Lemma 3. *For any pair of client i, j the allocations made by BAA are envy free.*

Proof. From lemma 2, if $i, j \in D$ both clients have the same number of IOPS on the HD; hence neither can improve its throughput by getting the others allocation. Similarly, if $i, j \in S$ they do not envy each other, since nether can increases its throughput by receiving the others allocation.

Finally, we consider the case when $i \in D$ and $j \in S$. From *fairness policy* (3), $\forall i \in D, j \in S$: $\frac{h_j}{h_i} \geq \frac{A_i}{A_j} \geq \frac{m_j}{m_i}$. Hence, the allocations on the SSD for clients i and j satisfy $A_i h_i \leq A_j h_j$, and the allocations on HD for clients i and j satisfy $A_i m_i \geq A_j m_j$. So any two flows in different bottleneck sets will not envy each other. Hence neither i nor j can get more than the other on both devices. □

The following Lemma shows the Sharing Incentive property holds in the "simple" case. The more difficult case is shown in Lemma 6. Informally, if the HD is a *system* bottleneck (*i.e.*, it is 100% utilized) then Lemma 4 shows that the clients in D will receive at least $1/n$ of the HD bandwidth. The clients in S may get less than that amount on the HD (and usually will get less). Similarly, if the SSD is a system bottleneck, then the clients in S will receive at least $1/n$ of the SSD bandwidth.

In the remainder of this section we assume that the clients $1, 2, \cdots n$, are ordered in non-decreasing order of their hit ratios, and that r of them are in D and the rest in S. Hence, $D = \{1, \cdots, r\}$ and $S = \{r+1, \cdots, n\}$.

Lemma 4. *Suppose the HD (SSD) has a utilization of 100%. Then every $i \in D$ (respectively $i \in S$) receives a throughput of at least f_i.*

Proof. Let j denote an arbitrary client in S. From *fairness policy* (3), $A_i m_i \geq A_j m_j$. That is, the throughput on the HD of a client in D is greater than or equal to the throughput on the HD of any client in S. Now, from lemma 2 the IOPS from the HD of all $i \in D$ are equal. Since, by hypothesis, the disk is 100% utilized, the total

IOPS from the HD is C_d. Hence, for every $i \in D$, the IOPS on the disk must be at least C_d/n. A symmetrical proof holds for clients in S. □

In order to show the Sharing Incentive property for clients whose bottleneck device is not the system bottleneck (*i.e.* is less than 100% utilized), we prove the following Lemma. Informally, it states that utilization of the SSD improves if the clients in S can be given a bigger allocation. The result, while intuitive, is not self evident. An increase in the SSD allocation to a client in S increases its HD usage as well. Since the HD is 100% utilized, this reduces HD allocations of clients in D, which in turn reduces their allocation on the SSD. We need to check that the net effect is positive in terms of SSD utilization.

Lemma 5. *Consider two allocations that satisfy fairness policy (1) - (3), and for which the HD has utilization of* 100% *and the SSD has utilization less than*100%. *Let* ρ_s *and* $\hat{\rho}_s$ *be the proportionally constants of clients in S for the two allocations, and let U and* $\hat{U}$ *be the respective system throughputs. If* $\hat{\rho}_s > \rho_s$ *then* $\hat{U} > U$. *A symmetrical result holds if the SSD is* 100% *utilized and the HD is less than* 100% *utilized.*

Proof. We show the case for HD 100% utilized. From Lemma 2, all clients in S have the same throughput $\rho_s C_s/n$ on the SSD. Define δ_s to be the difference between the SSD throughputs of a client in S in the two allocations. Since $\hat{\rho}_s > \rho_s$, $\delta_s > 0$. Similarly, define δ_d to be difference between the HD throughput of a client in D in the two allocations.

An increase of δ_s in the throughput of client $i \in S$ on the SSD implies an increase on the HD of $\delta_s \times (m_i/h_i)$. Since the HD is 100% utilized in both allocations, the aggregate allocations of clients in D must decrease by the total amount $\sum_{i\in S} \delta_s \times (m_i/h_i)$. By Lemma 2, since all clients in D have the same allocation on the HD, $\delta_d = \sum_{i\in S} \delta_s \times (m_i/h_i)/|D|$. As a result, the decrease in the allocation of client $j \in D$ on the SSD is $\hat{\delta}_s = \delta_d \times (h_j/m_j)$.

The total change in the allocation on the SSD in the two allocations, Δ is therefore: $\Delta = \sum_{i\in S} \delta_s - \sum_{j\in D} \hat{\delta}_s$. Substituting:

$$\Delta = \sum_{i\in S} \delta_s - \sum_{j\in D} \delta_d \times (h_j/m_j) \tag{9}$$

$$\Delta = |S| \times \delta_s - \sum_{j\in D} (\sum_{i\in S} \delta_s \times (m_i/h_i)/|D|) \times (h_j/m_j) \tag{10}$$

Now for all $i \in S$, $(m_i/h_i) \le (m_{r+1}/h_{r+1})$ and for all $j \in D$, $(h_j/m_j) \le (h_r/m_r)$.

Substituting in Equation 10:

$$\Delta \ge |S| \times \delta_s - |S|\delta_s \times (m_{r+1}/h_{r+1}) \times (h_r/m_r) \tag{11}$$

$$\Delta \ge |S| \times \delta_s (1 - \frac{m_{r+1}}{m_r} \times \frac{h_r}{h_{r+1}}) \tag{12}$$

Now, $m_{r+1} < m_r$ and $h_r < h_{r+1}$ since r and $r+1$ are in D and S respectively. Hence, $\Delta > 0$.

□

Finally, we show the Sharing Incentive property for clients whose bottleneck device is not the system bottleneck. The idea is to make the allocation to the clients in S as large as we can, before the EF requirements prevent further increase.

Lemma 6. *Suppose the HD (SSD) has utilization of* 100% *and the SSD (HD) has utilization less than* 100%. *Then every* $i \in S$ (*respectively* $i \in D$) *receives a throughput of at least* f_i.

Proof. We will show it for clients in S. A symmetrical proof holds in the other case.

Since BAA maximizes utilization subject to *fairness policy* (1) - (3), it follows from Lemma 5 that ρ_s must be as large as possible. If $i \in S$, the IOPS it receives on the HD are $\rho_s C_s/n \times (m_i/h_i)$ which from the EF requirements of Lemma 3 must be no more than $\rho_d C_d/n$, the IOPS on the HD for any client in D. Hence, $\rho_s C_s/n \times (m_i/h_i) \le \rho_d C_d/n$ or $\rho_s \le \rho_d (C_d/C_s) \times (h_i/m_i)$, for all $i \in S$. Since h_i/m_i is smallest for $i = r+1$, the maximum feasible value of ρ_s is $\rho_s = \rho_d (C_d/C_s) \times (h_{r+1}/m_{r+1})$. Now, $h_{r+1} > h_{bal}$, so $h_{r+1}/m_{r+1} > h_{bal}/(1-h_{bal}) = C_s/C_d$. Hence $\rho_s > \rho_d$. Since the HD is 100% utilized we know from Lemma 4 that $\rho_d \ge 1$, and so $\rho_s > 1$. □

From Lemmas 4 to 6 we can conclude:

Lemma 7. *Allocations made by BAA satisfy the Sharing Incentive property.*

5 Performance Evaluation

We evaluate our work using both simulation and Linux system implementation. For simulation, a synthetic set of workloads was created. Each request is randomly assigned to the SSD or HD based on its hit ratio. The request service time is an exponentially distributed random variable with mean equal to the reciprocal of the device IOPS capacity.

In the Linux system, we implemented a prototype by interposing the BAA scheduler in the IO path. Raw IO is performed to eliminate the influence of OS buffer caching. The storage server includes a 1TB SCSI Western Digital hard disk (7200 RPM 64MB Cache SATA 6.0Gb/s) and 120GB SAMSUNG 840 Pro Series SSD. Various block-level workloads from UMass Trace Repository [1] and Microsoft Exchange server [31] are

used for the evaluation. These traces are for a homogeneous server and do not distinguish between devices. Since we needed to emulate different proportions of HD and SSD requests we randomly partitioned the blocks between the two devices to meet the assumed hit ratio of the workload. The device utilizations are measured using Linux tool "iostat".

5.1 Simulation Experiments

5.1.1 System Efficiency

This experiment compares the system efficiency for three different schedulers: Fair Queuing (FQ), DRF, and BAA. The capacities of the HD and SSD are 100 IOPS and 5000 IOPS respectively. The first experiment employs two clients with hit ratios 0.5 and 0.99. FQ allocates equal amounts of throughput to the two clients. The DRF implementation uses the dominant resource shares policy of [17] to determine allocation weights, and BAA is the approach proposed in this paper. All workloads are assumed to be continuously backlogged.

The throughputs of the two clients with different schedulers are shown in Figure 3(a). The figure also shows the fair share allocation, *i.e.* the throughput the workload would get by partitioning the SSD and HD capacities equally between the two workloads. As can be seen, the throughput of client 2 under FQ is the lowest of the three schedulers. In fact, sharing is a disincentive for client 2 under FQ scheduling, since it would have been better off with a static partitioning of both devices. The problem is that the fair scheduler severely throttles the SSD-bound workload to force the 1 : 1 fairness ratio. DRF performs much better than FQ. Both clients get a little more than their fair shares. BAA does extremely well in this setup and client 2 is able to almost double the throughput it would have received with a static partition. We also show the system utilization for the three schedulers in Figure 3(b). BAA is able to fully utilize both devices, while DRF reaches system utilization of only around 65%.

Next we add another client with hit ratio of 0.8 to the workload mix. The throughputs of the clients are shown in Figure 4(a). Now the throughput of the DRF scheduler is also degraded, because it does not adjust the relative allocations to account for load imbalance. The BAA scheduler gets higher throughput (but less than 100%) because it adjusts the weights to balance the system load. The envy-free requirements put an upper-bound on the SSD-bound client's throughput, preventing the utilization from going any higher, but still maintaining fairness.

5.1.2 Adaptivity to Hit Ratio Changes

In this experiment, we show how the two-level scheduling framework restores system utilization following a change in an application's hit ratio. The capacities of the HD and SSD are 200 IOPS and 3000 IOPS respectively. In this simulation, allocations are recomputed every 100s and the hit ratio is monitored in a moving window of 60s. There are two clients with initial hit ratios of 0.45 and 0.95. At time 510s, the hit ratio of client 1 falls to 0.2.

Figure 5 shows a time plot of the throughputs of the clients. The throughputs of both clients falls significantly at time 510 as shown in Figure 5. The scheduler needs to be cognizant of changes in the application characteristics and recalibrate the allocations to increase the efficiency. At time 600s (the rescheduling interval boundary) the allocations are recomputed using the hit ratios that reflect the current application behavior, and the system throughput rises again.

In practice the frequency of calibration and the rate at which the workload hit ratios change can affect system performance and stability. As is the case in most adaptive situations, the techniques work best when significant changes in workload characteristics do not occur at a very fine time scale. We leave the detailed evaluation of robustness to future work.

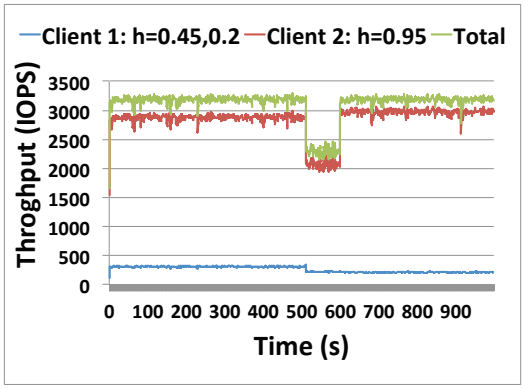
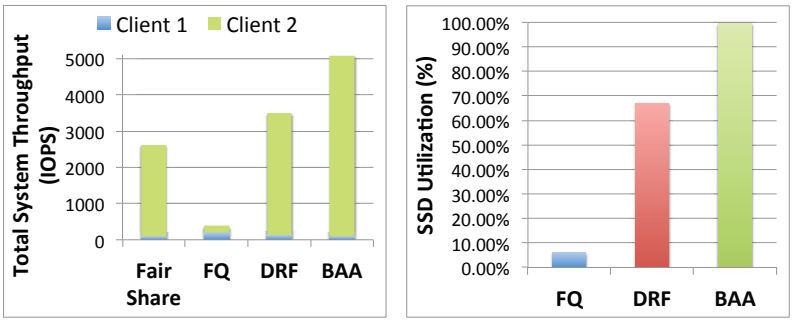


Figure 5: Scheduling with dynamic weights when hit ratio changes

5.2 Linux Experiments

We now evaluate BAA in a Linux system, and compare its behavior with allocations computed using the DRF policy [17] and the Linux CFQ [39] scheduler. The first set of experiments deals with evaluating the throughputs (or system utilization) of the three scheduling approaches. The second set compares the fairness properties.



(a) Throughputs (b) Utilizations

Figure 3: Throughputs and utilizations for 2 flows

(a) Throughputs (b) Utilizations

Figure 4: Throughputs and utilizations for 3 flows

5.2.1 Throughputs and Device Utilizations

Throughputs	BAA	CFQ	DRF
Client 1	100	101	95
Client 2	139	134	133
Total	239	235	228

Table 2: Throughputs: all clients in one bottleneck set

Clients in the same bottleneck set. Two workloads from Web Search [1] are used in this experiment. The requests include reads and writes and the request sizes range from 8KB to 32KB.

We first evaluate the performance when all the clients fall into the same bottleneck set; that is, all the clients are bottlenecked on the same device. We use hit ratios of 0.3 and 0.5 for the two workloads which makes them both HD bound. As shown in Table 2 all three schedulers get similar allocation. In this situation there is just one local bottleneck set in BAA, which (naturally) coincides with the system bottleneck device for CFQ as well as being the dominant resource for DRF. The device utilizations are the same for all schedulers, as can be expected.

Clients in different bottleneck sets. In this experiment, we evaluate the performance when the clients fall into different bottleneck sets; that is, some of the clients are bottlenecked on the HD and some on the SSD. Two clients, one running a Financial workload [1] (client 1) and the second running an Exchange workload [31] (client 2) with hit ratios of 0.3 and 0.95 respectively, are used in the experiment. The request sizes range from 512 bytes to 8MB, and are a mix of read and write requests. The total experiment time is 10 minutes.

Figure 6 shows the throughput of each client achieved by the three schedulers. As shown in the figure, BAA has better total system throughput than the others. CFQ performs better than DRF but not as good as BAA.

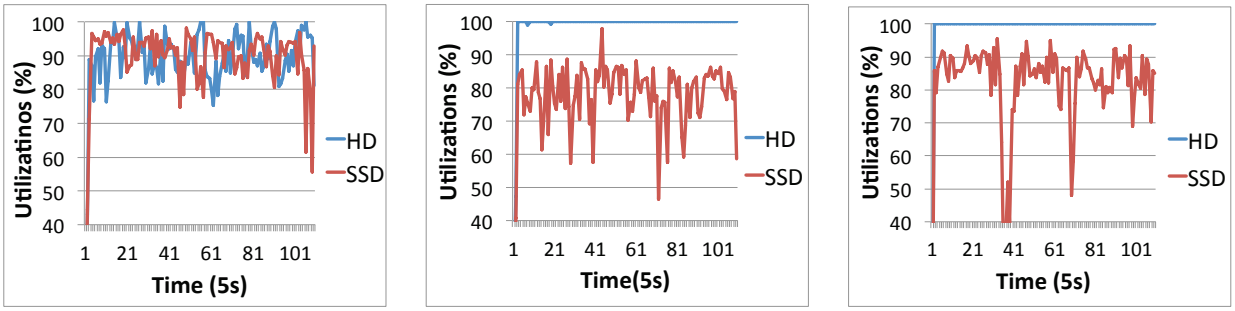
Figure 7 shows the measured utilizations for HD and SSD using the three schedulers. Figure 7(a) shows that BAA achieves high system utilization for both HD and SSD; DRF and CFQ have low SSD utilizations compared with BAA, as shown in Figure 7(b) and (c). HD utilizations are good for both DRF and CFQ (almost 100%), because the system has more disk-bound clients that saturate the disk.

5.2.2 Allocation Properties Evaluation

In this experiment, we evaluate the fairness properties of allocations (**P1** to **P4**). Four Financial workloads [1] with hit ratios of 0.2, 0.4, 0.98 and 1.0 are used as the input. The workloads have a mix of read and write requests and request sizes range from 512 bytes to 8MB.

(a) BAA (b) DRF (c) CFQ

Figure 6: Throughputs using three schedulers. BAA achieves higher system throughput (1396 IOPS) than both DRF-based Allocation (810 IOPS) and Linux CFQ (1011 IOPS).



(a) BAA (b) DRF (c) CFQ

Figure 7: System utilizations using three schedulers. The average utilization are: BAA (HD 94% and SSD 92%), DRF (HD 99% and SSD 78%), CFQ (HD 99.8% and SSD 83%)

Clients	Fair Share (IOPS)	Total IOPS	HD IOPS	SSD IOPS
Financial 1	50	76	60.8	15.2
Financial 2	67	101	60.8	40.4
Financial 3	561	1068	21.4	1047
Financial 4	550	1047	0	1047

Table 3: Allocations for Financial workloads using BAA

Table 3 shows the allocations of BAA-based scheduling. The second column shows the Fair Share for each workload. The third column shows the IOPS achieved by each client, and the portions from the HD and SSD are shown in the next two columns.

The average capacity of the HD for the workload is around 140-160 IOPS and the SSD is 2000-2200 IOPS. We use the upper-bound of the capacity to compute the fair shares shown in the second column. In this setup, Financial 1 and Financial 2 are bottlenecked on the HD and belong to D, while Financial 3 and Financial 4 are bottlenecked on the SSD and belong to S.

First we verify that clients in the same bottleneck set receive allocations in proportion to their fair share (**P1**). As shown in the Table 3, Financial 1 and 2 get throughputs of 76 and 101, which are in the same ratio as their fair share (50 : 67). Similarly, Financial 3 and 4 get throughputs 1068 and 1047, which are in the ratio of their fair share of (561 : 550).

HD-bottlenecked workloads Financial 1 and Financial 2 receive more HD allocation (60.8 IOPS) than both workloads Financial 3 (21.4 IOPS) and 4 (0 IOPS). Similarly, SSD-bottlenecked workloads Financial 3 and Financial 4 receive more SSD allocation (1047 and 1047 IOPS) than both workload 1 (15.2 IOPS) and 2 (40.4 IOPS).

It can be verified from columns 2 and 3 that every client receives at least its fair share. Finally, the system shows that both HD and SSD are almost fully utilized, indicating the allocation maximizes the system throughput subject to these fairness criteria. Similar experiments were also conducted with other workloads, including those from Web Search and Exchange Servers. The results show that properties P1 to P4 are always guaranteed.

6 Related Work

There has been substantial work dealing with proportional share schedulers for networks and CPU [9, 18, 44]. These schemes have since been extended to handle the

constraints and requirements of storage and IO scheduling [21, 19, 20, 45, 32, 27, 33]. Extensions of WFQ to provide reservations for constant capacity servers were presented in [41]. Reservation and limit controls for storage servers were studied in [29, 46, 22, 24]. All these models provide strict proportional allocation for a single resource based on static shares possibly subject to reservation and limit constraints.

As discussed earlier, Ghodsi et al [17] proposed the DRF policy, which provides fair allocation of multiple resources on the basis of dominant shares. Ghodsi et al. [16] extended DRF to packet networks and compared it to the global bottleneck allocation scheme of [12]. Dolev et al [11] proposed an alternative to DRF based on fairly dividing a global system bottleneck resource. Gutman and Nisan [25] considered generalizations of DRF in a more general utility model, and also gave a polynomial time algorithm for the construction in Dolev et al [11]. Parkes et al. [36] extended DRF in several ways, and in particular studied the case of indivisible tasks. Envy-freedom has been studied in the areas of economics [26] and in game theory [10].

Techniques for isolating random and sequential IOs using time-quanta based IO allocation were presented in [37, 34, 42, 43, 39, 8]. IO scheduling for SSDs is examined in [34, 35]. Placement and scheduling tradeoffs for hybrid storage were a studied in [47]. For a multi-tiered storage system, Reward scheduling [13, 14, 15] proposed making allocations in the ratio of the throughputs a client would receive when executed in isolation. Interestingly, both Reward and DRF perform identical allocations for the storage model of this paper [14] (concurrent operation of the SSD and the HD), although they start from very different fairness criteria. Hence, Reward also inherits the fairness properties proved for DRF [17]. For a sequential IO model where only 1 IO is served at a time, Reward will equalize the IO time allocated to each client. Note that neither DRF nor Reward explicitly address the problem of system utilization.

In the system area, Mesos [5] proposes a two-level approach to allocate resources to frameworks like Hadoop and MPI that may share an underlying cluster of servers. Mesos (and related solutions) rely on OS-level abstractions like resource containers [4].

7 Conclusions and Future Work

Multi-tiered storage made up of heterogeneous devices are raising new challenges in providing fair throughput allocation among clients sharing the system. The fundamental problem is finding an appropriate balance between fairness to the clients and increasing system utilization. In this paper we cast the problem within the broader framework of fair allocation for multiple resources, which has been drawing considerable amount of recent research attention. We find that existing methods almost exclusively emphasize the fairness aspect to the possible detriment of system utilization.

We presented a new allocation model BAA based on the notion of per-device bottleneck sets. The model provides clients that are bottlenecked on the same device with allocations that are proportional to their fair shares, while allowing allocation ratios between clients in different bottleneck sets to be set by the allocator to maximize utilization. We show formally that BAA satisfies the properties of Envy Freedom and Sharing Incentive that are well accepted fairness requirements in microeconomics and game theory. Within these fairness constraints BAA finds the best system utilization. We formulated the optimization as a compact 2-variable LP problem. We evaluated the performance of our method using both simulation and implementation on a Linux platform. The experimental results show that our method can provide both high efficiency and fairness.

One avenue of further research is to better understand the theoretical properties of the Linux CFQ scheduler. It performs remarkably well in a wide variety of situations; we feel it is important to better understand its fairness and efficiency tradeoffs within a suitable theoretical framework. We are also investigating single-level scheduling algorithms to implement the BAA policy, and plan to conduct empirical evaluations at larger scale beyond our modest experimental setup.

Our approach also applies, with suitable definitions and interpretation of quantities, to broader multi-resource allocation settings as in [17, 11, 36], including CPU, memory, and network allocations. It can also be generalized to handle client weights; in this case clients in the same bottleneck set receive allocations in proportion to their weighted fair shares. We are also investigating settings in which the SSD is used as a cache; this will involve active data migration between the devices, making the resource allocation problem considerably more complex.

Acknowledgments

We thank the reviewers of the paper for their insightful comments which helped shape the revision. We are grateful to our shepherd Arif Merchant whose advice and guidance helped improve the paper immensely. The support of NSF under Grant CNS 0917157 is greatly appreciated.

References

[1] Storage performance council (umass trace repository), 2007. http://traces.cs.umass.edu/index.php/Storage.

[2] EMC: Fully automate storage tiering. http://www.emc.com/about/glossary/fast.htm, 2012.

[3] Tintri: VM aware storage. http://www.tintri.com, 2012.

[4] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: a new facility for resource management in server systems. In *OSDI '99*.

[5] BENJAMIN, H., AND ET. AL. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI'11*.

[6] BERTSIMAS, D., FARIAS, V. F., AND TRICHAKIS, N. On the efficiency-fairness trade-off. *Manage. Sci. 58*, 12 (Dec. 2012), 2234–2250.

[7] BERTSIMAS, D., FARIAS, V. F., AND TRICHAKIS, V. F. The price of fairness. *Operations Research 59*, 1 (Jan. 2011), 17–31.

[8] BRUNO, J., BRUSTOLONI, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. Disk scheduling with Quality of Service guarantees. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems, Volume 2* (1999), IEEE Computer Society.

[9] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience 1*, 1 (September 1990), 3–26.

[10] DEVANUR, N. R., HARTLINE, J. D., AND YAN, Q. Envy freedom and prior-free mechanism design. *CoRR abs/1212.3741* (2012).

[11] DOLEV, D., FEITELSON, D. G., HALPERN, J. Y., KUPFERMAN, R., AND LINIAL, N. No justified complaints: On fair sharing of multiple resources. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (New York, NY, USA, 2012), ITCS '12, ACM, pp. 68–75.

[12] EGI, N., IANNACCONE, G., MANESH, M., MATHY, L., AND RATNASAMY, S. Improved parallelism and scheduling in multi-core software routers. *The Journal of Supercomputing 63*, 1 (2013), 294–322.

[13] ELNABLY, A., DU, K., AND VARMAN, P. Reward scheduling for QoS scheduling in cloud applications. In *12th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing* (CCGRID'12, May 2012).

[14] ELNABLY, A., AND VARMAN, P. Application specific QoS scheduling in storage servers. In *24th ACM Symposium on Parallel Algorithms and Architectures* (SPAA'12, June 2012).

[15] ELNABLY, A., WANG, H., GULATI, A., AND VARMAN, P. Efficient QoS for multi-tiered storage systems. In *4th USENIX Workshop on Hot Topics in Storage and File Systems* (June 2012).

[16] GHODSI, A., SEKAR, V., ZAHARIA, M., AND STOICA, I. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 1–12.

[17] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 24–24.

[18] GOYAL, P., VIN, H. M., AND CHENG, H. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. Netw. 5*, 5 (1997), 690–704.

[19] GULATI, A., AHMAD, I., AND WALDSPURGER, C. PARDA: Proportional Allocation of Resources in Distributed Storage Access. In *(FAST '09)Proceedings of the Seventh Usenix Conference on File and Storage Technologies* (Feb 2009).

[20] GULATI, A., KUMAR, C., AHMAD, I., AND KUMAR, K. Basil: Automated io load balancing across storage devices. In *Usenix FAST* (2010), pp. 169–182.

[21] GULATI, A., MERCHANT, A., AND VARMAN, P. *p*Clock: An arrival curve based approach for QoS in shared storage systems. In *ACM SIGMETRICS* (2007).

[22] GULATI, A., MERCHANT, A., AND VARMAN, P. *m*Clock: Handling Throughput Variability for Hypervisor IO Scheduling . In *USENIX OSDI* (2010).

[23] GULATI, A., SHANMUGANATHAN, G., ZHANG, X., AND VARMAN, P. Demand based hierarchical qos using storage resource pools. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 1–1.

[24] GULATI, A., SHANMUGANATHAN, G., ZHANG, X., AND VARMAN, P. Demand based hierarchical qos using storage resource pools. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 1–1.

[25] GUTMAN, A., AND NISAN, N. Fair allocation without trade. *CoRR abs/1204.4286* (2012).

[26] JACKSON, M. O., AND KREMER, I. Envy-freeness and implementation in large economies. *Review of Economic Design 11*, 3 (2007), 185–198.

[27] JIN, W., CHASE, J. S., AND KAUR, J. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS '04* (2004).

[28] JOE-WONG, C., SEN, S., LAN, T., AND CHIANG, M. In *INFOCOM* (2012), A. G. Greenberg and K. Sohraby, Eds., IEEE, pp. 1206–1214.

[29] KARLSSON, M., KARAMANOLIS, C., AND ZHU, X. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage 1*, 4 (2005), 457–480.

[30] KASH, I., PROCACCIA, A. D., AND SHAH, N. No agent left behind: Dynamic fair division of multiple resources. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems* (Richland, SC, 2013), AAMAS '13, International Foundation for Autonomous Agents and Multiagent Systems, pp. 351–358.

[31] KAVALANEKAR, S., WORTHINGTON, B., ZHANG, Q., AND SHARDA, V. Characterization of storage workload traces from production windows servers. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on* (2008), pp. 119–128.

[32] LUMB, C., MERCHANT, A., AND ALVAREZ, G. Façade: Virtual storage devices with performance guarantees. *File and Storage technologies (FAST'03)* (March 2003), 131–144.

[33] LUMB, C. R., SCHINDLER, J., GANGER, G. R., NAGLE, D. F., AND RIEDEL, E. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. In *Usenix OSDI* (2000).

[34] PARK, S., AND SHEN, K. Fios: A fair, efficient flash i/o scheduler. In *FAST* (2012).

[35] PARK, S., AND SHEN, K. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *Usenix ATC* (2013).

[36] PARKES, D. C., PROCACCIA, A. D., AND SHAH, N. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. In *Proceedings of the 13th ACM Conference on Electronic Commerce* (New York, NY, USA, 2012), EC '12, ACM, pp. 808–825.

[37] POVZNER, A., KALDEWEY, T., BRANDT, S., GOLDING, R., WONG, T. M., AND MALTZAHN, C. Efficient guaranteed disk request scheduling with Fahrrad. *SIGOPS Oper. Syst. Rev. 42*, 4 (2008), 13–25.

[38] PROCACCIA, A. D. Cake cutting: Not just child's play. *Communications of the ACM 56*, 7 (2013), 78–87.

[39] SHAKSHOBER, D. J. Choosing an I/O Scheduler for Red Hat Enterprise Linux 4 and the 2.6 Kernel. In *In Red Hat magazine* (June 2005).

[40] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round robin. In *Proc. of SIGCOMM '95* (August 1995).

[41] STOICA, I., ABDEL-WAHAB, H., AND JEFFAY, K. On the duality between resource reservation and proportional-share resource allocation. *SPIE* (February 1997).

[42] VALENTE, P., AND CHECCONI, F. High Throughput Disk Scheduling with Fair Bandwidth Distribution. In *IEEE Transactions on Computers* (2010), no. 9, pp. 1172–1186.

[43] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: performance insulation for shared storage servers. In *USENIX FAST* (Berkeley, CA, USA, 2007).

[44] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery scheduling: flexible proportional-share resource management. In *Usenix OSDI* (1994).

[45] WANG, Y., AND MERCHANT, A. Proportional-share scheduling for distributed storage systems. In *Usenix FAST* (Feb 2007).

[46] WONG, T. M., GOLDING, R. A., LIN, C., AND BECKER-SZENDY, R. A. Zygaria: Storage performance as a managed resource. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium* (Washington, DC, USA, 2006), RTAS '06, IEEE Computer Society, pp. 125–134.

[47] WU, X., AND REDDY, A. L. N. Exploiting concurrency to improve latency and throughput in a hybrid storage system. In *MASCOTS* (2010), pp. 14–23.

[48] ZHANG, J., SIVASUBRAMANIAM, A., WANG, Q., RISKA, A., AND RIEDEL, E. Storage performance virtualization via throughput and latency control. In *MASCOTS* (2005), pp. 135–142.

[49] ZHANG, L. VirtualClock: A new traffic control algorithm for packet-switched networks. *ACM Trans. Comput. Syst. 9*, 2, 101–124.

SpringFS: Bridging Agility and Performance in Elastic Distributed Storage

Lianghong Xu*, James Cipar*, Elie Krevat*, Alexey Tumanov*
Nitin Gupta*, Michael A. Kozuch†, Gregory R. Ganger*
*Carnegie Mellon University, †Intel Labs

Abstract

Elastic storage systems can be expanded or contracted to meet current demand, allowing servers to be turned off or used for other tasks. However, the usefulness of an elastic distributed storage system is limited by its agility: how quickly it can increase or decrease its number of servers. Due to the large amount of data they must migrate during elastic resizing, state-of-the-art designs usually have to make painful tradeoffs among performance, elasticity and agility.

This paper describes an elastic storage system, called SpringFS, that can quickly change its number of active servers, while retaining elasticity and performance goals. SpringFS uses a novel technique, termed *bounded write offloading*, that restricts the set of servers where writes to overloaded servers are redirected. This technique, combined with the read offloading and passive migration policies used in SpringFS, minimizes the work needed before deactivation or activation of servers. Analysis of real-world traces from Hadoop deployments at Facebook and various Cloudera customers and experiments with the SpringFS prototype confirm SpringFS's agility, show that it reduces the amount of data migrated for elastic resizing by up to two orders of magnitude, and show that it cuts the percentage of active servers required by 67–82%, outdoing state-of-the-art designs by 6–120%.

1 Introduction

Distributed storage can and should be elastic, just like other aspects of cloud computing. When storage is provided via single-purpose storage devices or servers, separated from compute activities, elasticity is useful for reducing energy usage, allowing temporarily unneeded storage components to be powered down. However, for storage provided via multi-purpose servers (e.g. when a server operates as both a storage node in a distributed filesystem and a compute node), such elasticity is even more valuable— providing cloud infrastructures with the freedom to use such servers for other purposes, as tenant demands and priorities dictate. This freedom may be particularly important for increasingly prevalent data-intensive computing activities (e.g., data analytics).

Data-intensive computing over big data sets is quickly becoming important in most domains and will be a major consumer of future cloud computing resources [7, 4, 3, 2]. Many of the frameworks for such computing (e.g., Hadoop [1] and Google's MapReduce [10]) achieve efficiency by distributing and storing the data on the same servers used for processing it. Usually, the data is replicated and spread evenly (via randomness) across the servers, and the entire set of servers is assumed to always be part of the data analytics cluster. Little-to-no support is provided for elastic sizing[1] of the portion of the cluster that hosts storage—only nodes that host no storage can be removed without significant effort, meaning that the storage service size can only grow.

Some recent distributed storage designs (e.g., Sierra [18], Rabbit [5]) provide for elastic sizing, originally targeted for energy savings, by distributing replicas among servers such that subsets of them can be powered down when the workload is low without affecting data availability; any server with the *primary replica* of data will remain active. These systems are designed mainly for performance or elasticity (how small the system size can shrink to) goals, while overlooking the importance of agility (how quickly the system can resize its footprint in response to workload variations), which we find has a significant impact on the machine-hour savings (and so the operating cost savings) one can potentially achieve. As a result, state-of-the-art elastic storage systems must make painful tradeoffs among these goals, unable to fulfill them at the same time. For example, Sierra balances load across all active servers and thus provides good performance. However, this even data layout limits elasticity— at least one third of the servers must always be active (assuming 3-way replication), wasting machine hours that could be used for other purposes when the workload is very low. Further, rebalancing the data layout when turning servers back on induces significant migration overhead, impairing system agility.

[1] We use "elastic sizing" to refer to dynamic online resizing, down from the full set of servers and back up, such as to adapt to workload variations. The ability to add new servers, as an infrequent administrative action, is common but does not itself make a storage service "elastic" in this context; likewise with the ability to survive failures of individual storage servers.

In contrast, Rabbit can shrink its active footprint to a much smaller size (≈10% of the cluster size), but its reliance on Everest-style write offloading [16] induces significant cleanup overhead when shrinking the active server set, resulting in poor agility.

This paper describes a new elastic distributed storage system, called SpringFS, that provides the elasticity of Rabbit *and* the peak write bandwidth characteristic of Sierra, while maximizing agility at each point along a continuum between their respective best cases. The key idea is to employ a small set of servers to store all primary replicas nominally, but (when needed) offload writes that would go to overloaded servers to only the *minimum* set of servers that can satisfy the write throughput requirement (instead of *all* active servers). This technique, termed *bounded write offloading,* effectively restricts the distribution of primary replicas during offloading and enables SpringFS to adapt dynamically to workload variations while meeting performance targets with a minimum loss of agility—most of the servers can be extracted without needing any pre-removal cleanup. SpringFS further improves agility by minimizing the cleanup work involved in resizing with two more techniques: *read offloading* offloads reads from write-heavy servers to reduce the amount of write offloading needed to achieve the system's performance targets; *passive migration* delays migration work by a certain time threshold during server re-integration to reduce the overall amount of data migrated. With these techniques, SpringFS achieves agile elasticity while providing performance comparable to a non-elastic storage system.

Our experiments demonstrate that the SpringFS design enables significant reductions in both the fraction of servers that need to be active and the amount of migration work required. Indeed, its design for where and when to offload writes enables SpringFS to resize elastically without performing any data migration at all in most cases. Analysis of traces from six real Hadoop deployments at Facebook and various Cloudera customers show the oft-noted workload variation and the potential of SpringFS to exploit it—SpringFS reduces the amount of data migrated for elastic resizing by up to two orders of magnitude, and cuts the percentage of active servers required by 67–82%, outdoing state-of-the-art designs like Sierra and Rabbit by 6–120%.

This paper makes three main contributions: First, to the best of our knowledge, it is the first to show the importance of agility in elastic distributed storage, highlighting the need to resize quickly (at times) rather than just hourly as in previous designs. Second, SpringFS introduces a novel write offloading policy that bounds the set of servers to which writes to over-loaded primary servers are redirected. Bounded write offloading, together with read offloading and passive migration significantly improve the system's agility by reducing the cleanup work during elastic resizing. These techniques apply generally to elastic storage with an uneven data layout. Third, we demonstrate the significant machine-hour savings that can be achieved with elastic resizing, using six real-world HDFS traces, and the effectiveness of SpringFS's policies at achieving a "close-to-ideal" machine-hour usage.

The remainder of this paper is organized as follows. Section 2 describes elastic distributed storage generally, the importance of agility in such storage, and the limitations of the state-of-the-art data layout designs in fulfilling elasticity, agility and performance goals at the same time. Section 3 describes the key techniques in SpringFS design and how they can increase agility of elasticity. Section 4 overviews the SpringFS implementation. Section 5 evaluates the SpringFS design.

2 Background and Motivation

This section motivates our work. First, it describes the related work on elastic distributed storage, which provides different mechanisms and data layouts to allow servers to be extracted while maintaining data availability. Second, it demonstrates the significant impact of agility on aggregate machine-hour usage of elastic storage. Third, it describes the limitations of state-of-the-art elastic storage systems and how SpringFS fills the significant gap between agility and performance.

2.1 Related Work

Most distributed storage is not elastic. For example, the cluster-based storage systems commonly used in support of cloud and data-intensive computing environments, such as the Google File System(GFS) [11] or the Hadoop Distributed Filesystem [1], use data layouts that are not amenable to elasticity. The Hadoop Distributed File System (HDFS), for example, uses a replication and data-layout policy wherein the first replica is placed on a node in the same rack as the writing node (preferably the writing node, if it contributes to DFS storage), the second and third on random nodes in a randomly chosen different rack than the writing node. In addition to load balancing, this data layout provides excellent availability properties—if the node with the primary replica fails, the other replicas maintain data availability; if an entire rack fails (e.g., through the failure of a communication link), data availability is maintained via the replica(s) in another rack. But, such a data layout prevents elasticity by requiring that almost all nodes be active—no more than one node per rack can be turned off without a high likelihood of making some data unavailable.

Recent research [5, 13, 18, 19, 17] has provided new data layouts and mechanisms for enabling elasticity in distributed storage. Most notable are Rabbit [5] and Sierra [18]. Both organize replicas such that one copy of data is always on a specific subset of servers, termed *primaries*, so as to allow the remainder of the nodes to be powered down without affecting availability, when the workload is low. With workload increase, they can be turned back on. The same designs and data distribution schemes would allow for servers to be used for other functions, rather than turned off, such as for higher-priority (or higher paying) tenants' activities. Writes intended for servers that are *inactive*[2] are instead written to other *active* servers—an action called *write availability offloading*—and then later reorganized (when servers become active) to conform to the desired data layout.

Rabbit and Sierra build on a number of techniques from previous systems, such as write availability offloading and power gears. Narayanan, Donnelly, and Rowstron [15] described the use of write availability offloading for power management in enterprise storage workloads. The approach was used to redirect traffic from otherwise idle disks to increase periods of idleness, allowing the disks to be spun down to save power. PARAID [20] introduced a geared scheme to allow individual disks in a RAID array to be turned off, allowing the power used by the array to be proportional to its throughput.

Everest [16] is a distributed storage design that used *write performance offloading*[3], rather than to avoid turning on powered-down servers, in the context of enterprise storage. In Everest, disks are grouped into distinct volumes, and each write is directed to a particular volume. When a volume becomes overloaded, writes can be temporarily redirected to other volumes that have spare bandwidth, leaving the overloaded volume to only handle reads. Rabbit applies this same approach, when necessary, to address overload of the primaries.

SpringFS borrows the ideas of write availability and performance offloading from prior elastic storage systems. We expand on previous work by developing new offloading and migration schemes that effectively eliminate the painful tradeoff between agility and write performance in state-of-the-art elastic storage designs.

[2]We generally refer to a server as *inactive* when it is either powered down or reused for other purposes. Conversely, we call a server *active* when it is powered on and servicing requests as part of a elastic distributed storage system.

[3]Write performance offloading differs from write availability offloading in that it offloads writes from overloaded *active* servers to other (relatively idle) *active* servers for better load balancing. The Everest-style and bounded write offloading schemes are both types of write performance offloading.

2.2 Agility is important

By "agility", we mean how quickly one can change the number of servers effectively contributing to a service. For most non-storage services, such changes can often be completed quickly, as the amount of state involved is small. For distributed storage, however, the state involved may be substantial. A storage server can service reads only for data that it stores, which affects the speed of both removing and re-integrating a server. Removing a server requires first ensuring that all data is available on other servers, and re-integrating a server involves replacing data overwritten (or discarded) while it was inactive.

The time required for such migrations has a direct impact on the machine-hours consumed by elastic storage systems. Systems with better agility are able to more effectively exploit the potential of workload variation by more closely tracking workload changes. Previous elastic storage systems rely on very infrequent changes (e.g., hourly resizing in Sierra [18]), but we find that over half of the potential savings is lost with such an approach due to the burstiness of real workloads.

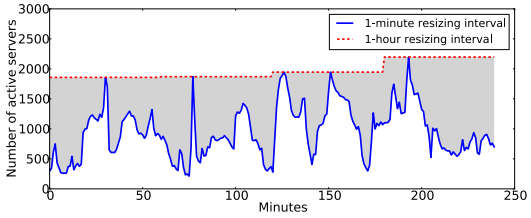


Figure 1: Workload variation in the Facebook trace. The shaded region represents the potential reduction in machine-hour usage with a 1-minute resizing interval.

As one concrete example, Figure 1 shows the number of active servers needed, as a function of time in the trace, to provide the required throughput in a randomly chosen 4-hour period from the Facebook trace described in Section 5. The dashed and solid curves bounding the shaded region represent the minimum number of active servers needed if using 1-hour and 1-minute resizing intervals, respectively. For each such period, the number of active servers corresponds to the number needed to provide the peak throughput in that period, as is done in Sierra to avoid significant latency increases. The area under each curve represents the machine time used for that resizing interval, and the shaded region represents the increased server usage (more than double) for the 1-hour interval. We observe similar burstiness and consequences of it across all of the traces.

2.3 Bridging Agility and Performance

Previous elastic storage systems overlook the importance of agility, focusing on performance and elasticity. This section describes the data layouts of state-of-the-art elastic storage systems, specifically Sierra and Rabbit, and how their layouts represent two specific points in the tradeoff space among elasticity, agility and performance. Doing so highlights the need for a more flexible elastic storage design that fills the void between them, providing greater agility and matching the best of each.

We focus on elastic storage systems that ensure data availability at all times. When servers are extracted from the system, at least one copy of all data must remain active to serve read requests. To do so, state-of-the-art elastic storage designs exploit data replicas (originally for fault tolerance) to ensure that all blocks are available at any power setting. For example, with 3-way replication[4], Sierra stores the first replica of every block (termed *primary replica*) in one third of servers, and writes the other 2 replicas to the other two thirds of servers. This data layout allows Sierra to achieve full peak performance due to balanced load across all active servers, but it limits the elasticity of the system by not allowing the system footprint to go below one third of the cluster size. We show in section 5.2 that such limitation can have a significant impact on the machine-hour savings that Sierra can potentially achieve, especially during periods of low workload.

Rabbit, on the other hand, is able to reduce its system footprint to a much smaller size (≈10% of the cluster size). It does so by storing the replicas according to an *equal-work* data layout, so that it achieves *power proportionality* for read requests. That is, read performance scales linearly with the number of active servers: if 50% of the servers are active, the read performance of Rabbit should be at least 50% of its maximum read performance. The equal-work data layout ensures that, with any number of active servers, each server is able to perform an equal share of the read workload. In a system storing B blocks, with p primary servers and x active servers, each active server must store at least B/x blocks so that reads can be distributed equally, with the exception of the primary servers. Since a copy of all blocks must be stored on the p primary servers, they each store B/p blocks. This ensures (probabilistically) that when a large quantity of data is read, no server must read more than the others and become a bottleneck. This data layout allows Rabbit to keep the number of primary servers ($p = N/e^2$) very small (e is Euler's constant). The small number of primary servers provides great agility—Rabbit is able to shrink its system size down to p without any cleanup work—but it can create bottlenecks for writes. Since the primary servers must store the primary replicas for all blocks, the maximum write throughput of Rabbit is limited by the maximum aggregate write throughout of the p primary servers, even when all servers are active. In contrast, Sierra is able to achieve the same maximum write throughput as that of HDFS, that is, the aggregate write throughput of $N/3$ servers (recall: N servers write 3 replicas for every data block).

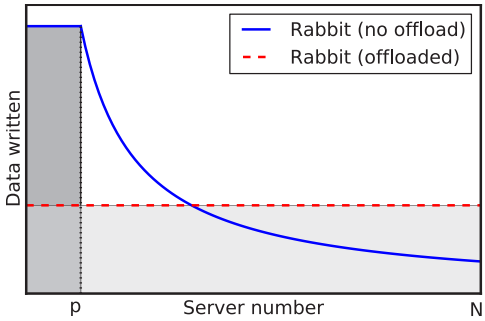


Figure 2: Primary data distribution for Rabbit without offloading (grey) and Rabbit with offloading (light grey). With offloading, primary replicas are spread across all active servers during writes, incurring significant cleanup overhead when the system shrinks its size.

Rabbit borrows write offloading from the Everest system [16] to solve this problem. When primary servers become the write performance bottleneck, Rabbit simply offloads writes that would go to heavily loaded servers across *all* active servers. While such write offloading allows Rabbit to achieve good peak write performance comparable to non-modified HDFS due to balanced load, it significantly impairs system agility by spreading primary replicas across all active servers, as depicted in Figure 2. Consequently, before Rabbit shrinks the system size, cleanup work is required to migrate some primary replicas to the remaining active servers so that at least one complete copy of data is still available after the resizing action. As a result, the improved performance from Everest-style write offloading comes at a high cost in system agility.

Figure 3 illustrates the very different design points represented by Sierra and Rabbit, in terms of the tradeoffs among agility, elasticity and peak write performance. Read performance is the same for all of these systems, given the same number of active servers. The number of servers that store primary replicas indicates the minimal system footprint one can shrink to without any cleanup work. As described above, state-of-the art elastic storage systems such as Sierra and Rabbit suffer from the painful tradeoff between agility and perfor-

[4]We assume 3-way replication for all data blocks throughout this paper, which remains the default policy for HDFS. The data layout designs apply to other replication levels as well. Different approaches than Sierra, Rabbit and SpringFS are needed when erasure codes are used for fault tolerance instead of replication.

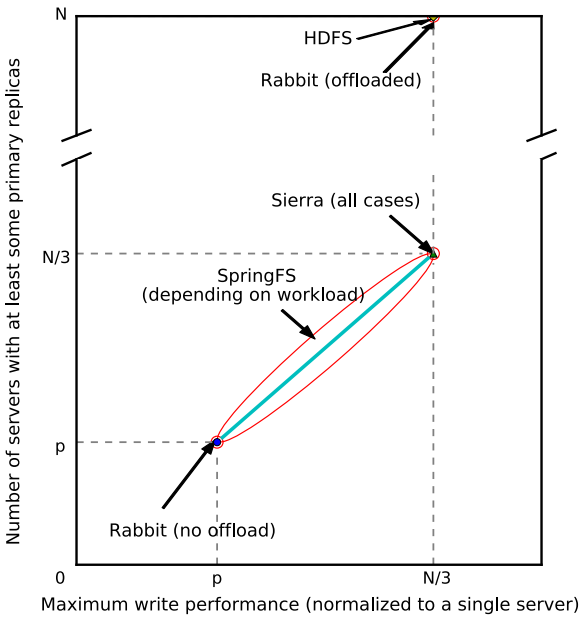


Figure 3: Elastic storage system comparison in terms of agility and performance. N is the total size of the cluster. p is the number of primary servers in the equal-work data layout. Servers with at least some primary replicas cannot be deactivated without first moving those primary replicas. SpringFS provides a continuum between Sierra's and Rabbit's (when no offload) single points in this tradeoff space. When Rabbit requires offload, SpringFS is superior at all points. Note that the y-axis is discontinuous.

mance due to the use of a rigid data layout. SpringFS provides a more flexible design that provides the best-case elasticity of Rabbit, the best-case write performance of Sierra, and much better agility than either. To achieve the range of options shown, SpringFS uses an explicit bound on the *offload set*, where writes of primary replicas to overloaded servers are offloaded to only the *minimum* set of servers (instead of *all* active servers) that can satisfy the current write throughput requirement. This additional degree of freedom allows SpringFS to adapt dynamically to workload changes, providing the desired performance while maintaining system agility.

3 SpringFS Design and Policies

This section describes SpringFS's data layout, as well as the bounded write offloading and read offloading policies that minimize the cleanup work needed before deactivation of servers. It also describes the passive migration policy used during a server's re-integration to address data that was written during the server's absence.

3.1 Data Layout and Offloading Policies

Data layout. Regardless of write performance, the equal-work data layout proposed in Rabbit enables the smallest number of primary servers and thus provides the best elasticity in state-of-the-art designs.[5] SpringFS retains such elasticity using a variant of the equal-work data layout, but addresses the agility issue incurred by Everest-style offloading when write performance bottlenecks arise. The key idea is to bound the distribution of primary replicas to a minimal set of servers (instead of offloading them to all active servers), given a target maximum write performance, so that the cleanup work during server extraction can be minimized. This *bounded write offloading* technique introduces a parameter called the *offload set*: the set of servers to which primary replicas are offloaded (and as a consequence receive the most write requests). The offload set provides an adjustable tradeoff between maximum write performance and cleanup work. With a small offload set, few writes will be offloaded, and little cleanup work will be subsequently required, but the maximum write performance will be limited. Conversely, a larger offload set will offload more writes, enabling higher maximum write performance at the cost of more cleanup work to be done later. Figure 4 shows the SpringFS data layout and its relationship with the state-of-the-art elastic data layout designs. We denote the size of the offload set as m, the number of primary servers in the equal-work layout as p, and the total size of the cluster as N. When m equals p, SpringFS behaves like Rabbit and writes all data according to the equal-work layout (no offload); when m equals $N/3$, SpringFS behaves like Sierra and load balances all writes (maximum performance). As illustrated in Figure 3, the use of the tunable offload set allows SpringFS to achieve both end points and points in between.

Choosing the offload set. The offload set is not a rigid setting, but determined on the fly to adapt to workload changes. Essentially, it is chosen according to the target maximum write performance identified for each resizing interval. Because servers in the offload set write one complete copy of the primary replicas, the size of the offload set is simply the maximum write throughput in the workload divided by the write throughput a single server can provide. Section 5.2 gives a more detailed description of how SpringFS chooses the offload set (and the number of active servers) given the target workload performance.

Read offloading. One way to reduce the amount of cleanup work is to simply reduce the amount of write offloading that needs to be done to achieve the system's

[5] Theoretically, no other data layout can achieve a smaller number of primary servers while maintaining power-proportionality for read performance.

performance targets. When applications simultaneously read and write data, SpringFS can coordinate the read and write requests so that reads are preferentially sent to higher numbered servers that naturally handle fewer write requests. We call this technique *read offloading*.

Despite its simplicity, read offloading allows SpringFS to increase write throughput without changing the offload set by taking read work away from the low numbered servers (which are the bottleneck for writes). When a read occurs, instead of randomly picking one among the servers storing the replicas, SpringFS chooses the server that has received the least number of total requests recently. (The one exception is when the client requesting the read has a local copy of the data. In this case, SpringFS reads the replica directly from that server to exploit machine locality.) As a result, lower numbered servers receive more writes while higher numbered servers handle more reads. Such read/write distribution balances the overall load across all the active servers while reducing the need for write offloading.

Replica placement. When a block write occurs, SpringFS chooses target servers for the 3 replicas in the following steps: The primary replica is load balanced across (and thus bounded in) the m servers in the current offload set. (The one exception is when the client requesting the write is in the offload set. In this case, SpringFS writes the primary copy to that server, instead of the server with the least load in the offload set, to exploit machine locality.) For non-primary replicas, SpringFS first determines their target servers according to the equal-work layout. For example, the target server for the secondary replica would be a server numbered between $p+1$ and ep, and that for the tertiary replica would be a server numbered between $ep+1$ and e^2p, both following the probability distribution as indicated by the equal-work layout (lower numbered servers have higher probability to write the non-primary replicas). If the target server number is higher than m, the replica is written to that server. However, if the target server number is between $p+1$ and m (a subset of the offload set), the replica is instead redirected and load balanced across servers outside the offload set, as shown in the shaded regions in Figure 4. Such redirection of non-primary replicas reduces the write requests going to the servers in the offload set and ensures that these servers store only the primary replicas.

Fault tolerance and multi-volume support. The use of an uneven data layout creates new problems for fault tolerance and capacity utilization. For example, when a primary server fails, the system may need to re-integrate some non-primary servers to restore the primary replicas onto a new server. SpringFS includes the data layout refinements from Rabbit that minimize the number of additional servers that must be re-activated if such failure happens. Writes that would have gone to the failed primary server are instead redirected to other servers in the offload set to preserve system agility. Like Rabbit, SpringFS also accommodates multi-volume data layouts in which independent volumes use distinct servers as primaries in order to allow small values of p without limiting storage capacity utilization to $3p/N$.

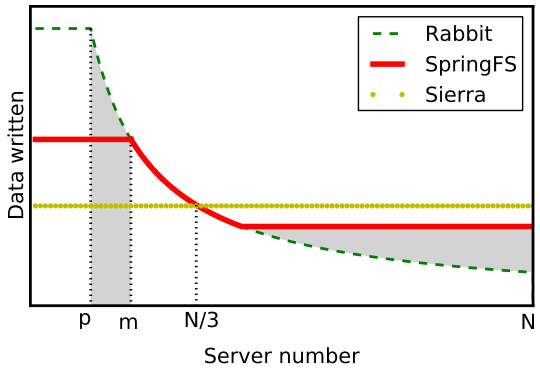


Figure 4: SpringFS data layout and its relationship with previous designs. The offload set allows SpringFS to achieve a dynamic tradeoff between the maximum write performance and the cleanup work needed before extracting servers. In SpringFS, all primary replicas are stored in the m servers of the offload set. The shaded regions indicate writes of non-primary replicas that would have gone to the offload set (in SpringFS) are instead redirected and load balanced outside the set.

3.2 Passive Migration for Re-integration

When SpringFS tries to write a replica according to its target data layout but the chosen server happens to be inactive, it must still maintain the specified replication factor for the block. To do this, another host must be selected to receive the write. *Availability offloading* is used to redirect writes that would have gone to inactive servers (which are unavailable to receive requests) to the active servers. As illustrated in Figure 5, SpringFS load balances availability offloaded writes together with the other writes to the system. This results in the availability offloaded writes going to the less-loaded active servers rather than adding to existing write bottlenecks on other servers.

Because of availability offloading, re-integrating a previously deactivated server is more than simply restarting its software. While the server can begin servicing its share of the write workload immediately, it can only service reads for blocks that it stores. Thus, filling it according to its place in the target equal-work layout is part of full re-integration.

When a server is reintegrated to address a workload

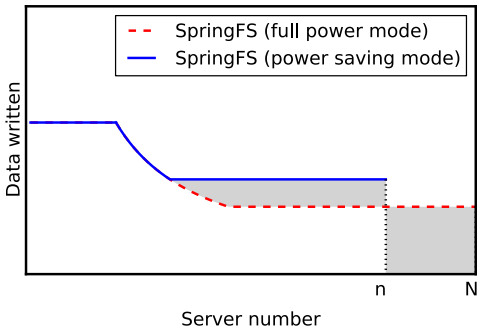


Figure 5: Availability offloading. When SpringFS works in the power saving mode, some servers ($n+1$ to N) are deactivated. The shaded regions show that writes that would have gone to these inactive servers are offloaded to higher numbered active servers for load balancing.

increase, the system needs to make sure that the active servers will be able to satisfy the read performance requirement. One option is to aggressively restore the equal work data layout before reintegrated servers begin servicing reads. We call this approach *aggressive migration*. Before anticipated workload increases, the migration agent would activate the right number of servers and migrate some data to the newly activated servers so that they store enough data to contribute their full share of read performance. The migration time is determined by the number of blocks that need to be migrated, the number of servers that are newly activated, and the I/O throughput of a single server. With aggressive migration, cleanup work is never delayed. Whenever a resizing action takes place, the property of the equal-work layout is obeyed—server x stores no less than $\frac{B}{x}$ blocks.

SpringFS takes an alternate approach called *passive migration*, based on the observation that cleanup work when re-integrating a server is not as important as when deactivating a server (for which it preserves data availability), and that the total amount of cleanup work can be reduced by delaying some fraction of migration work while performance goals are still maintained (which makes this approach better than aggressive migration). Instead of aggressively fixing the data layout (by activating the target number of servers in advance for a longer period of time), SpringFS temporarily activates more servers than would minimally be needed to satisfy the read throughput requirement and utilizes the extra bandwidth for migration work and to address the reduced number of blocks initially on each reactivated server. The number of extra servers that need to be activated is determined in two steps. First, an initial number is chosen to ensure that the number of valid data blocks still stored on the activated servers is more than the fraction of read workload they need to satisfy, so that the performance requirement is satisfied. Second, the number may be increased so that the extra servers provide enough I/O bandwidth to finish a fraction ($1/T$, where T is the migration threshold as described below) of migration work. To avoid migration work building up indefinitely, the migration agent sets a time threshold so that whenever a migration action takes place, it is guaranteed to finish within T minutes. With $T > 1$ (the default resizing interval), SpringFS delays part of the migration work while satisfying throughput requirement. Because higher numbered servers receive more writes than their equal-work share, due to write offloading, some delayed migration work can be replaced by future writes, which reduces the overall amount of data migration. If T is too large, however, the cleanup work can build up so quickly that even activating all the servers cannot satisfy the throughput requirement. In practice, we find a migration threshold $T = 10$ to be a good choice and use this setting for the trace analysis in Section 5. Exploring automatic setting of T is an interesting future work.

4 Implementation

SpringFS is implemented as a modified instance of the Hadoop Distributed File System (HDFS), version 0.19.1[6]. We build on a *Scriptable Hadoop* interface that we built into Hadoop to allow experimenters to implement policies in external programs that are called by the modified Hadoop. This enables rapid prototyping of new policies for data placement, read load balancing, task scheduling, and re-balancing. It also enables us to emulate both Rabbit and SpringFS in the same system, for better comparison. SpringFS mainly consists of four components: data placement agent, load balancer, resizing agent and migration agent, all implemented as python programs called by the Scriptable Hadoop interface.

Data placement agent. The data placement agent determines where to place blocks according to the SpringFS data layout. Ordinarily, when a HDFS client wishes to write a block, it contacts the HDFS NameNode and asks where the block should be placed. The NameNode returns a list of pseudo-randomly chosen DataNodes to the client, and the client writes the data directly to these DataNodes. The data placement agent starts together with the NameNode, and communicates with the NameNode using a simple text-based protocol over `stdin` and `stdout`. To obtain a placement decision for the R replicas of a block, the NameNode writes the name of the client machine as well as a list of candi-

[6]0.19.1 was the latest Hadoop version when our work started. We have done a set of experiments to verify that HDFS performance differs little, on our experimental setup, between version 0.19.1 and the latest stable version (1.2.1). We believe our results and findings are not significantly affected by still using this older version of HDFS.

date DataNodes to the placement agent's `stdin`. The placement agent can then filter and reorder the candidates, returning a prioritized list of targets for the write operation. The NameNode then instructs the client to write to the first R candidates returned.

Load balancer. The load balancer implements the read offloading policy and preferentially sends reads to higher numbered servers that handle fewer write requests whenever possible. It keeps an estimate of the load on each server by counting the number of requests sent to each server recently. Every time SpringFS assigns a block to a server, it increments a counter for the server. To ensure that recent activity has precedence, these counters are periodically decayed by 0.95 every 5 seconds. While this does not give the exact load on each server, we find its estimates good enough (within 3% off optimal) for load balancing among relatively homogeneous servers.

Resizing agent. The resizing agent changes SpringFS's footprint by setting an *activity state* for each DataNode. On every read and write, the data placement agent and load balancer will check these states and remove all "INACTIVE" DataNodes from the candidate list. Only "ACTIVE" DataNodes are able to service reads or writes. By setting the activity state for DataNodes, we allow the resources (e.g., CPU and network) of "INACTIVE" nodes to be used for other activities with no interference from SpringFS activities. We also modified the HDFS mechanisms for detecting and repairing under-replication to assume that "INACTIVE" nodes are not failed, so as to avoid undesired re-replication.

Migration agent. The migration agent crawls the entire HDFS block distribution (once) when the NameNode starts, and it keeps this information up-to-date by modifying HDFS to provide an interface to get and change the current data layout. It exports two metadata tables from the NameNode, mapping file names to block lists and blocks to DataNode lists, and loads them into a SQLite database. Any changes to the metadata (e.g., creating a file, creating or migrating a block) are then reflected in the database on the fly. When data migration is scheduled, the SpringFS migration agent executes a series of SQL queries to detect layout problems, such as blocks with no primary replica or hosts storing too little data. It then constructs a list of migration actions to repair these problems. After constructing the full list of actions, the migration agent executes them in the background. To allow block-level migration, we modified the HDFS client utility to have a "relocate" operation that copies a block to a new server. The migration agent uses GNU Parallel to execute many relocates simultaneously.

5 Evaluation

This section evaluates SpringFS and its offloading policies. Measurements of the SpringFS implementation show that it provide performance comparable to unmodified HDFS, that its policies improve agility by reducing the cleanup required, and that it can agilely adapt its number of active servers to provide required performance levels. In addition, analysis of six traces from real Hadoop deployments shows that SpringFS's agility enables significantly reduced commitment of active servers for the highly dynamic demands commonly seen in practice.

5.1 SpringFS prototype experiments

Experimental setup: Our experiments were run on a cluster of 31 machines. The modified Hadoop software is run within KVM virtual machines, for software management purposes, but each VM gets its entire machine and is configured to use all 8 CPU cores, all 8 GB RAM, and 100 GB of local hard disk space. One machine was configured as the Hadoop master, hosting both the NameNode and the JobTracker. The other 30 machines were configured as slaves, each serving as an HDFS DataNode and a Hadoop TaskTracker. Unless otherwise noted, SpringFS was configured for 3-way replication ($R = 3$) and 4 primary servers ($p = 4$).

To simulate periods of high I/O activity, and effectively evaluate SpringFS under different mixes of I/O operations, we used a modified version of the standard Hadoop TestDFSIO storage system benchmark called TestDFSIO2. Our modifications allow for each node to generate a mix of block-size (128 MB) reads and writes, distributed randomly across the block ID space, with a user-specified write ratio.

Except where otherwise noted, we specify a file size of 2GB per node in our experiments, such that the single Hadoop map task per node reads or writes 16 blocks. The total time taken to transfer all blocks is aggregated and used to determine a global throughput. In some cases, we break down the throughput results into the average aggregate throughput of just the block reads or just the block writes. This enables comparison of SpringFS's performance to the unmodified HDFS setup with the same resources. Our experiments are focused primarily on the relative performance changes as agility-specific parameters and policies are modified. Because the original Hadoop implementation is unable to deliver the full performance of the underlying hardware, our system can only be compared reasonably with it and not the capability of the raw storage devices.

Effect of offloading policies: Our evaluation focuses on how SpringFS's offloading policies affect per-

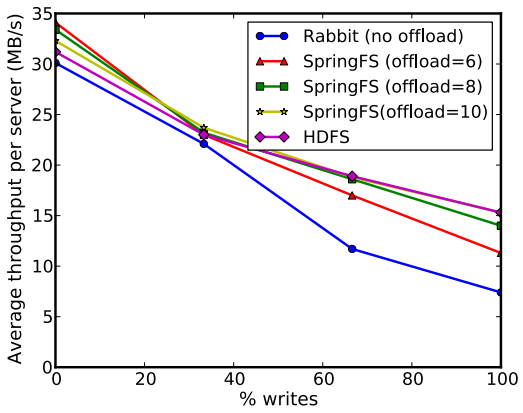


Figure 6: Performance comparison of Rabbit with no offload, original HDFS, and SpringFS with varied offload set.

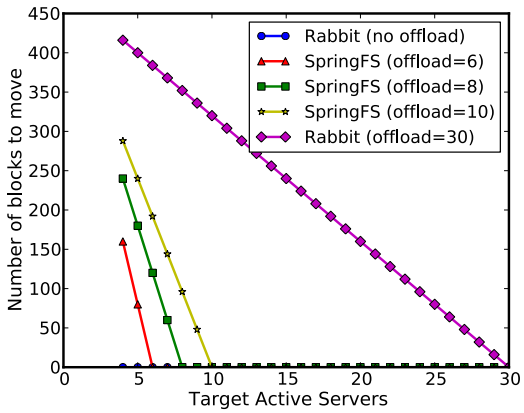


Figure 7: Cleanup work (in blocks) needed to reduce active server count from 30 to X, for different offload settings. The "(offload=6)", "(offload=8)" and "(offload=10)" lines correspond to SpringFS with bounded write offloading. The "(offload=30)" line corresponds to Rabbit using Everest-style write offloading. Deactivating only non-offload servers requires no block migration. The amount of cleanup work is linear in the number of target active servers.

formance and agility. We also measure the cleanup work created by offloading and demonstrate that SpringFS's number of active servers can be adapted agilely to changes in workload intensity, allowing machines to be extracted and used for other activities.

Figure 6 presents the peak sustained I/O bandwidth measured for HDFS, Rabbit and SpringFS at different offload settings. (Rabbit and SpringFS are identical when no offloading is used.) In this experiment, the write ratio is varied to demonstrate different mixes of read and write requests. SpringFS, Rabbit and HDFS achieve similar performance for a read-only workload, because in all cases there is a good distribution of blocks and replicas across the cluster over which to balance the load. The read performance of SpringFS slightly outperforms the original HDFS due to its explicit load tracking for balancing.

When no offloading is needed, both Rabbit and SpringFS are highly elastic and able to shrink 87% (26 non-primary servers out of 30) with no cleanup work. However, as the write workload increases, the equal-work layout's requirement that one replica be written to the primary set creates a bottleneck and eventually a slowdown of around 50% relative to HDFS for a maximum-speed write-only workload. SpringFS provides the flexibility to tradeoff some amount of agility for better write throughput under periods of high write load. As the write ratio increases, the effect of SpringFS's offloading policies becomes more visible. Using only a small number of offload servers, SpringFS significantly reduces the amount of data written to the primary servers and, as a result, significantly improves performance over Rabbit. For example, increasing the offload set from four (i.e., just the four primaries) to eight doubles maximum throughput for the write-only workload, while remaining agile—the cluster is still able to shrink 74% with no cleanup work.

Figure 7 shows the number of blocks that need to be relocated to preserve data availability when reducing the number of active servers. As desired, SpringFS's data placements are highly amenable to fast extraction of servers. Shrinking the number of nodes to a count exceeding the cardinality of the offload set requires no clean-up work. Decreasing the count into the write offload set is also possible, but comes at some cost. As expected, for a specified target, the cleanup work grows with an increase in the offload target set. SpringFS with no offload reduces to the based equal-work layout, which needs no cleanup work when extracting servers but suffers from write performance bottlenecks. The most interesting comparison is Rabbit's full offload (offload=30) against SpringFS's full offload (offload=10). Both provide the cluster's full aggregate write bandwidth, but SpringFS's offloading scheme does it with much greater agility—66% of the cluster could still be extracted with no cleanup work and more with small amounts of cleanup. We also measured actual cleanup times, finding (not surprisingly) that they correlate strongly with the number of blocks that must be moved.

SpringFS's read offloading policy is simple and reduces the cleanup work resulting from write offloading. To ensure that its simplicity does not result in lost opportunity, we compare it to the optimal, oracular scheduling policy with claircognizance of the HDFS layout. We

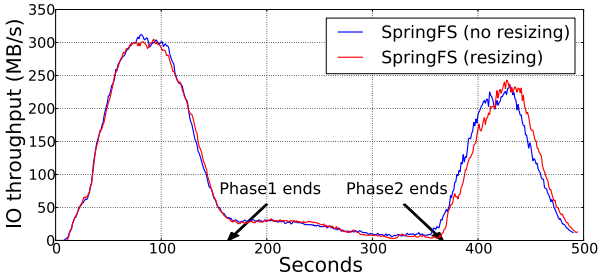


Figure 8: Agile resizing in a 3-phase workload

use an Integer Linear Programming (ILP) model that minimizes the number of reads sent to primary servers from which primary replica writes are offloaded. The SpringFS read offloading policy, despite its simple realization, compares favorably and falls within 3% from optimal on average.

Agile resizing in SpringFS: Figure 8 illustrates SpringFS's ability to resize quickly and deliver required performance levels. It uses a sequence of three benchmarks to create phases of workload intensity and measures performance for two cases: "SpringFS (no resizing)" where the full cluster stays active throughout the experiment and "SpringFS (resizing)" where the system size is changed with workload intensity. As expected, the performance is essentially the same for the two cases, with a small delay observed when SpringFS re-integrates servers for the third phase. But, the number of machine hours used is very different, as SpringFS extracts machines during the middle phase.

This experiment uses a smaller setup, with only 7 DataNodes, 2 primaries, 3 in the offload set, and 2-way replication. The workload consists of 3 consecutive benchmarks. The first benchmark is a TestDFSIO2 benchmark that writes 7 files, each 2GB in size for a total of 14GB written. The second benchmark is one SWIM job [9] randomly picked from a series of SWIM jobs synthesized from a Facebook trace which reads 4.2GB and writes 8.4GB of data. The third benchmark is also a TestDFSIO2 benchmark, but with a write ratio of 20%. The TestDFSIO2 benchmarks are I/O intensive, whereas the SWIM job consumes only a small amount of the full I/O throughput. For the resizing case, 4 servers are extracted after the first write-only TestDFSIO2 benchmark finishes (shrinking the active set to 3), and those servers are reintegrated when the second TestDFSIO2 job starts. In this experiment, the resizing points are manually set when phase switch happens. Automatic resizing can be done based on previous work on workload prediction [6, 12, 14].

The results in Figure 8 are an average of 10 runs for both cases, shown with a moving average of 3 seconds. The I/O throughput is calculated by summing read throughput and write throughput multiplied by the replication factor. Decreasing the number of active SpringFS servers from 7 to 3 does not have an impact on its performance, since no cleanup work is needed. As expected, resizing the cluster from 3 nodes to 7 imposes a small performance overhead due to background block migration, but the number of blocks to be migrated is very small—about 200 blocks are written to SpringFS with only 3 active servers, but only 4 blocks need to be migrated to restore the equal-work layout. SpringFS's offloading policies keep the cleanup work small, for both directions. As a result, SpringFS extracts and re-integrates servers very quickly.

5.2 Policy analysis with real-world traces

This subsection evaluates SpringFS in terms of machine-hour usage with real-world traces from six industry Hadoop deployments and compares it against three other storage systems: Rabbit, Sierra, and the default HDFS. We evaluate each system's layout policies with each trace, calculate the amount of cleanup work and the estimated cleaning time for each resizing action, and summarize the aggregated machine-hour usage consumed by each system for each trace. The results show that SpringFS significantly reduces machine-hour usage even compared to the state-of-the-art elastic storage systems, especially for write-intensive workloads.

Trace overview: We use traces from six real Hadoop deployments representing a broad range of business activities, one from Facebook and five from different Cloudera customers. The six traces are described and analyzed in detail by Chen et al. [8]. Table 1 summarizes key statistics of the traces. The Facebook trace (FB) comes from Hadoop DataNode logs, each record containing timestamp, operation type (HDFS_READ or HDFS_WRITE), and the number of bytes processed. From this information, we calculate the aggregate HDFS read/write throughput as well as the total throughput, which is the sum of read and write throughput multiplied by the replication factor (3 for all the traces). The five Cloudera customer traces (CC-a through CC-e, using the terminology from [8]) all come from Hadoop job history logs, which contain per-job records of job duration, HDFS input/output size, etc. Assuming the amount of HDFS data read or written for each job is distributed evenly within the job duration, we also obtain the aggregated HDFS throughput at any given point of time, which is then used as input to the analysis program.

Trace analysis and results: To simplify calculation, we make several assumptions. First, the maximum measured total throughput in the traces corresponds to the maximum aggregate performance across all the machines in the cluster. Second, the maximum throughput

Table 1: Trace summary. CC is "Cloudera Customer" and FB is "Facebook". HDFS bytes processed is the sum of HDFS bytes read and HDFS bytes written.

Trace	Machines	Date	Length	Bytes processed
CC-a	<100	2011	1 month	69TB
CC-b	300	2011	9 days	473TB
CC-c	700	2011	1 month	13PB
CC-d	400-500	2011	2.8 months	5PB
CC-e	100	2011	9 days	446TB
FB	3000	2010	10 days	10.5PB

a single machine can deliver, not differentiating reads and writes, is derived from the maximum measured total throughput divided by the number of machines in the cluster. In order to calculate the machine hour usage for each storage system, the analysis program needs to determine the number of active servers needed at any given point of time. It does this in the following steps: First, it determines the number of active servers needed in the imaginary "ideal" case, where no cleanup work is required at all, by dividing the total HDFS throughput by the maximum throughput a single machine can deliver. Second, it iterates through the number of active servers as a function of time. For each decrease in the active set of servers, it checks for any cleanup work that must be done by analyzing the data layout at that point. If any cleanup is required, it delays resizing until the work is done or the performance requirement demands an increase of the active set, to allow additional bandwidth for necessary cleanup work. For increases in the active set of servers, it turns on some extra servers to satisfy the read throughput and uses the extra bandwidth to do a fraction of migration work, using the passive migration policy (for all the systems) with the migration threshold set to be T=10.

Figures 9 and 10 show the number of active servers needed, as a function of time, for the 6 traces. Each graph has 4 lines, corresponding to the "ideal" storage system, SpringFS, Rabbit and Sierra, respectively. We

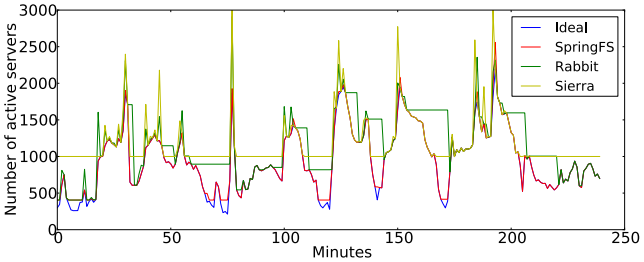


Figure 9: Facebook trace

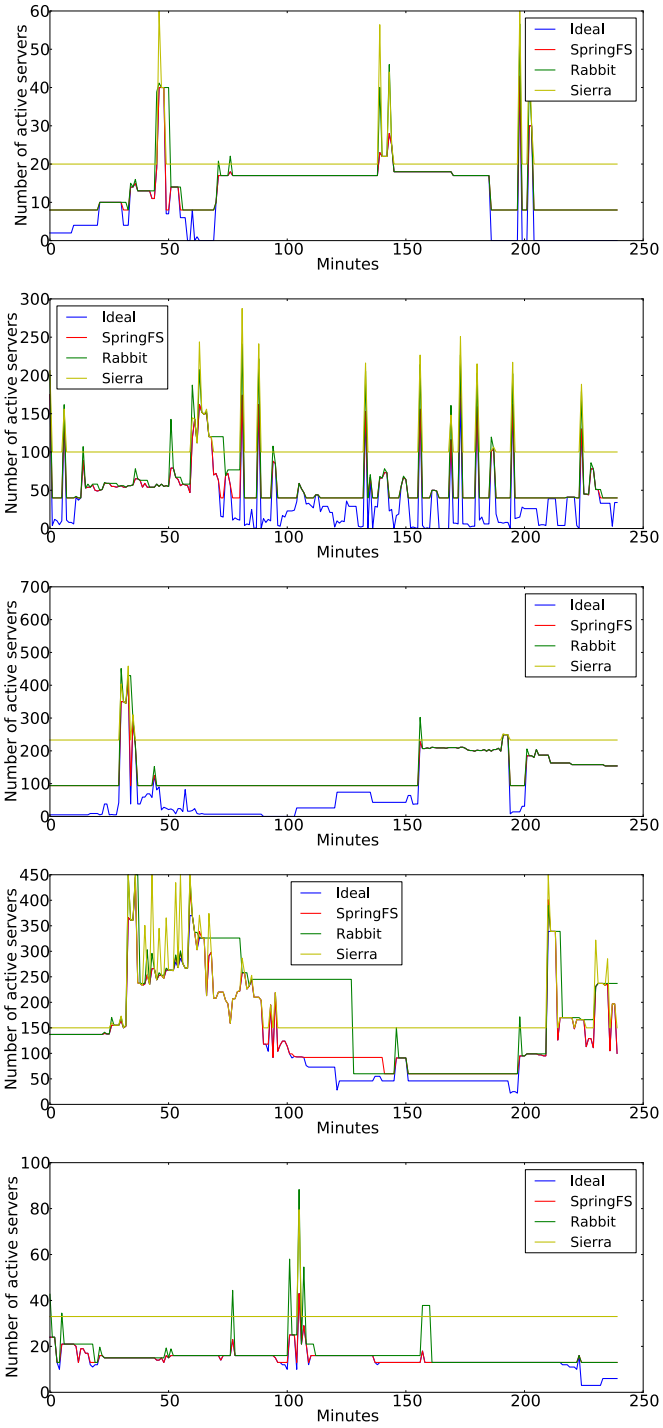


Figure 10: Traces: CC-a, CC-b, CC-c, CC-d, and CC-e

do not show the line for the Default HDFS, but since it is not elastic, its curve would be a horizontal line with the number of active servers always being the full cluster size (the highest value on the Y axis). While the original trace durations range from 9 days to 2.8 months, we only show a 4-hour-period for each trace for clarity. We start trace

replaying more than 3 days before the 4-hour period, to make sure it represents the situation when systems are in a steady state and includes the effect of delaying migration work.

As expected, SpringFS exhibits better agility than Rabbit, especially when shrinking the size of the cluster, since it needs no cleanup work until resizing down to the offload set. Such agility difference between SpringFS and Rabbit is shown in Figure 9 at various points of time (e.g., at minute 110, 140, and 160). The gap between the two lines indicates the number of machine hours saved due to the agility-aware read and bounded write policies used in SpringFS. SpringFS also achieves lower machine-hour usage than Sierra, as confirmed in all the analysis graphs. While a Sierra cluster can shrink down to 1/3 of its total size without any cleanup work, it is not able to further decrease the cluster size. In contrast, SpringFS can shrink the cluster size down to approximately 10% of the original footprint. When I/O activity is low, the difference in minimal system footprint can have a significant impact on the machine-hour usage (e.g., as illustrated in Figure 10(b), Figure 10(c) and Figure 10(e)). In addition, when expanding cluster size, Sierra incurs more cleaning overhead than SpringFS, because deactivated servers need to migrate more data to restore its even data layout. These results are summarized in Figure 11, which shows the extra number of machine hours used by each storage system, compared and normalized to the ideal system. In these traces, SpringFS outperforms the other systems by 6% to 120%. For the traces with a relatively high write ratio, such as the FB, CC-d and CC-e traces, SpringFS is able to achieve a "close-to-ideal" (within 5%) machine-hour usage. SpringFS is less close to ideal for the other three traces because they frequently need even less than the 13% primary servers that SpringFS cannot deactivate.

Figure 12 summarizes the total amount of data migrated by Rabbit, Sierra and SpringFS while running each trace. With bounded write offloading and read offloading, SpringFS is able to reduce the amount of data migration by a factor of 9–208, as compared to Rabbit. SpringFS migrates significantly less data than Sierra as well, because data migrated to restore the equal-work data layout is much less than that to restore an even data layout.

All of the trace analysis above assumes passive migration during server reintegration for all three systems compared, since it is useful to all of them. To evaluate the advantage of passive migration, specifically, we repeated the same trace analysis using the aggressive migration policy. The results show that passive migration reduces the amount of data migrated, relative to aggressive migration, by 1.5–7× (across the six traces) for SpringFS, 1.2–5.6× for Sierra, and 1.2–3× for Rabbit. The benefit for Sierra and SpringFS is more significant, because their data migration occurs primarily during server reintegration.

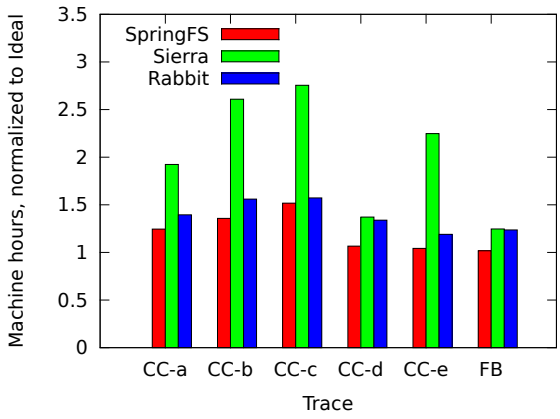


Figure 11: Number of machine hours needed to execute each trace for each system, normalized to the "Ideal" system (1 on the y-axis, not shown).

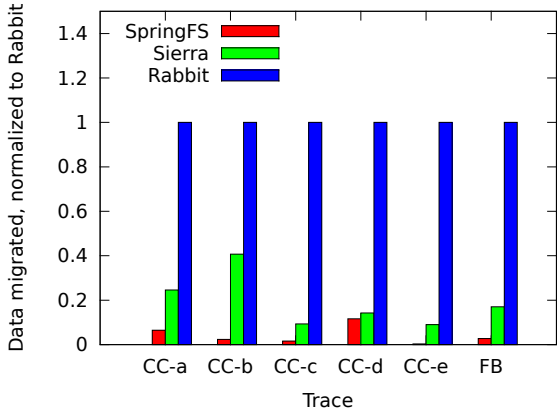


Figure 12: Total data migrated for Rabbit, Sierra and SpringFS, normalized to results for Rabbit.

6 Conclusion

SpringFS is a new elastic storage system that fills the space between state-of-the-art designs in the tradeoff among agility, elasticity, and performance. SpringFS's data layout and offloading/migration policies adapt to workload demands and minimize the data redistribution cleanup work needed for elastic resizing, greatly increasing agility relative to the best previous elastic storage designs. As a result, SpringFS can satisfy the time-varying performance demands of real environments with many fewer machine hours. Such agility provides an important building block for resource-efficient data-intensive

computing (a.k.a. Big Data) in multi-purpose clouds with competing demands for server resources.

There are several directions for interesting future work. For example, the SpringFS data layout assumes that servers are approximately homogeneous, like HDFS does, but some real-world deployments end up with heterogeneous servers (in terms of I/O throughput and capacity) as servers are added and replaced over time. The data layout could be refined to exploit such heterogeneity, such as by using more powerful servers as primaries. Second, SpringFS's design assumes a relatively even popularity of data within a given dataset, as exists for Hadoop jobs processing that dataset, so it will be interesting to explore what aspects change when addressing the unbalanced access patterns (e.g., Zipf distribution) common in servers hosting large numbers of relatively independent files.

Acknowledgements: We thank Cloudera and Facebook for sharing the traces and Yanpei Chen for releasing SWIM. We thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Facebook, Fusion-io, Google, HP Labs, Hitachi, Huawei, Intel, Microsoft Research, NEC Labs, NetApp, Oracle, Panasas, Samsung, Seagate, Symantec, VMWare, and Western Digital) for their interest, insights, feedback, and support. This research was sponsored in part by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC). Experiments were enabled by generous hardware donations from Intel, NetApp, and APC.

References

[1] Hadoop, 2012. http://hadoop.apache.org.

[2] MIT, Intel unveil new initiatives addressing 'Big Data', 2012. http://web.mit.edu/newsoffice/2012/big-data-csail-intel-center-0531.html.

[3] AMPLab, 2013. http://amplab.cs.berkeley.edu.

[4] ISTC-CC Research, 2013. www.istc-cc.cmu.edu.

[5] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan. Robust and flexible power-proportional storage. *ACM Symposium on Cloud Computing*, pages 217–228, 2010.

[6] P. Bodik, M. Armbrust, K. Canini, A. Fox, M. Jordan, and D. Patterson. A case for adaptive datacenters to conserve energy and improve reliability. *University of California at Berkeley, Tech. Rep. UCB/EECS-2008-127*, 2008.

[7] R. E. Bryant. Data-Intensive Supercomputing: The Case for DISC. Technical report, Carnegie Mellon University, 2007.

[8] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross industry study of mapreduce workloads. *VLDB*, 2012.

[9] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. *MASCOTS*, 2011.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–108, January 2008.

[11] S. Ghemawat, H. Gobioff, and S. tak Leung. The Google File System. In *SOSP*, pages 29–43, 2003.

[12] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload analysis and demand prediction of enterprise data center applications. *IISWC*, 2007.

[13] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. *HotPower*, 2009.

[14] M. Lin, A. Wierman, L. L. Andrew, and E. Thereska. Dynamic right-sizing for power-proportional data centers. *INFOCOM*, 2011.

[15] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *USENIX Conference on File and Storage Technologies*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[16] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through I/O off-loading. In *OSDI*, 2008.

[17] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building Distributed Enterprise Disk Arrays from Commodity Components. In *ASPLOS*, pages 48–58, 2004.

[18] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *EuroSys*, pages 169–182, 2011.

[19] N. Vasić, M. Barisits, V. Salzgeber, and D. Kostic. Making Cluster Applications Energy-Aware. In *Workshop on Automated Control for Datacenters and Clouds*, pages 37–42, New York, 2009.

[20] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. L. Reiher, and G. H. Kuenning. PARAID: A Gear-Shifting Power-Aware RAID. *TOS*, 2007.

Migratory Compression: Coarse-grained Data Reordering to Improve Compressibility

Xing Lin[1], Guanlin Lu[2], Fred Douglis[2], Philip Shilane[2], Grant Wallace[2]
[1]*University of Utah*, [2]*EMC Corporation – Data Protection and Availability Division*

Abstract

We propose *Migratory Compression* (MC), a coarse-grained data transformation, to improve the effectiveness of traditional compressors in modern storage systems. In MC, similar data chunks are re-located together, to improve compression factors. After decompression, migrated chunks return to their previous locations. We evaluate the compression effectiveness and overhead of MC, explore reorganization approaches on a variety of datasets, and present a prototype implementation of MC in a commercial deduplicating file system. We also compare MC to the more established technique of delta compression, which is significantly more complex to implement within file systems.

We find that Migratory Compression improves compression effectiveness compared to traditional compressors, by 11% to 105%, with relatively low impact on runtime performance. Frequently, adding MC to a relatively fast compressor like gzip results in compression that is more effective in both space and runtime than slower alternatives. In archival migration, MC improves gzip compression by 44–157%. Most importantly, MC can be implemented in broadly used, modern file systems.

1 Introduction

Compression is a class of data transformation techniques to represent information with fewer bits than its original form, by exploiting statistical redundancy. It is widely used in the storage hierarchy, such as compressed memory [24], compressed SSD caches [15], file systems [4] and backup storage systems [28]. Generally, there is a tradeoff between computation and compressibility: often much of the available compression in a dataset can be achieved with a small amount of computation, but more extensive computation (and memory) can result in better data reduction [7].

There are various methods to improve compressibility, which largely can be categorized as increasing the *look-back* window and *reordering data*. Most compression techniques find redundant strings within a window of data; the larger the window size, the greater the opportunity to find redundant strings, leading to better compression. However, to limit the overhead in finding redundancy, most real-world implementations use small window sizes. For example, DEFLATE, used by `gzip`, has a 64 KB sliding window [6] and the maximum window for `bzip2` is 900 KB [8]. The only compression algorithm we are aware of that uses larger window sizes is LZMA in `7z` [1], which supports history up to 1 GB.[1] It usually compresses better than `gzip` and `bzip2` but takes significantly longer. Some other compression tools such as `rzip` [23] find identical sequences over a long distance by computing hashes over fixed-sized blocks and then rolling hashes over blocks of that size throughout the file; this effectively does intra-file deduplication but cannot take advantage of small interspersed changes. Delta compression (DC) [9] can find small differences in similar locations between two highly similar files. While this enables highly efficient compression between similar files, it cannot delta-encode widely dispersed regions in a large file or set of files without targeted pair-wise matching of similar content [12].

Data reordering is another way to improve compression: since compression algorithms work by identifying repeated strings, one can improve compression by grouping similar characters together. The Burrows-Wheeler Transform (BWT) [5] is one such example that works on relatively small blocks of data: it permutes the order of the characters in a block, and if there are substrings that appear often, the transformed string will have single characters repeat in a row. BWT is interesting because the operation to invert the transformed block to obtain the original data requires only that an index be stored with the transformed data and that the transformed data be sorted lexicographically to use the index to identify the original contents. `bzip2` uses BWT as the second layer in its compression stack.

[1] The specification of LZMA supports windows up to 4 GB, but we have not found a practical implementation for Linux that supports more than 1 GB and use that number henceforth. One alternative compressor, xz [27], supports a window of 1.5 GB, but we found its decrease in throughput highly disproportionate to its increase in compression.

What we propose is, in a sense, a coarse-grained BWT over a large range (typically tens of GBs or more). We call it *Migratory Compression* (MC) because it tries to rearrange data to make it more compressible, while providing a mechanism to reverse the transformation after decompression. Unlike BWT, however, the unit of movement is kilobytes rather than characters and the scope of movement is an entire file or group of files. Also, the recipe to reconstruct the original data is a nontrivial size, though still only ~0.2% of the original file.

With MC, data is first partitioned into chunks. Then we 'sort' chunks so that similar chunks are grouped and located together. Duplicated chunks are removed and only the first appearance of that copy is stored. Standard compressors are then able to find repeated strings across adjacent chunks.[2] Thus MC is a *preprocessor* that can be combined with arbitrary adaptive lossless compressors such as `gzip`, `bzip2`, or `7z`; if someone invented a better compressor, MC could be integrated with it via simple scripting. We find that MC improves `gzip` by up to a factor of two on datasets with high rates of similarity (including duplicate content), usually with better performance. Frequently `gzip` with MC compresses both better and faster than other off-the-shelf compressors like `bzip2` and `7z` at their default levels.

We consider two principal use cases of MC:

mzip is a term for using MC to compress a single file. With `mzip`, we extract the resemblance information, cluster similar data, reorder data in the file, and compress the reordered file using an off-the-shelf compressor. The compressed file contains the recipe needed to restore the original contents after decompression. The bulk of our evaluation is in the context of stand-alone file compression; henceforth `mzip` refers to integrating MC with traditional compressors (gzip by default unless stated otherwise).

Archival involves data migration from backup storage systems to archive tiers, or data stored directly in an archive system such as Amazon Glacier [25]. Such data are cold and rarely read, so the penalty resulting from distributing a file across a storage system may be acceptable. We have prototyped MC in the context of the archival tier of the Data Domain File System (DDFS) [28].

There are two runtime overheads for MC. One is to detect similar chunks: this requires a preprocessing stage to compute *similarity features* for each chunk, followed by clustering chunks that share these features. The other overhead comes from the large number of I/Os necessary to reorganize the original data, first when performing compression and later to transform the uncompressed output back to its original contents. We quantify the effectiveness of using fixed-size or variable-size chunks, three chunk sizes (2 KB, 8 KB and 32 KB), and different numbers of features, which trade compression against runtime overhead. For the data movement overhead, we evaluate several approaches as well as the relative performance of hard disks and solid state storage.

In summary, our work makes the following contributions. First, we propose *Migratory Compression*, a new data transformation algorithm. Second, we evaluate its effectiveness with real-world datasets, quantify the overheads introduced and evaluate three data reorganization approaches with both HDDs and SSDs. Third, we compare `mzip` with DC and show that these two techniques are comparable, though with different implementation characteristics. Last, we demonstrate its effectiveness with an extensive evaluation of Migratory Compression during archival within a deduplicating storage system, DDFS.

2 Alternatives

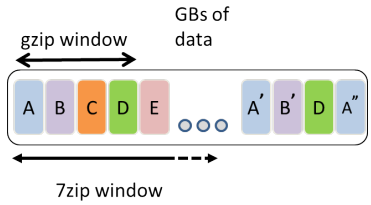
One goal of any compressor is to distill data into a minimal representation. Another is to perform this transformation with minimal resources (computation, memory, and I/O). These two goals are largely conflicting, in that additional resources typically result in better compression, though frequently with diminishing returns [7]. Here we consider two alternatives to spending extra resources for better compression: moving similar data together and delta-compressing similar data in place.

2.1 Migratory v Traditional Compression

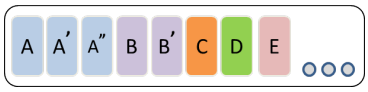
Figure 1 compares traditional compression and Migratory Compression. The blue chunks at the end of the file (*A'* and *A"*) are similar to the blue chunk at the start (*A*), but they have small changes that keep them from being entirely identical. With (a) traditional compression, there is a limited window over which the compressor will look for similar content, so *A'* and *A"* later in the file don't get compressed relative to *A*. With (b) MC, we move these chunks to be together, followed by two more similar chunks *B* and *B'*. Note that the two green chunks labeled *D* are identical rather than merely similar, so the second is replaced by a reference to the first.

One question is whether we could simply obtain extra compression by increasing the window size of a standard compressor. We see later (Section 5.4.4) that the "maximal" setting for `7z`, which uses a 1 GB lookback

[2] It is possible for an adaptive compressor's history to be smaller than size of two chunks, in which case it will not be able to take advantage of these adjacent chunks. For instance, if the chunks were 64 KB, gzip would not match the start of one chunk against the start of the next chunk. By making the chunk size small relative to the compressor's window size, we avoid such issues.



(a) Traditional compression.



(b) Migratory Compression.

Figure 1: Compression alternatives. With MC similar data moves close enough together to be identified as redundant, using the same compression window.

window (and a memory footprint over 10 GB) and substantial computation, often results in worse compression with poorer throughput than the default 7z setting integrated with MC.

2.2 Migratory v Delta Compression

Another obvious question is how MC compares to a similar technology, *delta compression* (DC) [9]. The premise of DC is to encode an object A' relative to a similar object A, and it is effectively the same as compressing A, discarding the output of that compression, and using the compressor state to continue compressing A'. Anything in A' that repeats content in A is replaced by a reference to its location in A, and content within A' that repeats previous content in A' can also be replaced with a reference.

When comparing MC and DC, there are striking similarities because both can use features to identify similar chunks. These features are compact (64 bytes per 8 KB chunk by default), allowing GBs or even TBs of data to be efficiently searched for similar chunks. Both techniques improve compression by taking advantage of redundancy between similar chunks: MC reads the chunks and writes them consecutively to aid standard compressors, while DC reads two similar chunks and encodes one relative to the other. We see in Section 5.3 that MC generally improves compression and has faster performance than intra-file DC, but these differences are rather small and could be related to internal implementation details.

One area where MC is clearly superior to DC is in its simplicity, which makes it compatible with numerous compressors and eases integration with storage systems. Within a storage system, MC is a nearly seamless addition since all of the content still exists after migration—it is simply at a different offset than before migration. For storage systems that support indirection, such as deduplicated storage [28], MC causes few architectural changes, though it likely increases fragmentation. On the other hand, DC introduces dependencies between data chunks that span the storage system: storage functionality has to be modified to handle indirections between delta compressed chunks and the *base* chunks against which they have been encoded [20]. Such modifications affect such system features as garbage collection, replication, and integrity checks.

3 Approach

Much of the focus of our work on Migratory Compression is in the context of reorganizing and compressing a single file (`mzip`), described in Section 3.1. In addition, we compare `mzip` to in-place delta-encoding of similar data (Section 3.2) and reorganization during migration to an archival tier within DDFS (Section 3.3).

3.1 Single-File Migratory Compression

The general idea of MC is to partition data into chunks and reorder them to store similar chunks sequentially, increasing compressors' opportunity to detect redundant strings and leading to better compression. For standalone file compression, this can be added as a pre-processing stage, which we term `mzip`. A reconstruction process is needed as a post-processing stage in order to restore the original file after decompression.

3.1.1 Similarity Detection with Super-features

The first step in MC is to partition the data into chunks. These chunks can be fixed size or variable size "content-defined chunks." Prior work suggests that in general variable-sized chunks provide a better opportunity to identify duplicate and similar data [12]; however, virtual machines use fixed-sized blocks, and deduplicating VM images potentially benefits from fixed-sized blocks [21]. We default to variable-sized chunks based on the comparison of fixed-sized and variable-sized units below.

One big challenge to doing MC is to identify similar chunks efficiently and scalably. A common practice is to generate *similarity features* for each chunk; two chunks are likely to be similar if they share many features. While it is possible to enumerate the closest matches by comparing all features, a useful approximation is to group sets of features into *super-features* (SFs): two data objects that have a single SF in common are likely to be fairly similar [3]. This approach has been used numerous times to successfully identify similar web pages, files, and/or chunks within files [10, 12, 20].

Because it is now well understood, we omit a detailed explanation of the use of SFs here. We adopt the "First-Fit" approach of Kulkarni, et al. [12], which we will term

the *greedy* matching algorithm. Each time a chunk is processed, its N SFs are looked up in N hash tables, one per SF. If any SF matches, the chunk is associated with the other chunks sharing that SF (i.e., it is added to a list and the search for matches terminates). If no SF matches, the chunk is inserted into each of the N hash tables so that future matches can be identified.

We explored other options, such as sorting all chunks on each of the SFs to look for chunks that match several SFs rather than just one. Across the datasets we analyzed, this sort marginally improved compression but the computational overhead was disproportionate. Note, however, that applying MC to a file that is so large that its metadata (fingerprints and SFs) is too large to process in memory would require some out-of-core method such as sorting.

3.1.2 Data Migration and Reconstruction

Given information about which chunks in a file are similar, our `mzip` preprocessor outputs two recipes: *migrate* and *restore*. The migrate recipe contains the chunk order of the reorganized file: chunks identified to be similar are located together, ordered by their offset within the original file. (That is, a later chunk is moved to be adjacent to the first chunk it is similar to.) The restore recipe contains the order of chunks in the reorganized file and is used to reconstruct the original file. Generally, the overhead of generating these recipes is orders of magnitude less than the the overhead of physically migrating the data stored in disk.

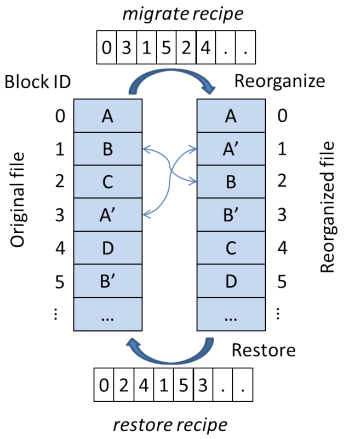


Figure 2: An example of the reorganization and restore procedures.

Figure 2 presents a simplified example of these two procedures, assuming fixed chunk sizes. We show a file with a sequence of chunks A through D, and including A' and B' to indicate chunks that are similar to A and B respectively. The reorganized file places A' after A and B' after B, so the *migrate* recipe specifies that the reorganized file consists of chunk 0 (A), chunk 3 (A'), chunk 1 (B), chunk 5 (B'), and so on. The *restore* recipe shows that to obtain the original order, we output chunk 0 (A), chunk 2 (B), chunk 4 (C), chunk 1 (A'), etc. from the reorganized file. (For variable-length chunks, the recipes contain byte offsets and lengths rather than block offsets.)

Once we have the migrate recipe and the restore recipe, we can create the reorganized file. Reorganization (migration) and reconstruction are complements of each other, each moving data from a specific location in an input file to a desired location in the output file. (There is a slight asymmetry resulting from deduplication, as completely identical chunks can be omitted completely in the reorganized file, then copied 1-to-N when reconstructing the original file.)

There are several methods for moving chunks.

In-Memory. When the original file can fit in memory, we can read in the whole file into memory and output chunks in the reorganized order sequentially. We call this the 'in-mem' approach.

Chunk-level. When we cannot fit the original file in memory, the simplest way to reorganize a file is to scan the chunk order in the migrate recipe: for every chunk needed, seek to the offset of that chunk in the original file, read it, and output it to the reorganized file. When using HDDs, this could become very inefficient because of the number of random I/Os involved.

Multi-pass. We also designed a 'multi-pass' algorithm, which scans the original file repeatedly from start to finish; during each pass, chunks in a particular *reorg range* of the reorganized file are saved in a memory buffer while others are discarded. At the end of each pass, chunks in the memory buffer are output to the reorganized file and the reorg range is moved forward. This approach replaces random I/Os with multiple scans of the original file. (Note that if the file fits in memory, the in-memory approach is the multi-pass approach with a single pass.)

Our experiments in Section 5.2 show that the in-memory approach is best, but when memory is insufficient, the multi-pass approach is more efficient than chunk-level. We can model the relative costs of the two approaches as follows. Let T be elapsed time, where T_{mp} is the time for multipass and T_c is the time when using individual chunks. Focusing only on I/O costs, T_{mp} is the time to read the entire file sequentially N times, where N is the number of passes over the data. If disk throughput is D and the file size is S, $T_{mp} = S * N/D$. For a size of 15GB, 3 passes, and 100MB/s throughput, this works

out to 7.7 minutes for I/O. If CS represents the chunk size, the number of chunk-level I/O operations is S/CS and the elapsed time is $T_c = \frac{S/CS}{IOPS}$. For a disk with 100 IOPS and an 8KB chunk size, this equals 5.4 hours. Of course there is some locality, so what fraction of I/Os must be sequential or cached for the chunk approach to break even with the multi-pass one? If we assume that the total cost for chunk-level is the cost of reading the file once sequentially[3] plus the cost of random I/Os, then we solve for the cost of the random fraction (RF) of I/Os equaling the cost of $N-1$ sequential reads of the file:

$$S*(N-1)/D = \frac{S*RF/CS}{IOPS}$$

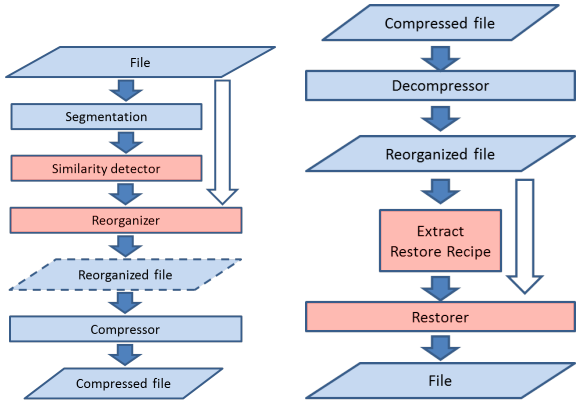
giving

$$RF = \frac{(N-1)*IOPS*CS}{D}.$$

In the example above, this works out to $\frac{16}{1024} = 1.6\%$; *i.e.*, if more than 1.6% of the data has dispersed similarity matches, then the multi-pass method should be preferred.

Solid-state disks, however, offer a good compromise. Using SSDs to avoid the penalty of random I/Os on HDDs causes the chunk approach to come closer to the in-memory performance.

3.1.3 `mzip` Workflow



(a) Compression (b) Decompression

Figure 3: Migratory Compression workflow.

Figure 3 presents the compression and decompression workflows in `mzip`. Compression/decompression and segmentation are adopted from existing tools, while similarity detection and reorganization/restoration are specially developed and highlighted in red. The original file is read once by the segmenter, computing cryptographically secure fingerprints (for deduplication) and resemblance features, then it is read again by the reorganizer to produce a file for compression. (This file may exist only as a pipeline between the reorganizer and the compressor, not separately stored, something we did for all compressors but `rzip` as it requires the ability to *seek*.) To restore the file, the compressed file is decompressed and its restore recipe is extracted from the beginning of the resulting file. Then the rest of that file is processed by the restorer, in conjunction with the recipe, to produce the original content.

[3]Note that if there are so many random I/Os that we do not read large sequential blocks, T_{mp} is reduced by a factor of $1-RF$.

3.2 Intra-file Delta Compression

When applied in the context of a single file, we hypothesized that `mzip` would be slightly better than delta compression (DC) because its compression state at the time the similar chunk is compressed includes content from many KBs-MBs of data, depending on the compressor. To evaluate how `mzip` compares with DC within a file, we implemented a version of DC that uses the same workflows as `mzip`, except the 'reorganizer' and the 'restorer' in `mzip` are replaced with a 'delta-encoder' and a 'delta-decoder.' The delta-encoder encodes each similar chunk as a delta against a base chunk while the delta-decoder reconstructs a chunk, by patching the delta to its base chunk. (In our implementation, the chunk earliest in the file is selected as the base for each group of similar chunks. We use `xdelta` [14] for encoding, relying on the later compression pass to compress anything that has not been removed as redundant.)

3.3 Migratory Compression in an Archival Storage System

In addition to reorganizing the content of individual files, MC is well suited for reducing data requirements within an entire file system. However, this impacts read locality, which is already an issue for deduplicating storage systems [13]. This performance penalty therefore makes it a good fit for systems with minimal requirements for read performance. An archival system, such as Amazon Glacier [25] is a prime use case, as much of its data will not be reread; when it is, significant delays can be expected. When the archival system is a tier within a backup environment, such that data moves in bulk at regular intervals, the data migration is an opportune time to migrate similar chunks together.

To validate the MC approach in a real storage system, we've implemented a prototype using the existing deduplicating Data Domain Filesystem (DDFS) [28]. After deduplication, chunks in DDFS are aggregated into *compression regions* (CRs), which in turn are aggregated into *containers*. DDFS can support two storage tiers: an *active* tier for backups and a long-term retention tier for *archival*; while the former stores the most recent data

within a time period (e.g., 90 days), the latter stores the relatively 'cold' data that needs to be retained for an extended time period (e.g., 5 years) before being deleted. Data migration is important for customers who weigh the dollar-per-GB cost over the migrate/retrieval performance for long-term data.

A daemon called *data migration* is used to migrate selected data periodically from the active tier to the archive tier. For performance reasons, data in the active tier is compressed with a simple LZ algorithm while we use `gzip` in the archive tier for better compression. Thus, for each file to be migrated in the namespace, DDFS reads out the corresponding compression regions from the active tier, uncompresses each, and recompresses with `gzip`.

The MC technique would offer customers a further tradeoff between the compression ratio and migrate/retrieval throughput. It works as follows:

Similarity Range. Similarity detection is limited to files migrated in one iteration, for instance all files written in a span of two weeks or 90 days.

Super-features. We use 12 similarity features, combined as 3 SFs. For each container to be migrated, we read out its metadata region, extract the SFs associated with each chunk, and write these to a file along with the chunk's fingerprint.

Clustering. Chunks are grouped in a similar fashion to the greedy single SF matching algorithm described in Section 3.1.1, but via sorting rather than a hash table.

Data reorganization. Similar chunks are written together by collecting them from the container set in multiple passes, similar to the single-file multi-pass approach described in Section 3.1.2 but without a strict ordering. Instead, the passes are selected by choosing the largest clusters of similar chunks in the first one-third, then smaller clusters, and finally dissimilar chunks. Since chunks are grouped by any of 3 SFs, we use 3 Bloom filters, respectively, to identify which chunks are desired in a pass. We then copy the chunks needed for a given pass into the CR designated for a given chunk's SF; the CR is flushed to disk if it reaches its maximum capacity.

Note that DDFS already has the notion of a mapping of a file identifier to a tree of chunk identifiers, and relocating a chunk does not affect the chunk tree associated with a file. Only the low-level index mapping a chunk fingerprint to a location in the storage system need be updated when a chunk is moved. Thus, there is no notion of a *restore recipe* in the DDFS case, only a recipe specifying which chunks to co-locate.

In theory, MC could be used in the backup tier as well as for archival: the same mechanism for grouping similar data could be used as a background task. However, the impact on data locality would not only impact read performance [13], it could degrade ingest performance during backups by breaking the assumptions underlying data locality: DDFS expects an access to the fingerprint index on disk to bring nearby entries into memory [28].

4 Methodology

We discuss evaluation metrics in Section 4.1, tunable parameters in Section 4.2, and datasets in Section 4.3.

4.1 Metrics

The high-level metrics by which to evaluate a compressor are the **compression factor** (CF) and the **resource usage** of the compressor. CF is the ratio of an original size to its compressed size, i.e higher CFs correspond to more data eliminated through compression; deduplication ratios are analogous.

In general, resource usage equates to processing time per unit of data, which can be thought of as the **throughput** of the compressor. There are other resources to consider, such as the required memory: in some systems memory is plentiful and even the roughly 10 GB of DRAM used by `7z` with its maximum 1 GB dictionary is fine; in some cases the amount of memory available or the amount of compression being done in parallel results in a smaller limit.

Evaluating the performance of a compressor is further complicated by the question of parallelization. Some compressors are inherently single-threaded while others support parallel threads. Generally, however, the fastest compression is also single-threaded (e.g., `gzip`), while a slower but more effective compressor such as `7z` is slower despite its multiple threads. We consider end-to-end time, not CPU time.

Most of our experiments were run inside a virtual machine, hosted by an ESX server with 2x6 Intel 2.67GHz Xeon X5650 cores, 96 GB memory, and 1-TB 3G SATA 7.2k 2.5in drives. The VM is allocated 90 GB memory except in cases when memory is explicitly limited, as well as 8 cores and a virtual disk with a 100 GB ext4 partition on a two-disk RAID-1 array. For 8 KB random accesses, we have measured 134 IOPS for reads (as well as 385 IOPS for writes, but we are not evaluating random writes), using a 70 GB file and an I/O queue depth of 1. For 128 KB sequential accesses, we measured 108 MB/s for reads and 80 MB/s for writes. The SSD used is a Samsung Pro 840, with 22K IOPS for random 8 KB reads and 264 MB/s for 128 KB sequential reads (write

Dataset		Size (GB)	Dedupe (X)	Compression Factor of Standalone Compressors (X)			
Type	Name			gzip	bzip2	7z	rzip
Workstation Backup	WS1	17.36	1.69	2.70	3.22	4.44	4.46
	WS2	15.73	1.77	2.32	2.61	3.16	3.12
Email Server Backup	EXCHANGE1	13.93	1.06	1.83	1.92	3.35	3.99
	EXCHANGE2	17.32	1.02	2.78	3.13	4.75	4.79
VM Image	Ubuntu-VM	6.98	1.51	3.90	4.26	6.71	6.69
	Fedora-VM	27.95	1.19	3.21	3.49	4.22	3.97

Table 1: Dataset summary: size, deduplication factor of 8 KB variable chunking and compression ratios of standalone compressors.

throughputs become very low because of no TRIM support in the hypervisor: 20 MB/s for 128 KB sequential writes). To minimize performance variation, all other virtual machines were shut down except those providing system services. Each experiment was repeated three times; we report averages. We don't plot error bars because the vast majority of experiments have a relative standard error under 5%; in a couple of cases, decompression timings vary with 10–15% relative error.

To compare the complexity of MC with other compression algorithms, we ran most experiments in-memory. In order to evaluate the extra I/O necessary when files do not fit in memory, some experiments limit memory size to 8 GB and use either an SSD or hard drive for I/O.

The tool that computes chunk fingerprints and features is written in C, while the tools that analyze that data to cluster similar chunks and reorganize the files are written in Perl. The various compressors are off-the-shelf Linux tools installed from repositories.

4.2 Parameters Explored

In addition to varying the workload by evaluating different datasets, we consider the effect of a number of parameters. Defaults are shown in **bold**.

Compressor. We consider **gzip**, `bzip2`, `7z`, and `rzip`, with or without MC.

Compression tuning. Each compressor can be run with a parameter that trades off performance against compressibility. We use the **default parameters** unless specified otherwise.

MC chunking. Are chunks fixed or **variable** sized?

MC chunk size. How large are chunks? We consider 2, **8**, and 32 KB; for variable-sized chunks, these represent target averages.

MC resemblance computation. How are super-features matched? (Default: **Four SFs**, matched greedily, one SF at a time.)

MC data source. When reorganizing an input file or reconstructing the original file after decompression, where is the input stored? We consider an **in-memory file system**, SSD, and hard disk.

4.3 Datasets

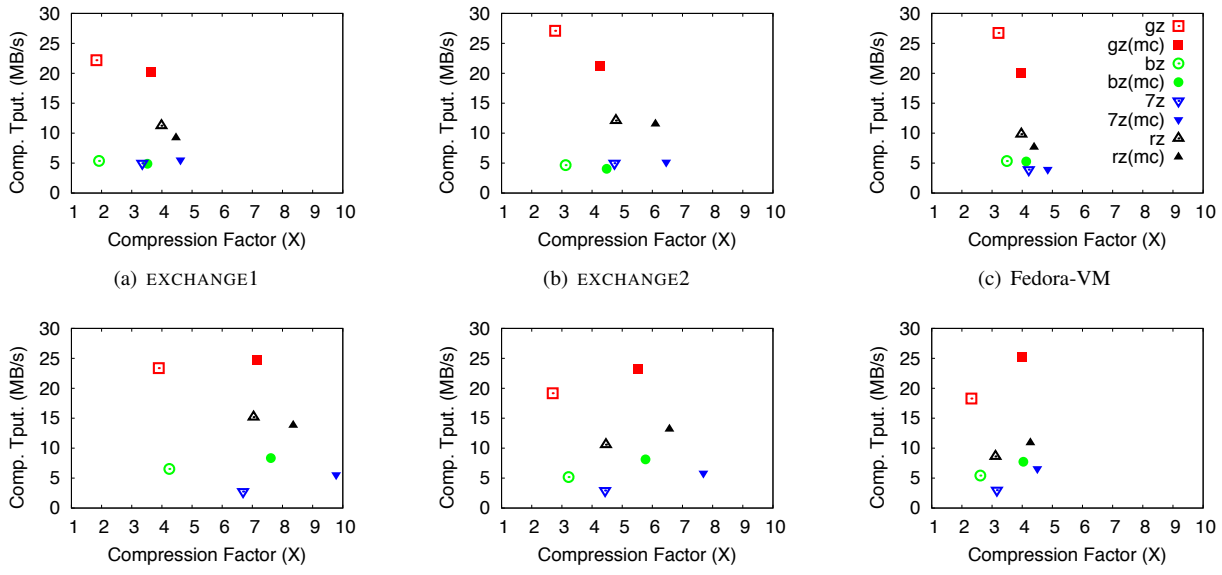
Table 1 summarizes salient characteristics of the input datasets used to test `mzip`, two types of backups and a pair of virtual machine images. Each entry shows the total size of the file processed, its deduplication ratio (half of them can significant boost their CF using MC simply through deduplication), and the CF of the four off-the-shelf compressors. We find that `7z` and `rzip` both compress significantly better than the others and are similar to each other.

- We use four single backup image files taken from production deduplication backup appliances. Two are backups of workstations while the other two are backups of Exchange email servers.

- We use two virtual machine disk images consisting of VMware VMDK files. One has Ubuntu 12.04.01 LTS installed while the other uses Fedora Core release 4 (a dated but stable build environment).

5 `mzip` Evaluation

The most important consideration in evaluating MC is whether the added effort to find and relocate similar content is justified by the improvement in compression. Section 5.1 compares CF and throughput across the six datasets. Section 5.2 looks specifically at the throughput when memory limitations force repeated accesses to disk and finds that SSDs would compensate for random I/O penalties. Section 5.3 compares `mzip` to a similar intra-file DC tool. Finally, Section 5.4 considers additional sensitivity to various parameters and configurations.



(a) EXCHANGE1 (b) EXCHANGE2 (c) Fedora-VM

(d) Ubuntu-VM (e) WS1 (f) WS2

Figure 4: Compression throughput vs. Compression Factor for all datasets, using unmodified compression or MC, for four compressors. The legend for all plots appears in (c).

5.1 Compression Effectiveness and Performance Tradeoff

Figure 4 plots compression throughput versus compression factor, using the six datasets. All I/O was done using an in-memory file system. Each plot shows eight points, four for the off-the-shelf compressors (`gzip`, `bzip2`, `7z`, and `rzip`) using default settings and four for these compressors using MC.

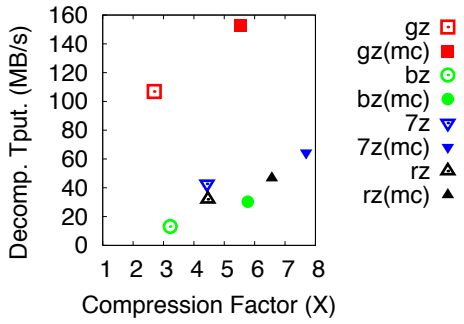
Generally, adding MC to a compressor significantly improves the CF (23–105% for `gzip`, 18–84% for `bzip2`, 15–74% for `7z` and 11–47% for `rzip`). It is unsurprising that `rzip` has the least improvement, since it already finds duplicate chunks across a range of a file, but MC further increases that range. Depending on the compressor and dataset, throughput may decrease moderately or it may actually improve as a result of the compressor getting (a) deduplicated and (b) more compressible input. We find that `7z` with MC always gets the highest CF, but often another compressor gets nearly the same compression with better throughput. We also note that in general, for these datasets, off-the-shelf `rzip` compresses just about as well as off-the-shelf `7z` but with much higher throughput. Better, though, the combination of `gzip` and MC has a comparable CF to any of the other compressors without MC, and with still higher throughput, making it a good choice for general use.

Decompression performance may be more important than compression performance for use cases where something is compressed once but uncompressed many times. Figure 5 shows decompression throughput versus CF for two representative datasets. For WS1, we see that adding MC to existing compressors tends to improve CF while significantly improving decompression throughput. It is likely because deduplication leads to less data to decompress. For EXCHANGE1, CF improves substantially as well, with throughput not greatly affected. Only for Fedora-VM (not shown) does `gzip` decompression throughput decrease in any significant fashion (from about 140 MB/s to 120).

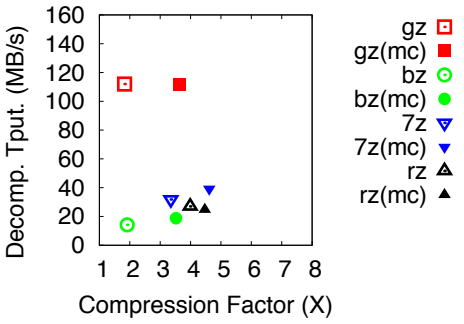
5.2 Data Reorganization Throughput

To evaluate how `mzip` may work when a file does not fit in memory, we experimented with a limit of 8 GB RAM when the input data is stored in either a solid state disk (SSD) or hard disk drive (HDD). The output file is stored in the HDD. When reading from HDD, we evaluated two approaches: chunk-level and multi-pass. Since SSD has no random-access penalty, we use only chunk-level and compare SSD to in-mem.

Figure 6 shows the compression throughputs for SSD-based and HDD-based `mzip`. (Henceforth `mzip` refers to `gzip` + MC.) We can see that SSD approaches in-memory performance, but as expected there is a significant reduction in throughput using the HDD. This reduction can be mitigated by the multipass approach. For instance, using a reorg range of 60% of memory, 4.81 GB, if the file does not fit in memory, the throughput can be improved significantly for HDD-based `mzip` by compar-



(a) WS1



(b) EXCHANGE1

Figure 5: Decompression throughput vs. Compression Factor for two representative datasets (WS1 and EXCHANGE1), using unmodified compression or MC.

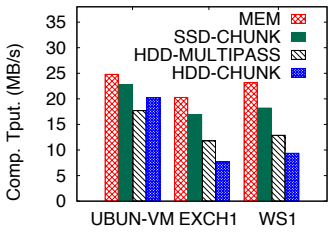


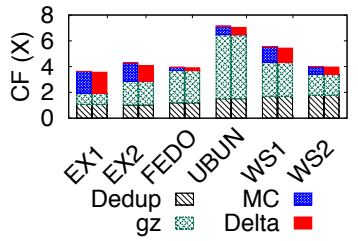
Figure 6: Compression throughput comparison for HDD-based or SSD-based `gzip` (MC).

ison to accessing each chunk in the order it appears in the reorganized file (and paying the corresponding costs of random I/Os).

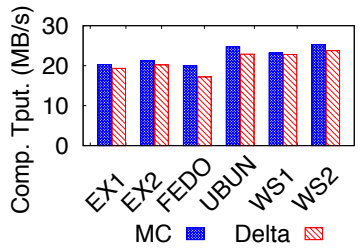
Note that Ubuntu-VM can approximately fit in available memory, so the chunk-level approach performs better than multi-pass: multi-pass reads the file sequentially twice, while chunk-level can use OS-level caching.

5.3 Delta Compression

Figure 7 compares the compression and performance achieved by `mzip` to compression using in-place delta-



(a) Compression Factor, by contributing technique



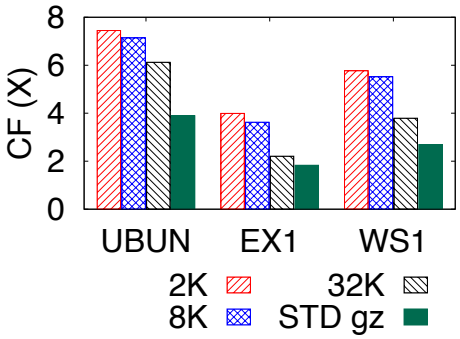
(b) Compression Throughput

Figure 7: Comparison between `mzip` and `gzip` (delta compression) in terms of compression factor and compression throughput. CFs are broken down by dedup and `gzip` (same for both), plus the additional benefit of either MC or DC.

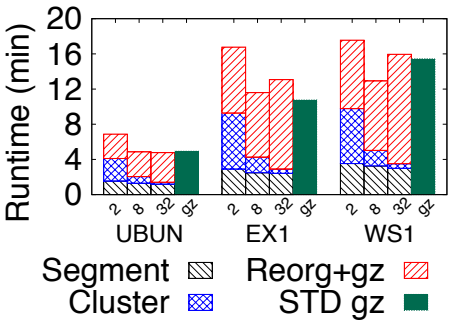
encoding,[4] as described in Section 3.2. Both use `gzip` as the final compressor. Figure 7(a) shows the CF for each dataset, broken out by the contribution of each technique. The bottom of each stacked bar shows the impact of deduplication (usually quite small but up to a factor of 1.8). The next part of each bar shows the *additional* contribution of `gzip` after deduplication has been applied, but with no reordering or delta-encoding. Note that these two components will be the same for each pair of bars. The top component is the additional benefit of either `mzip` or delta-encoding. `mzip` is always slightly better (from 0.81% to 4.89%) than deltas, but with either technique we can get additional compression beyond the gain from deduplication and traditional compression: $>$ 80% more for EXCHANGE1, $>$ 40% more for EXCHANGE2 and $>$ 25% more for WS1.

Figure 7(b) plots the compression throughput for `mzip` and DC, using an in-memory file system (we omit decompression due to space limitations). `mzip` is consistently faster than DC. For compression, `mzip` averages 7.21% higher throughput for these datasets. while for decompression `mzip` averages 29.35% higher throughput.

[4]Delta-encoding plus compression is delta compression. Some tools such as `vcdiff` [11] do both simultaneously, while our tool delta-encodes chunks and then compresses the entire file.

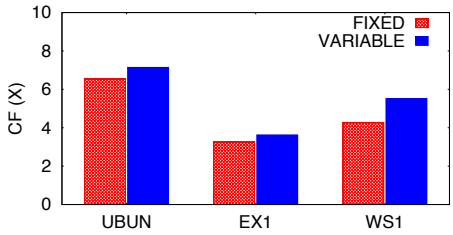


(a) Compression Factor

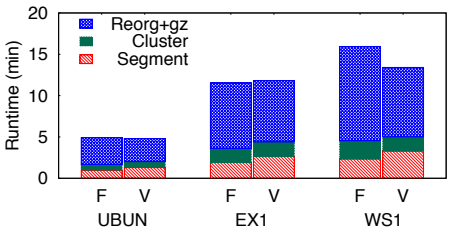


(b) Runtime, by component cost

Figure 8: Compression factor and runtime for `mzip`, varying chunk size.



(a) Compression Factor



(b) Runtime

Figure 9: Compression factor and runtime for `mzip`, when either fixed-size or variable-size chunking is used.

5.4 Sensitivity to Environment

The effectiveness and performance of MC depend on how it is used. We looked into various chunk sizes, compared fixed-size with variable-size chunking, evaluated the number of SFs to use in clustering and studied different compression levels and window sizes.

5.4.1 Chunk Size

Figure 8 plots `gzip`-MC (a) CF and (b) runtime as a function of chunk size (we show runtime to break down individual components by their contribution to the overall delay). We shrink and increase the default 8 KB chunk size by a factor of 4. Compression increases slightly in shrinking from 8 KB to 2 KB but decreases dramatically moving up to 32 KB. The improvement from the smaller chunksize is much less than seen when only deduplication is performed [26], because MC eliminates redundancy among similar chunks as well as identical ones. The reduction when increasing to 32 KB is due to a combination of fewer chunks to be detected as identical and similar and the small `gzip` lookback window: similar content in one chunk may not match content from the preceding chunk.

Figure 8(b) shows the runtime overhead, broken down by processing phase. The right bar for each dataset corresponds to standalone `gzip` without MC, and the remaining bars show the additive costs of segmentation, clustering, and the pipelined reorganization and compression. Generally performance is decreased by moving to a smaller chunk size, but interestingly in two of the three cases it is also worse when moving to a larger chunk size. We attribute the lower throughput to the poorer deduplication and compression achieved, which pushes more data through the system.

5.4.2 Chunking Algorithm

Data can be divided into fixed-sized or variable-sized blocks. For MC, supporting variable-sized chunks requires tracking individual byte offsets and sizes rather than simply block offsets. This increases the recipe sizes by about a factor of two, but because the recipes are small relative to the original file, the effect of this increase is limited. In addition, variable chunks result in better deduplication and matching than fixed, so CFs from using variable chunks are 14.5% higher than those using fixed chunks.

Figure 9 plots `mzip` compression for three datasets, when fixed-size or variable-size chunking is used. From Figure 9(a), we can see that variable-size chunking gives consistently better compression. Figure 9(b) shows that the overall performance of both approaches is comparable and sometimes variable-size chunking has better performance. Though variable-size chunking spends more time in the segmentation stage, the time to do compression can be reduced considerably when more chunks are duplicated or grouped together.

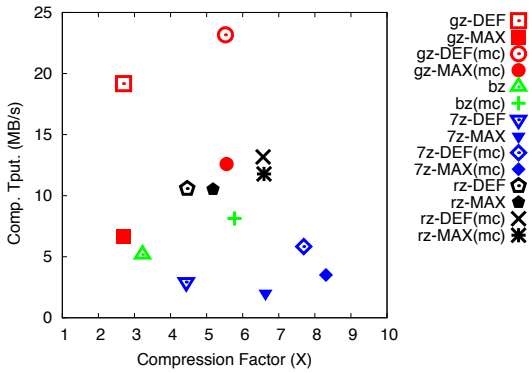


Figure 10: Comparison between the default and the maximum compression level, for standard compressors with and without MC, on the WS1 dataset.

5.4.3 Resemblance Computation

By default we use sixteen features, combined into four SFs, and a match on any SF is sufficient to indicate a match between two chunks. In fact most similar chunks are detected by using a single SF; however, considering three more SFs has little change in compression throughputs and sometimes improves compression factors greatly (e.g., a 13.6% improvement for EXCHANGE1). We therefore default to using 4 SFs.

5.4.4 Compression Window

For most of this paper we have focused on the default behavior of the three compressors we have been considering. For `gzip`, the "maximal" level makes only a small improvement in CF but with a significant drop in throughput, compared to the default. In the case of `bzip2`, the default is equivalent to the level that does the best compression, but overall execution time is still manageable, and lower levels do not change the results significantly. In the case of `7z`, there is an enormous difference between its default level and its maximal level: the maximal level generally gives a much higher CF with only a moderate drop in throughput. For `rzip`, we use an undocumented parameter "-L20"to increase the window to 2 GB; increasing the window beyond that had diminishing returns because of the increasingly coarse granularity of duplicate matching.

Figure 10 shows the compression throughput and CF for WS1 when the default or maximum level is used, for different compressors with and without MC. (The results are similar for other datasets.) From this figure, we can tell that maximal `gzip` reduces throughput without discernible effect on CF; `7z` without MC improves CF disproportionately to its impact on performance; and maximal `7z` (MC) moderately improves CF and reduces performance. More importantly, with MC and standard compressors, we can achieve higher CFs with much higher compression throughout than compressors' standard maximal level. For example, the open diamond marking `7z`-DEF(MC) is above and to the right of the close inverted triangle marking `7z`-MAX. Without MC, `rzip`'s maximal level improves performance with comparable throughput; with MC, `rzip` gets the same compression as `7z`-MAX with much better throughput, and `rzip`-MAX decreases that throughput without improving CF. The best compression comes from `7z`-MAX with MC, which also has better throughput than `7z`-MAX without MC.

6 Archival Migration in DDFS

In addition to using MC in the context of a single file, we can implement it in the file system layer. As an example, we evaluated MC in DDFS, running on a Linux-based backup appliance equipped with 8x2 Intel 2.53GHz Xeon E5540 cores and 72 GB memory. In our experiment, either the active tier or archive tier is backed by a disk array of 14 1-TB SATA disks. To minimize performance variation, no other workloads ran during the experiment.

6.1 Datasets

DDFS compresses each compression region using either LZ or `gzip`. Table 2 shows the characteristics of a few backup datasets using either form of compression. (Note that the WORKSTATIONS dataset is the union of several workstation backups, including WS1 and WS2, and all datasets are many backups rather than a single file as before.) The logical size refers to pre-deduplication data, and most datasets deduplicate substantially.

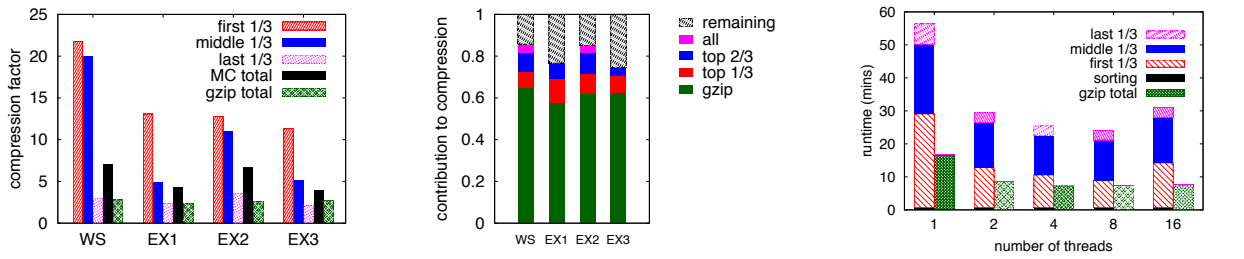
The table shows that `gzip` compression is 25–44% better than LZ on these datasets, hence DDFS uses `gzip` by default for archival. We therefore compare base `gzip` with `gzip` after MC preprocessing. For these datasets, we reorganize all backups together, which is comparable to an archive migration policy that migrates a few months at a time; if archival happened more frequently, the benefits would be reduced.

6.2 Results

Figure 11(a) depicts the compressibility of each dataset, including separate phases of data reorganization. As described in Section 3.3, we migrate data in thirds. The top third contains the biggest clusters and achieves the greatest compression. The middle third contain smaller clusters and may not compress quite as well, and the bottom third contains the smallest clusters, including clusters of a single chunk (nothing similar to combine it with). The next bar for each dataset shows the aggregate CF

Type	Name	Logical Size (GB)	Dedup. Size (GB)	Dedup. + LZ Size (GB)	LZ CF	Dedup. + gzip (GB)	gzip CF
Workstation	WORKSTATIONS	2471	454	230	1.97	160	2.84
Email Server	EXCHANGE1	570	51	27	1.89	22	2.37
	EXCHANGE2	718	630	305	2.07	241	2.61
	EXCHANGE3	596	216	103	2.10	81	2.67

Table 2: Datasets used for archival migration evaluation.



(a) CFs as a function of migration phase

(b) Fraction of data saved in each migration phase

(c) Durations, as a function of threads, for EXCHANGE1

Figure 11: Breakdown of the effect of migrating data, using just gzip or using MC in 3 phases.

using MC, while the right-most bar shows the compression achieved with gzip and no reorganization. Collectively, MC achieves 1.44–2.57× better compression than the gzip baseline. Specifically, MC outperforms gzip most (by 2.57×) on the workstations dataset, while it improves the least (by 1.44×) on EXCHANGE3.

Figure 11(b) provides a different view into the same data. Here, the cumulative fraction of data saved for each dataset is depicted, from bottom to top, normalized by the post-deduplicated dataset size. The greatest savings (about 60% of each dataset) come from simply doing gzip, shown in green. If we reorganize the top third of the clusters, we additionally save the fraction shown in red. By reorganizing the top two-thirds we include the fraction in blue; interestingly, in the case of WORKSTATIONS, the reduction achieved by MC in the middle third relative to gzip is higher than that of the top third, because gzip alone does not compress the middle third as well as it compresses the top. If we reorganize everything that matches other data, we may further improve compression, but only two datasets have a noticeable impact from the bottom third. Finally, the portion in gray at the top of each bar represents the data that remains after MC.

There are some costs to the increased compression. First, MC has a considerably higher memory footprint than the baseline: compared to gzip, the extra memory usage for reorganization buffers is 6 GB (128 KB compression regions * 48 K regions filled simultaneously). Second, there is run-time overhead to identify clusters of similar chunks and to copy and group the similar data. To understand what factors dominate the run-time overhead of MC, Figure 11(c) reports the elapsed time to copy the post-deduplication 51 GB EXCHANGE1 dataset to the archive tier, with and without MC, as a function of the number of threads (using a log scale). We see that multithreading significantly improves the processing time of each pass. We divide the container range into multiple subranges and copy the data chunks from each subrange into in-memory data reorganization buffers with multiple worker threads. As the threads increase from 1 to 16, the baseline (gzip) duration drops monotonically and is uniformly less than the MC execution time. On the other hand, MC achieves the minimum execution time with 8 worker threads; further increasing the thread count does not reduce execution time, an issue we attribute to intra-bucket serialization within hash table operations and increased I/O burstiness.

Reading the entire EXCHANGE1 dataset, there is a 30% performance degradation after MC compared to simply copying in the original containers. Such a read penalty would be unacceptable for primary storage, problematic for backup [13], but reasonable for archival data given lower performance expectations. But reading back just the final backup within the dataset is 7× slower than without reorganization, if all chunks are relocated whenever possible. Fortunately, there are potentially significant benefits to *partial* reorganization. The greatest compression gains are obtained by grouping the biggest clusters, so migrating only the top-third of clusters can provide high benefits at moderate cost. Interestingly, if just the top third of clusters are reorganized, there is only a 24% degradation reading the final backup.

7 Related Work

Compression is a well-trodden area of research. Adaptive compression, in which strings are matched against patterns found earlier in a data stream, dates back to the variants of Lempel-Ziv encoding [29, 30]. Much of the early work in compression was done in a resource-poor environment, with limited memory and computation, so the size of the adaptive dictionary was severely limited. Since then, there have been advances in both encoding algorithms and dictionary sizes, so for instance Pavlov's `7z` uses a "Lempel-Ziv-Markov-Chain" (LZMA) algorithm with a dictionary up to 1 GB [1]. With `rzip`, standard compression is combined with rolling block hashes to find large duplicate content, and larger lookahead windows decrease the granularity of duplicate detection [23].

The Burrows-Wheeler Transform (BWT), incorporated into `bzip2`, rearranges data—within a relatively small window—to make it more compressible [5]. This transform is reasonably efficient and easily reversed, but it is limited in what improvements it can effect.

Delta compression, described in Section 2.2, refers to compressing a data stream relative to some other known data [9]. With this technique, large files must normally be compared piecemeal, using subfiles that are identified on the fly using a heuristic to match data from the old and new files [11]. MC is similar to that sort of heuristic, except it permits deltas to be computed at the granularity of small chunks (such as 8 KB) rather than a sizable fraction of a file. It has been used for network transfers, such as updating changing Web pages over HTTP [16]. One can also deduplicate identical chunks in network transfers at various granularities [10, 17].

DC has also been used in the context of deduplicating systems. Deltas can be done at the level of individual chunks [20] or large units of MBs or more [2]. Fine-grained comparisons have a greater chance to identify similar chunks but require more state.

These techniques have limitations in the range of data over which compression will identify repeated sequences; even the 1 GB dictionary used by `7-zip` is small compared to many of today's files. There are other ways to find redundancy spread across large corpa. As one example, REBL performed fixed-sized or content-defined chunking and then used resemblance detection to decide which blocks or chunks should be delta-encoded [12]. Of the approaches described here, MC is logically the most similar to REBL, in that it breaks content into variable sized chunks and identifies similar chunks to compress together. The work on REBL only reported the savings of pair-wise DC on any chunks found to be similar, not the end-to-end algorithm and overhead to perform standalone compression and later reconstruct the original data. From the standpoint of *rearranging* data to make it more compressible, MC is most similar to BWT.

8 Future Work

We briefly mention two avenues of future work, application domains and performance tuning.

Compression is commonly used with networking when the cost of compression is offset by the bandwidth savings. Such compression can take the form of simple in-line coding, such as that built into modems many years ago, or it can be more sophisticated traffic shaping that incorporates delta-encoding against past data transmitted [19, 22]. Another point along the compression spectrum would be to use `mzip` to compress files prior to network transfer, either statically (done once and saved) or dynamically (when the cost of compression must be included in addition to network transfer and decompression). We conducted some initial experiments using `rpm` files for software distribution, finding that a small fraction of these files gained a significant benefit from `mzip`, but expanding the scope of this analysis to a wider range of data would be useful. Finally, it may be useful to combine `mzip` with other redundancy elimination protocols, such as content-based naming [18].

With regard to performance tuning, we have been gaining experience with MC in the context of the archival system. The tradeoffs between compression factors and performance, both during archival and upon later reads to an archived file, bear further analysis. In addition, it may be beneficial to perform small-scale MC in the context of the backup tier (rather than the archive tier), recognizing that the impact to read performance must be minimized. `mzip` also has potential performance improvements, such as multi-threading and reimplementing in a more efficient programming language.

9 Conclusions

Storage systems must optimize space consumption while remaining simple enough to implement. Migratory Compression reorders content, improving traditional compression by up to 2× with little impact on throughput and limited complexity. When compressing individual files, MC paired with a typical compressor (e.g., `gzip` or `7z`) provides a clear improvement. More importantly, MC delivers slightly better compression than delta-encoding without the added complexities of tracking dependencies (for decoding) between non-adjacent chunks. Migratory Compression can deliver significant additional consumption for broadly used file systems.

Acknowledgments

We acknowledge Nitin Garg for his initial suggestion of improving data compression by collocating similar content in the Data Domain File System. We thank Remzi Arpaci-Dusseau, Scott Auchmoody, Windsor Hsu, Stephen Manley, Harshad Parekh, Hugo Patterson, Robert Ricci, Hyong Shim, Stephen Smaldone, Andrew Tridgell, and Teng Xu for comments and feedback on earlier versions and/or the system. We especially thank our shepherd, Zheng Zhang, and the anonymous reviewers; their feedback and guidance have been especially helpful.

References

[1] 7-zip. `http://www.7-zip.org/`. Retrieved Sep. 7, 2013.

[2] ARONOVICH, L., ASHER, R., BACHMAT, E., BITNER, H., HIRSCH, M., AND KLEIN, S. T. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009).

[3] BRODER, A. Z. On the resemblance and containment of documents. In *In Compression and Complexity of Sequences (SEQUENCES97)* (1997), IEEE Computer Society.

[4] BURROWS, M., JERIAN, C., LAMPSON, B., AND MANN, T. On-line data compression in a log-structured file system. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems* (1992), ASPLOS V.

[5] BURROWS, M., AND WHEELER, D. J. A block-sorting lossless data compression algorithm. Tech. Rep. SRC-RR-124, Digital Equipment Corporation, 1994.

[6] DEUTSCH, P. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.

[7] FIALA, E. R., AND GREENE, D. H. Data compression with finite windows. *Communications of the ACM 32*, 4 (Apr. 1989), 490–505.

[8] GILCHRIST, J. Parallel data compression with bzip2. In *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems* (2004), vol. 16, pp. 559–564.

[9] HUNT, J. J., VO, K.-P., AND TICHY, W. F. Delta algorithms: an empirical analysis. *ACM Trans. Softw. Eng. Methodol. 7* (April 1998), 192–214.

[10] JAIN, N., DAHLIN, M., AND TEWARI, R. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *4th USENIX Conference on File and Storage Technologies* (2005).

[11] KORN, D. G., AND VO, K.-P. Engineering a differencing and compression data format. In *USENIX Annual Technical Conference* (2002).

[12] KULKARNI, P., DOUGLIS, F., LAVOIE, J., AND TRACEY, J. M. Redundancy elimination within large collections of files. In *USENIX 2004 Annual Technical Conference* (June 2004).

[13] LILLIBRIDGE, M., ESHGHI, K., AND BHAGWAT, D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *11th USENIX Conference on File and Storage Technologies* (Feb 2013).

[14] MACDONALD, J. File system support for delta compression. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[15] MAKATOS, T., KLONATOS, Y., MARAZAKIS, M., FLOURIS, M. D., AND BILAS, A. Using transparent compression to improve SSD-based I/O caches. In *Proceedings of the 5th European Conference on Computer Systems* (2010), EuroSys '10.

[16] MOGUL, J. C., DOUGLIS, F., FELDMANN, A., AND KRISHNAMURTHY, B. Potential benefits of delta encoding and data compression for http. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication* (1997), SIGCOMM '97.

[17] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001), SOSP '01.

[18] PARK, K., IHM, S., BOWMAN, M., AND PAI, V. S. Supporting practical content-addressable caching with CZIP compression. In *USENIX ATC* (2007).

[19] RIVERBED TECHNOLOGY. Wan Optimization (Steelhead). http://www.riverbed.com/products-solutions/products/wan-optimization-steelhead/, 2014. Retrieved Jan. 13, 2014.

[20] SHILANE, P., WALLACE, G., HUANG, M., AND HSU, W. Delta compressed and deduplicated storage using stream-informed locality. In *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems* (June 2012), USENIX Association.

[21] SMALDONE, S., WALLACE, G., AND HSU, W. Efficiently storing virtual machine backups. In *Proceedings of the 5th USENIX conference on Hot Topics in Storage and File Systems* (June 2013), USENIX Association.

[22] SPRING, N. T., AND WETHERALL, D. A protocol-independent technique for eliminating redundant network traffic. In *ACM SIGCOMM* (2000).

[23] TRIDGELL, A. *Efficient algorithms for sorting and synchronization*. PhD thesis, Australian National University Canberra, 1999.

[24] TUDUCE, I. C., AND GROSS, T. Adaptive main memory compression. In *USENIX 2005 Annual Technical Conference* (April 2005).

[25] VARIA, J., AND MATHEW, S. Overview of amazon web services, 2012.

[26] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *FAST'12: Proceedings of the 10th Conference on File and Storage Technologies* (2012).

[27] xz. `http://tukaani.org/xz/`. Retrieved Sep. 25, 2013.

[28] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies* (Feb 2008).

[29] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (May 1977), 337–343.

[30] ZIV, J., AND LEMPEL, A. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on 24*, 5 (1978), 530–536.

Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split

Wook-Hee Kim[†], Beomseok Nam[†], Dongil Park[‡], Youjip Won[‡]

[†] *Ulsan National Institute of Science and Technology, Korea*
{*okie90,bsnam*}*@unist.ac.kr*
[‡] *Hanyang University, Korea*
{*idoitlpg,yjwon*}*@hanyang.ac.kr*

Abstract

Misaligned interaction between SQLite and EXT4 of the Android I/O stack yields excessive random writes. In this work, we developed multi-version B-tree with lazy split (LS-MVBT) to effectively address the Journaling of Journal anomaly in Android I/O. LS-MVBT is carefully crafted to minimize the write traffic caused by fsync() call of SQLite. The contribution of LS-MVBT consists of two key elements: (i) Multi-version B-tree effectively reduces "the number of fsync() calls" via weaving the crash recovery information within the database itself instead of maintaining a separate file, and (ii) it significantly reduces "the number of dirty pages to be synchronized in a single `fsync()` call" via optimizing the multi-version B-tree for Android I/O. The optimization of multi-version B-tree consists of three elements: lazy split, metadata embedding, and disabling sibling redistribution. We implemented LS-MVBT in Samsung Galaxy S4 with Android 4.3 Jelly Bean. The results are impressive. For SQLite, the LS-MVBT exhibits 70% (704 insertions/sec vs. 416 insertions/sec), and 1,220% performance improvement against WAL mode and TRUNCATE mode (704 insertions/sec vs. 55 insertions/sec), respectively.

1 Introduction

In the era of mobile computing, smartphones and smart devices generate more network traffic than PCs [1]. It has been reported that 80% of the smartphones sold in the third quarter of 2013 are Android smartphones [2]. Despite the rapid proliferation of Android smartphones, the I/O stack of Android platform leaves much to be desired as it fails to fully leverage the maximum performance from hardware resources. Kim et al. [3] reported that in an Android device, storage I/O performance indeed has significant impact on the overall system performance although it has been believed that the slow storage performance should be masked due to even slower network subsystem. The poor storage performance mainly comes from the discrepancies in interaction between SQLite and EXT4.

SQLite is a serverless database engine that is used extensively in Android applications to persistently manage the data. SQLite maintains crash recovery information for a transaction in a separate file which is log for write-ahead logging (or rollback journal). In an SQLite transaction, every update in the log or rollback journal and actual updates in the database table are separately committed to the storage device via `fsync()` calls. In TRUNCATE[1] mode, a single insert operation of 100 byte record entails 2 `fsync()` calls and eventually generates 9 write operations (36 KB) to the storage device. 100 byte of database insert amplifies to over 36 KB when it reaches the storage device[4]. The main cause of this unexpected behavior is that EXT4 filesystem journals the journaling activity of SQLite through heavy-weight `fsync()` calls. This is called the Journaling of Journal anomaly [4].

There are several ways to resolve the Journaling of Journal anomaly. One way is to tune the I/O stack in OS layer, such as eliminating unnecessary metadata flushes and storing journal blocks on a separate block device [5]. Another way is to integrate the recovery information into the database file itself so that the database can be restored without an external journal file. *Multi-Version B-Tree (MVBT)* proposed by Becker et al. [6] is an example of the latter. The excessive I/O operations also cause other problems such as shortening the lifetime of NAND eMMC since NAND flash cells can only be erased or written to a limited number of times before they fail.

In this work, we dedicate our efforts on resolving the Journaling of Journal anomaly from which the Android I/O stack suffers. Journaling of Journal anomaly is caused by two reasons: the number of `fsync()` calls in an SQLite transaction and the overhead of a single `fsync()` call in EXT4. In order to reduce the number

[1] one of the journal modes in SQLite

of `fsync()` calls as well as the overhead of a single `fsync()` call, we developed a variant of Multi-version B-tree, *LS-MVBT (Lazy Split Multi-Version B-Tree)*. The contributions of this work are summarized as follows.

- **LS-MVBT** We resolve the Journaling of Journal anomaly with multi-version B-tree that weaves transaction recovery information into the database file itself instead of using separate rollback journal file or WAL log file.
- **Lazy split** LS-MVBT reduces the number of dirty pages flushed to the storage device when a B-tree node overflows. Our proposed *lazy split* algorithm minimizes the number of modified B-tree nodes by combining a historical dead node with one of its new split nodes.
- **Buffer reservation** LS-MVBT further reduces the chances of dirtying an extra node by padding some buffer space in lazy split nodes. If a lazy split node is accessed again and additional data items need to be stored, they are stored in reserved buffer space instead of splitting it.
- **Metadata embedding** LS-MVBT reduces the I/O traffic by not flushing the database header page to the storage device. Instead, our proposed metadata embedding method moves the file change counter metadata from database header page into the last modified B-tree node which should be flushed anyway.
- **Disabling sibling redistribution** Sibling redistribution (migration of overflown data into left and right sibling nodes) has been widely used in database systems, but we show that it significantly increases the number of dirty nodes. LS-MVBT prevents sibling redistribution to improve write performance at the cost of slightly slowing search performance.
- **Lazy garbage collection** Version-based data structures require garbage collection for dead entries. LS-MVBT reclaims dead entries of a B-tree node only when the node needs to be modified by a current write transaction. This lazy garbage collection does not increase the amount of data to be flushed, since it only cleans up dirty nodes.

We implemented LS-MVBT in one of the most recent smartphone models, Galaxy-S4. Our extensive experimental study shows that LS-MVBT exhibits 70% performance improvement against WAL mode and 1,220% improvement against TRUNCATE mode in SQLite transactions. WAL mode may suffer from long recovery latency for replaying the log. LS-MVBT outperforms WAL mode not only in terms of transaction performance, e.g., insertion/sec, but also in terms of recovery time. Our experiment shows recovery time in LS-MVBT is up to 440% faster than that in WAL mode.

The rest of the paper is organized as follows: In section 2, we discuss other research efforts related to the Android I/O stack and database recovery modes including multi-version B-trees. In section 3, we present how multi-version B-tree (MVBT) resolves the Journaling of Journal anomaly. In section 4, we present our design of a variant of MVBT, LS-MVBT (Lazy Split Multi-Version B-tree). In section 5, we propose further optimizations including metadata embedding, disabling sibling redistribution, and lazy garbage collection that reduce the number of dirty pages. Section 6 provides the performance results and analysis. In section 7, we conclude the paper.

2 Related Work

SQLite is a key component in the Android I/O stack which allows the applications to manage their data in a persistent manner [7]. In Android based smartphones, contrary to common perception, the major performance bottleneck is shown to be the storage device rather than the air-link [3], and the journaling activity is shown to be the dominant source of storage traffic [3, 4]. Lee et al. showed Android applications generate excessive amount of EXT4 journal I/O's, most of which are caused by SQLite [5]. The excessive I/O traffic is found to be caused by the misaligned interaction between the SQLite and EXT4 [4]. Jeong et al. improved the Android I/O stack by employing a set of optimizations, which include `fdatasync()` instead of `fsync()`, F2FS, external journaling, polling-based I/O, and WAL mode in SQLite instead of other journal modes. With these optimization methods, Jeong et. al achieved 300% improvement in SQLite performance without any hardware assistance [4].

Database recovery has been implemented in many different ways. While log-based recovery methods such as ARIES [8] are commonly used in many other server-based database management systems, rollback journal is used as the default atomic commit and rollback method in SQLite although WAL (Write-Ahead Logging) has become available since SQLite 3.7 [7].

In addition to the rollback journal and log-based recovery methods, many version-based atomic commit and rollback methods have been studied in the past. Version-based recovery methods integrate the recovery information into the database itself so that the database can be restored without an external journal file [6, 9, 10, 11]. Some examples include the *write-once balanced tree*

(WOBT) for indelible storage [9], version-based hashing method for accessing temporal data [12], and the *time-split B+-tree (TSBT)* [10] which is implemented in Microsoft SQL Server. Multi-version B+-tree (MVBT) proposed by Becker et al. [6] is designed to give a new unique version for each write operation. The version-based B-tree is proved to be asymptotically optimal in a sense that its time and space complexity are the same as those of the single-version B-tree. Becker's MVBT does not support multiple updates within a single transaction, but this drawback was overcome by *Transactional MVBT* which improved the MVBT by giving the same timestamp to the data items updated by the same transaction [13]. Our LS-MVBT is implemented based on the Transactional MVBT with several optimizations we propose in section 4.

The latest non-volatile semiconductor storage, such as NAND flash memory and STT-MRAM, sheds new light on the version-based atomic commit and rollback methods [14, 15]. Venkataraman et al. proposed a B-tree structure called CDDS (Consistent and Durable Data Structure) B-tree which is almost identical to MVBT except that it focuses on implementing multi-version information on non-volatile memory (NVRAM) [15]. For durability and consistency, CDDS uses a combination of `mfence` and `clflush` instructions to guarantee that memory writes are atomically flushed to NVRAM.

As write operations on flash memory systems have high latency, Li et al. developed FD-tree which is optimized for write operations on flash storage devices [16]. As the FD-tree needs a recovery scheme such as journaling or write-ahead-logging, the version-based recovery scheme can also be employed by FD-tree. If so, our proposed optimizations for multi-version B-tree can be employed on FD-tree as well.

Current database recovery schemes are based on the traditional two layers - volatile memory and non-volatile disks - but the advent of the NVRAM presents new challenges, i.e., write-ahead logging (WAL) causes some complications if the memory is non-volatile [17]. WAL recovery scheme is designed in a way that any update operation to a B-tree page has to be recorded in a permanent write-ahead-log file first while the dirty B-tree nodes stay in volatile memory. If a database node is also in permanent NVRAM, the logging is not "write-ahead". With NVRAM, the WAL scheme must be redesigned. An alternative solution is to use version-based recovery scheme for NVRAM as in CDDS B-tree. Lazy split, metadata embedding, and other optimizations that we propose in this work can be used to reduce the number of write operations even for CDDS B-tree.

3 Multi-Version B-tree

3.1 Journaling of Journal Anomaly in Android I/O

In the Android platform, `fsync()` call is triggered by the commit of an SQLite transaction. As the journaling activity of SQLite propagates expensive metadata update operations to the file systems, SQLite spends most of its insertion (or update) time on `fsync()` function call for journal and database files [4]. The issue of resolving Journaling of Journal anomaly boils down to two technical ingredients: (i) reducing the number of `fsync()` calls in an SQLite transaction and (ii) reducing the number of dirty pages which need to be synchronized to the storage in a single `fsync()` call. Both of these two constituents eventually aim at reducing the write traffic to the block device.

In rollback journal modes (DELETE, TRUNCATE, and PERSIST) of SQLite, a single transaction consists of two phases: database journaling and the database update. SQLite calls `fsync()` at the end of each phase to make the result of each phase persistent. In EXT4 with *ordered mode* journal, `fsync()` consists of two phases: (i) writing the updated data blocks to a file and (ii) committing the updated metadata for the respective file to the journal. Most database updates in a smartphone, e.g. inserting a schedule in the calendar, inserting a phone number in the address book, or writing a note in the Facebook timeline, are less than a few hundred bytes [5]. As a result, in the first phase of `fsync()`, the number of updated file blocks rarely goes beyond a single block (4 KB). In the second phase of `fsync()`, committing a journal transaction to the filesystem journal entails four or more `write` operations, including journal descriptor, group descriptor, block bitmap, inode table, and journal commit mark, to the storage. Each of these entries corresponds to a single filesystem block.

In an effort to reduce the number of `fsync()` calls in an SQLite transaction, we implemented version-based B-tree, *multi-version* B-tree by Becker et al. [6], which maintains update history within the B-tree itself instead of maintaining it in a separate rollback journal file (or log file). This saves SQLite one or more `fsync()` calls.

3.2 Multi-Version B-Tree

In multi-version B-tree (MVBT), each insert, delete, or update transaction increases "the most recent consistent version" in the header page of a B-tree. Each key-value pair stored in MVBT defines its own life span - $[version_{start}, version_{end})$. When a key-value pair is inserted with a new version v, the life span of the new key-value pair is set to $[v, \infty)$. When a key-value pair

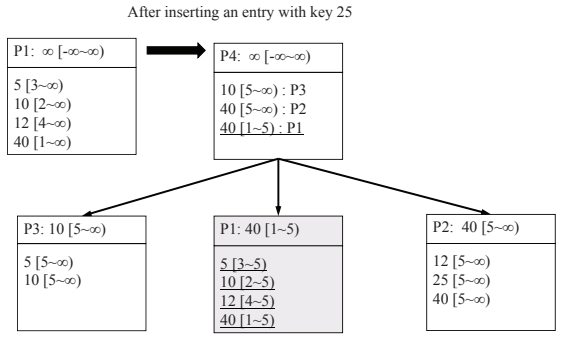


Figure 1: Multi-Version B-Tree split: After inserting an entry with key 25 into MVBT, three new nodes are created.

is deleted at version v, its life span is set to - $[v_{old}, v)$. Update transaction creates a new cell entry that has the transaction's version as its starting version $[v, \infty)$ and the old cell entry updates its valid end version to the previous consistent version number $[v_{old}, v)$. The key-value pair whose $version_{end}$ of life span is not ∞ is called a *dead* entry. The one with infinite life span is called a *live* entry. In multi-version B-trees, the search operation is trivial. A read transaction first examines the latest consistent version number and uses it to find valid entries in B-tree nodes, i.e., if a version of a read transaction is not within the life span of a key-value pair, the respective data is ignored by the read transaction.

If a node overflows, the entries in the overflown node are distributed into two newly allocated nodes, which is referred to as "node split". An additional new node is then allocated as a parent node or an existing parent node is updated with the two newly created nodes. The life spans of the two new nodes are set to $[v, \infty)$. An overflown node becomes dead via setting the node's version range $[v_{old}, \infty)$ to $[v_{old}, v)$. In summary, a single node split creates at least four dirty nodes in version-based B-tree structures. (Please refer to [6] and [15] for more detailed discussions on the insertion and split algorithms of version-based B-tree.). In the commit phase of a transaction, SQLite writes dirty nodes in the B-tree using the `write()` system call and triggers `fsync()` to make the result of the `write()` persistent.

Figure 1 shows how an MVBT splits a node when it overflows. Suppose a B-tree node can hold at most four entries in the example. When a new entry with key 25 is inserted by a transaction whose version is 5, the node `P1` splits and a half of the live entries are copied to a new node, `P2`, and the other half of the live entries are copied to another new node, `P3`. The previous node `P1` now becomes a *dead node* and it becomes available only for the transactions whose versions are older (smaller) than 5. The two new nodes should be pointed by a parent node and the version range of the dead node should also be updated in the parent node. In the example, a new root node, `P4`, is created and the pointers to the three child nodes are stored.

The recovery in multi-version B-tree is simple and straightforward. Multi-version B-tree maintains the version numbers of currently outstanding transactions at the storage. In current SQLite, there can be at most one outstanding write transaction for a given B-tree [7]. In the recovery phase, the recovery module first reconstructs the multi-version B-tree in memory from the storage and determines the version number of aborted transaction. Then, it scans all the nodes and adjusts the life span of each cell entry to obliterate the effect of aborted transaction. The life span which ends at v, i.e., $[v_{old}, v)$, is revoked to $[v_{old}, \infty)$ and all cell entries which start at v are deleted.

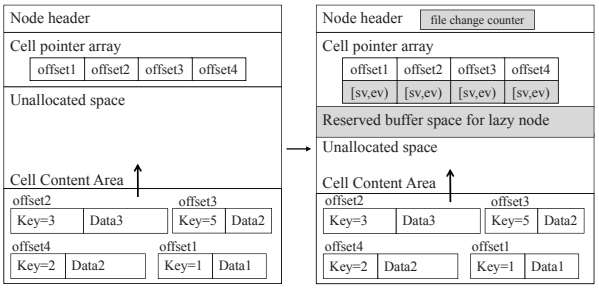
The recent eMMC controllers generate error correction code for 4 KB or 8 KB page, hence multi-version B-tree can rely on `fsync()` to atomically move from one consistent state to the next in the unit of page size. Even if the eMMC controller promises that only single sector writes are atomic and the B-tree node size is a multiple of the sector size, multi-version B-tree guarantees correct recovery as it creates a new key-value pair with new version information instead of overwriting previous key-value pairs. A multi-version B-tree node can be considered a combination of B-tree node and journal.

4 Lazy Split Multi-version B-Trees

MVBT successfully reduces the number of `fsync()` calls in an SQLite transaction as it eliminates the journaling activity of SQLite. Our next effort is dedicated to minimizing the overhead of a single `fsync()` call in MVBT. The essence of the optimization is to minimize the number of dirty nodes which are flushed to the disk as a result of a single SQLite transaction.

4.1 Multi-Version B-Tree Node in SQLite

We modified the B-tree node structure of SQLite and implemented a multi-version B-tree. Figure 2 shows the layout of an SQLite B-tree node which consists of two area: (i) *cell content area* that holds key-value pairs and (ii) *cell pointer array* which contains the array of pointers (offsets) each of which points to the actual key-value pair. Cell pointer array is sorted in key order. In the modified B-tree node structure, each key-value pair defines its own life span - $[version_{start}, version_{end})$, illustrated as $[sv, ev)$. The augmentation with start and end version number is universal across all the version-based B-tree structures [6, 15]. In our MVBT node design, we set



(a) Layout of SQLite B-Tree Node (b) Modified Layout of SQLite B-Tree Node

Figure 2: In modified Multi-Version B-Tree node, each key-value pair is tagged with its valid starting version and ending version.

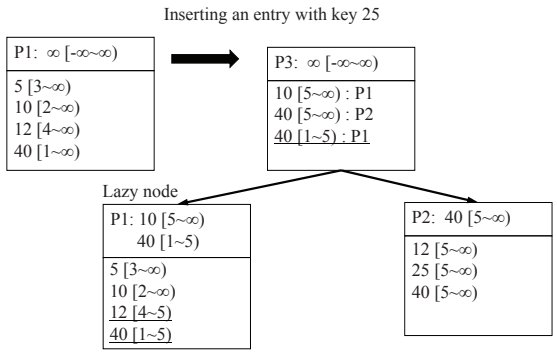


Figure 3: LS-MVBT: With the lazy split, an overflown node creates a single sibling node.

Algorithm 1
Lazy Split Algorithm

procedure
LazySplit$(n, parent, v)$

```
1: // n is an overflown B-tree node.
2: // parent is the parent node.
3: // v is the version of a current transaction.
4: newNode ← allocateNewBtreeNode()
5: Find the median key value k to split
6: for i ← 0, n.numCells − 1 do
7:     if k < n.cell[i].key ∧ v ≤ n.cell[i].endVersion then
8:         n.cell[i].endVersion ← v
9:         newNode.insert(n.cell[i])
10:        n.liveCells − −
11:    end if
12: end for
13: // Update the parent with the split key and version
14: maxLiveKey ← findMaxLiveKey(n, v)
15: parent.update(n, maxLiveKey, ∞)
16: maxDeadKey ← findMaxDeadKey(n, v)
17: parent.insert(n, maxDeadKey, v)
18: maxLiveKey2 ← findMaxLiveKey(newNode, v)
19: parent.insert(newNode, maxLiveKey2, ∞)
```

end procedure

aside a small fraction of bytes in the header of each node for *lazy split* and *metadata embedding* improvement.

4.2 Lazy Split

We develop an alternative split algorithm, *Lazy Split*, for MVBT that significantly reduces the number of dirty pages.

In MVBT, a single node split operation results in at least four dirty B-tree nodes as shown in Figure 1. The objective of maintaining a separate dead node in MVBT is to make garbage collection and recovery simple. On the other hand, creating a separate dead node yields an additional dirty page which needs to be flushed to disk. Unlike in other client/server databases, rollback operations do not occur frequently in SQLite, because SQLite allows only one process at a time to have write permission to a database file [7], and rollback operations of a version-based B-tree are already very simple. Therefore, we argue that benefit of creating a separate dead node in the legacy split algorithm of MVBT hardly offsets the additional performance overhead during `fsync()` that it induces.

Algorithm 1 shows our *lazy split* algorithm that postpones marking an overflown node as dead, if possible. Instead of creating an extra dead node, lazy split algorithm combines a dead node with a live sibling node. I.e., the *lazy node* is a half dead node combined with one of the new split nodes. In the lazy split algorithm, the overflown node creates only one new sibling node. Once the median key value to split is determined, the key-value pairs whose keys are greater than the median value are copied to the new sibling node as live entries. In the overflown node, the end versions of the copied key-value pairs are changed from ∞ to the current transaction's version in order to mark them as dead entries. In the original MVBT, the key-value pairs whose keys are smaller than the median key value are copied to another new left sibling node, but lazy split algorithm does not create the left sibling node and does not change the end versions of the smaller half of the key-value pairs.

Figure 3 shows an example of lazy split. When key 25 is inserted into node `P1`, the greater half of the key-value pairs (key 12 and key 40) are moved to a new node, `P2`, and they are marked dead in `P1`. Instead of creating another new node and moving the smaller half of the key-value pairs to it, lazy split algorithm keeps them in the overflown node. The dead entries in the lazy node will be garbage collected by the next write transaction that modifies the lazy node. Note that the lazy node has two pointers pointing to it in its parent node: one for the dead entries and the other for the live entries. The same insert operation in the original MVBT will create a left sibling

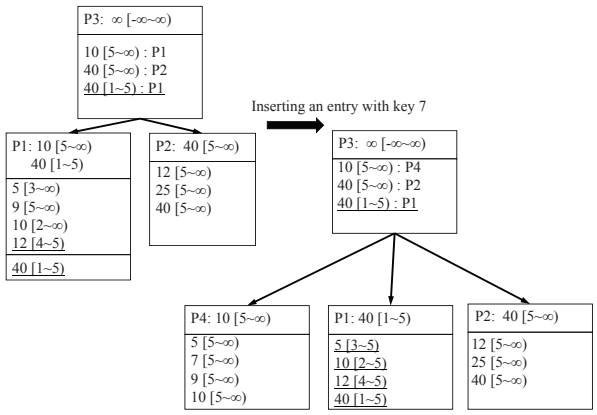


Figure 4: A new entry with key 9 is inserted into an overflown lazy node but its dead entries can not be deleted because transaction 5 is the current transaction and it may abort later. In this case, the reserved space can be used to hold the new entries and delay the node split again. But if the same transaction inserts an entry with key 7, the reserved space of the lazy node also overflows and we do not have any other option but to create a new left sibling node P4 and move the live entries ($5[5,\infty)$, $7[5,\infty)$, $9[5,\infty)$, and $10[5,\infty)$) to P4.

node, store the key 5 and key 10 in the left sibling node, and mark the two key-value entries dead in the historic dead node as shown in Figure 1. In the example, the valid version ranges of key 5 and key 10 are partitioned in the two nodes. This redundancy does not help anything especially when we consider the short lifespan of SQLite transactions. The dead entries are not needed by any subsequent write transactions and thus can be safely garbage collected in the next modification of the lazy node because a write transaction holds an exclusive lock for the database file. The legacy split algorithm of MVBT creates four dirty nodes but lazy split decreases the number of dirty nodes by one, creating only three dirty nodes.

4.3 Reserved Buffer Space for Lazy Split

The *lazy node* does not have any space left for additional data items to be inserted after the split. If an inserted key is greater than the median key value and is stored in a new node as in Figure 1, the lazy split succeeds. However, if a new inserted item needs to be stored in the lazy node, a new sibling node must be created as in the original MVBT split algorithm. In order to avoid splitting a lazy node, we reserve a certain amount of space in a LS-MVBT node to accommodate the inserted key in the lazy split node as shown in Figure 4.

To avoid cascade split, the size of the reserved buffer space should be sufficiently large to accommodate the newly inserted entries by a transaction. However, reserving too much space for buffer will make node utilization low and may entail more frequent node split creating larger amount of dirty pages. The size of the reserved buffer space needs to be carefully determined considering the workload characteristics. In smartphone applications, most write transactions do not insert more than one data item. Therefore, it is unlikely that an overflown node (lazy node) is accessed multiple times by a single write transaction.

In order to evaluate the effect of the reserved buffer space size, we ran experiments varying the sizes of reserved buffer space. Large reserved buffer space is only beneficial when a single transaction inserts a large number of entries into the same B-tree node. However, a large buffer space did not significantly reduce the number of dirty nodes in our experiments, but it hurt tree node utilization especially when the B-tree node size was small. In smartphone applications, it is very common that a transaction inserts just a single data item, hence we set the size of the buffer space just large enough to hold only one key-value item throughout the presented experiments in this paper. Even if reserved buffer space for one key-value item is used, a subsequent write transaction that finds the dead entries in the lazy node will reclaim the dead entries and create empty spaces.

Algorithm 2
Rollback Algorithm

procedure
Rollback(n, v)

```
1:  // n is a B-tree node
2:  // v is the version of aborted transaction
3:  for i ← 0, n.numCells − 1 do
4:      if n.cell[i].startVersion == v then
5:          remove n.cell[i]
6:          if n is an internal node then
7:              freeNode(n.child[i], v)
8:              continue
9:          end if
10:         deleteEntry(n.child[i])
11:     else if n.cell[i].endVersion == v then
12:         n.cell[i].endVersion ← ∞
13:         if n is an internal node then
14:             Delete a median key entry k that was used to split
                the lazy node.
15:         end if
16:     end if
17:     Rollback(n.child[i], v)
18: end for
end procedure
```

4.4 Rollback with Lazy Node

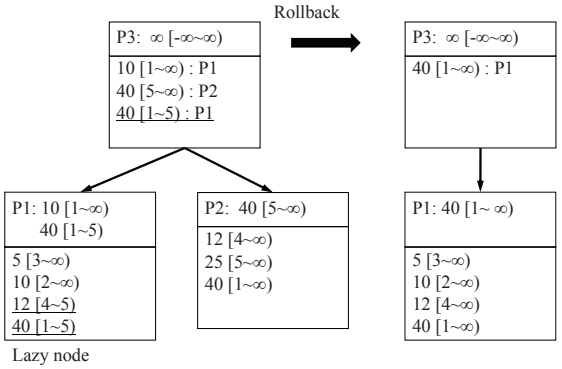


Figure 5: Rollback of transaction version 5 deletes node P2, reverts the end version of dead entries from 5 to ∞, and merges the entries in the parent node.

The rollback algorithm for the LS-MVBT is intuitive and simple. More importantly, as in the lazy split algorithm, the number of dirty nodes touched by the rollback algorithm of LS-MVBT is smaller than that of MVBT. Algorithm 2 shows the pseudo code of the LS-MVBT rollback algorithm. When a transaction aborts and rolls back, the LS-MVBT reverts its B-tree structures back to their previous states by reverting the end versions of the lazy nodes back to ∞ and deleting entries whose start versions are the aborted transaction's version. In the parent node, the lazy node has two entries: one for live entries and the other for dead entries. The parent entry of the live entries should be deleted from the parent node and the parent entry for the dead entries should be updated with its previous end version, ∞, to become active.

Figure 5 shows a rollback example. Note that node `P2` was created by a transaction whose version is 5, thus `P2` should be deleted. Since all the live entries in `P2` were copied from the lazy node `P1` by a transaction whose version is 5 and `P1` has historical entries, `P2` can be safely removed. The dead entries in `P1` should be reverted back to live entries by modifying the end versions. As the lazy node has two parent entries, the rollback process merges them and reverts back to the previous status by choosing the larger key value and by merging the valid version ranges.

5 Optimizing LS-MVBT for Android I/O

5.1 Lazy Garbage Collection

In multi-version B-trees, garbage collection mechanism is needed as dead entries must be garbage-collected to create empty spaces and to decrease the size of the trees. While a periodic garbage collector that sweeps the entire B-tree is commonly used in version-based B-trees [18, 15], we implemented *lazy garbage collection* scheme in SQLite in order to avoid making extra B-tree nodes dirty and to reduce the overhead of `fsync()`.

When a B-tree node needs to be modified, lazy garbage collection scheme checks if the node contains any dead entries whose versions are not needed by an active transaction. If so, the dead entries can be safely deleted. The dead entries in a B-tree node will be reclaimed only when a new live entry is modified or is added to the node. Since the node will become dirty anyway by the live entry, our lazy garbage collection does not increase the number of dirty nodes at all.

5.2 Metadata Embedding

In SQLite, the first page of a database file (header page) is used to store metadata about the database such as B-tree node size, list of free pages, file change counter, etc. The file change counter in header page is used for concurrency control in SQLite.[2] When multiple processes are accessing a database file concurrently, each process can detect if other processes have changed the database file by monitoring the file change counter. However, this concurrency control design of SQLite induces significant overhead on I/O traffic since the header page must be flushed just to update 4 bytes of file change counter for every write transaction. This results in a large performance gap between WAL mode and the other journal modes in SQLite (DELETE, TRUNCATE, and PERSIST) since WAL mode does not use the file change counter.

In this work, we devised a method called "Metadata Embedding" to reduce the overhead of flushing database header page. In metadata embedding, we maintain the database header page at the RAM disk so that the most recent consistent and valid version ("file change counter") in the database header page is shared by transactions and the database header page is exempt from being flushed to the storage in every `fsync()` call. Since the RAM disk is volatile, the file change counter in the RAM disk can be lost. Therefore, in metadata embedding, we let the most recent file change counter be flushed along with the last modified B-tree node. When a transaction starts, it reads the database header page at the RAM disk to access the file change counter. When a write transaction modifies the database table, it increases the file change counter and flushes it to the database header page at the RAM disk and to the last modified B-tree node. Since the last modified B-tree node has to be flushed to the storage anyway, metadata embedding makes the modified file change counter persistent without extra overhead.

[2]The race condition is handled by file system lock (`fcntl()`) in SQLite.

When a system recovers, the entire multi-version B-tree has to be scanned by a recovery process. Therefore, it is not a problem to find the largest valid consistent version number in the database and use it to rollback some changes made to the database file. If other parts of the header page are changed, we flush the header page as normal. Note that other parts of the header page are modified much less frequently than the file change counter.

5.3 Disabling Sibling Redistribution

Another optimization method used in LS-MVBT to reduce the I/O traffic is disabling redistribution of data entries between sibling nodes. If a B-tree node overflows in SQLite (and in many other server-based database engines), it redistributes its data entries to left and right sibling nodes. This is to avoid node split which requires allocation of additional nodes and changes in the tree organization. This redistribution modifies four nodes - two sibling nodes, the overflown node, and its parent node. In general, it is well known in the database community that sibling redistribution improves the node utilization, keeps the tree height short, and makes search operation faster, but we observed that it significantly hurt the write performance in the Android I/O stack.

In flash memory, time to write a page (page program latency) is 10 times longer than the time to read a page (read latency)[19] and subsequently, from SQLite's point of view, database updates, e.g., insert, update, and delete, take much longer than database search. Furthermore, search operations in smartphones are not as dominant as in client/server enterprise databases. Given these facts, we devise an approach opposite to the conventional wisdom: we disable sibling redistribution. In LS-MVBT, if a node overflows, we do not attempt to redistribute the entries in the overflown node to its siblings. Instead, LS-MVBT immediately triggers a lazy split operation.

6 Evaluation

We implemented the lazy split multi-version B-tree in SQLite 3.7.12. In this section, we evaluate and analyze the performance of the LS-MVBT compared to other traditional journal modes and WAL mode. Our testbed is *Samsung Galaxy-S4* that runs Android OS 4.3 (Jelly Bean) on Exynos 5 Octa Core 5410 1.6GHz CPU, 2GB DDR2 memory, and 16GB eMMC flash memory formatted with EXT4 file systems.

Many latest smartphones, including Samsung Galaxy S4, adjust the CPU frequency in order to save the power consumption. We fixed the frequency to the maximum 1.6 GHz so as to reduce the standard deviation of the experiments.

The evaluation section flows as follows. First, we examine the performance of SQLite transaction (`insert`) under three different SQLite modes: LS-MVBT, WAL mode, which is the default in Jelly Bean, and TRUNCATE mode, which is the default mode in Ice Cream Sandwich. Second, we take a detailed look at the block I/O behavior of SQLite transaction for LS-MVBT and WAL. Third, we observe how the versioning nature of LS-MVBT affects the search performance via examining the SQLite performance under varying mixture of `search` and `insert/delete` transactions. Fourth, we examine the recovery overhead of LS-MVBT and WAL. The final segment of the evaluation section is dedicated to quantifying the performance gain of each of the optimization techniques proposed in this paper, which are lazy split, metadata embedding, and disabling sibling redistribution, in an itemized as well as in an aggregate manner.

6.1 Workload Characteristics

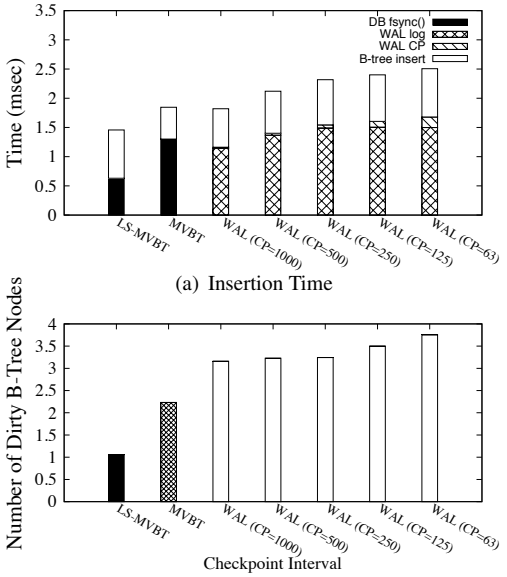
To accurately capture the workload characteristics of the smartphone apps, we extracted the database information from Gmail, Facebook, and Dolphin web browser apps in a testbed smartphone. Out of 136 tables in the device, the largest table contains about 4,500 records, and only 15 tables have more than 1,000 records. It is very common for smartphone apps to have such small number of records in a single database table unlike enterprise server/client databases. As most tables have less than thousands of records, we focused on evaluating the performance of LS-MVBT with rather small database tables. As for the reserved buffer space of LS-MVBT, we fix it to one cell for all the presented experiments.

6.2 Analysis of `insert` Performance

In evaluating the SQLite transaction performance, we focus on `insert` since `insert`, `update`, and `delete` generate similar amount of I/O traffic and show similar performances.

For the first set of experiments, we initialize a table with 2,000 records and submit 1,000 transactions, each of which inserts and deletes a random key value pair[3]. In WAL mode, checkpoint interval directly affects the transaction performance as well as recovery latency: with longer checkpoint interval, the transaction performance improves but the recovery latency gets longer. In SQLite, the default checkpoint interval is when 1,000 pages become dirty. The default interval can be changed by a pragma statement or a compile-time option. Checkpoint also occurs when *.db file is closed. If an app opens

[3]The performance of sequential key insertion/deletion is not very different from the presented results.



(a) Insertion Time

(b) Number of Dirty B-Tree Nodes per Transaction

Figure 6: Insertion Performance of LS-MVBT, MVBT, and WAL with Varying Checkpointing Interval (Avg. of 5 runs)

and closes a database file often, WAL mode will perform checkpointing operations frequently. For the comprehensiveness of the study, we vary the checkpoint intervals to 63, 125, 250, 500 and 1,000 pages. We first examine the time for a single `insert` transaction. For a fair comparison, the average insertion time in WAL mode includes the amortized average checkpointing overhead.

Figure 6(a) illustrates the result. Insertion time of MVBT and LS-MVBT consists of two elements: (i) the time to manipulate the database which is essentially an operation of updating the page content in memory, *B-tree insert*, and (ii) the time to `fsync()` the dirty pages, *DB fsync()*. Insertion time of WAL mode consists of three elements: (i) the time to manipulate the database, *B-tree insert*, (ii) the time to commit the log to storage, *WAL log*, and (iii) the time for checkpointing, *WAL CP*.

The average insertion time of LS-MVBT (1.4 ms) is up to 78% faster than that of WAL mode (2.0∼2.5 ms), but the insertion time of the original MVBT is no better than that of WAL mode. Throughout the various checkpointing intervals, LS-MVBT consistently outperforms WAL mode (even without including the checkpointing overhead). There is another important benefit of using LS-MVBT. In WAL mode, according to our measurement, the average elapsed time for each checkpoint is 7.6∼9.2 msec which is ×3 the average `insert` latency. Therefore, in WAL mode, the transactions that trigger checkpointing suffer from sudden increases in the latency. LS-MVBT outperforms WAL in terms of average query response time as well as in terms of the worst case bound.

We examine the number of dirty B-tree nodes per `insert` in MVBT, LS-MVBT, and WAL mode (Figure 6(b)). The number of dirty B-tree nodes in LS-MVBT is significantly lower than WAL mode. For an `insert`, LS-MVBT makes just one B-tree node dirty on average while WAL mode generates three or more dirty B-tree nodes. In WAL mode, not all dirty B-tree nodes are flushed to storage, but `fsync()` is called for log file commit, and the dirty nodes are flushed by the next checkpointing.

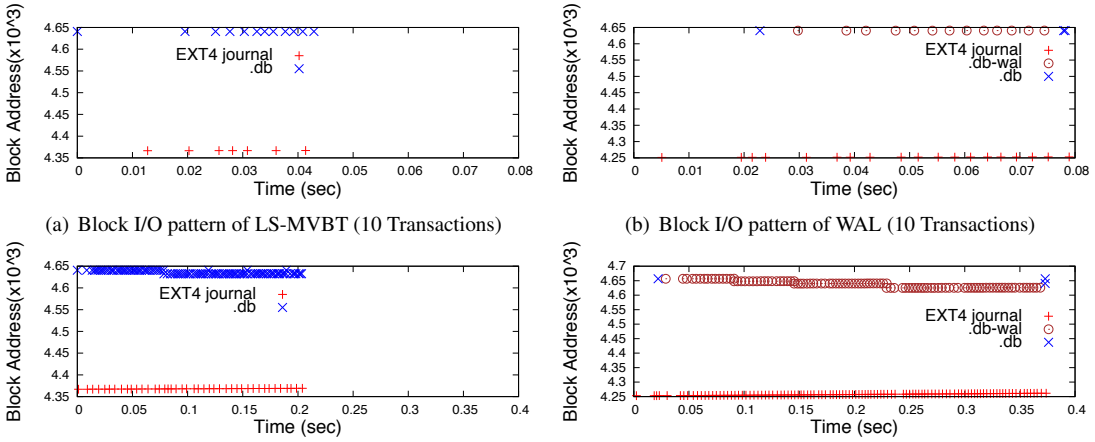
An interesting observation from Figure 6 is that the insertion performance gap between LS-MVBT and WAL is significant (40%) even when the checkpointing interval is set to 1,000 pages. When the checkpoint interval is 63 pages, the average transaction response time of WAL (2.5 msec) is 78% higher than that of LS-MVBT.

6.3 Analysis of Block I/O Behavior

For more detailed understanding, we delve into the block I/O behaviors of SQLite transactions in LS-MVBT and WAL mode. Figure 7 shows block I/O traces of an insert operation in LS-MVBT and WAL mode. Let us first examine the detailed block I/Os in LS-MVBT. When an `fsync()` is called, the updated database file contents are written to the disk. Then, the updated metadata for the file is committed to EXT4 journal. For a single insert transaction, one 4 KB block is written to the disk for file update. Three 4 KB blocks are written to EXT4 journal, which correspond to journal descriptor header, metadata, and journal commit mark. In WAL mode, 8 KB blocks are written to the disk for log file update. Eight 4 KB blocks are written to EXT4 journal. If checkpointing occurs, there will be more accesses to a block device.

Figure 7(a) and 7(b) show the number of accesses to a block device when 10 insert transactions are submitted. Interestingly, the total number of block device accesses for 10 insert transactions in WAL mode is 84% higher than that in LS-MVBT. However, with 100 insert transactions, the number of block device accesses in WAL mode is only 46% higher than that in LS-MVBT as shown in Figure 7(c) and 7(d). In LS-MVBT, the number of block device accesses increases linearly with the increased number of insertions whereas WAL mode accesses block devices less frequently when the size of batch insert transaction is larger.

Since WAL mode writes more data than LS-MVBT per each block device access, we measure the amount of I/O traffic caused in every 10 msec. Figure 8 shows the block access I/O traffic for LS-MVBT and WAL mode. For the experiment we submit 1,000 insert transactions and measure how many blocks are accessed per every



(a) Block I/O pattern of LS-MVBT (10 Transactions)

(b) Block I/O pattern of WAL (10 Transactions)

(c) Block I/O pattern of LS-MVBT (100 Transactions)

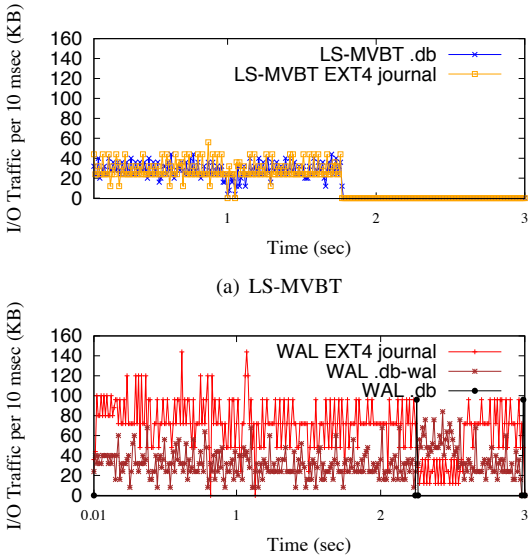
(d) Block I/O pattern of WAL (100 Transactions)

Figure 7: Block Trace of Insert SQLite Operation: LS-MVBT vs WAL

10 milliseconds. The block access I/O traffic per 10 milliseconds for LS-MVBT fluctuates between 24 KB to 40 KB, and the EXT4 journal blocks are accessed about 24∼44 KB per 10 milliseconds. In WAL mode, the database file blocks are accessed only three times: when the database file is opened, when checkpointing occurs in 2.25 seconds, and when the database file is closed.

When the checkpointing occurs at 2.25 seconds, the I/O traffic for WAL log file increases by approximately 20 KB, from 40 KB to 60 KB, but it decreases to 40 KB when the checkpointing finishes at 2.6 seconds. In WAL mode, the number of accesses to the EXT4 journal blocks is consistently higher than any other block access types, which explains why WAL mode shows poor insertion performance. We are currently investigating what causes this high number of EXT4 journal accesses in WAL mode.

In summary, LS-MVBT accesses 9.9 MB (5 MB EXT4 journal blocks and 4.9 MB database file blocks) in just 1.8 seconds, while WAL accesses 31 MB blocks (20.7 MB EXT4 journal blocks, 9.764 MB WAL log blocks, and only 0.9 MB database file blocks) in 3 seconds.



(a) LS-MVBT

(b) WAL

Figure 8: I/O Traffic at Block Device Driver Level (1,000 insertions)

6.4 Search Overhead

LS-MVBT makes the insert/update/delete queries faster at the cost of slow search performance. In LS-MVBT, node access has to check its children's version information in addition to the key range. Moreover, LS-MVBT does not perform sibling redistribution which results in poor node utilization. Lee et al. [5] reported that write operations are dominant in smartphone applications, and the SQL traces that we extracted from our testbed device confirm this. However, the search and the write ratio can depend on individual user's smartphone usage pattern, hence we examine the effectiveness of LS-MVBT with varying the ratio of search and write transactions. We initialize a database table with 1,000 records, and submit a total of 1,000 transactions with varying ratios between the number of `insert/delete` and `search` transactions. Each `insert/delete` transaction inserts and deletes a random data from the database table, and the `search` transaction searches a random data from the table. For notational simplicity, we term `insert/delete` as `write`.

Figure 9 illustrates the result. We examine the throughput under three different SQLite implementations: LS-MVBT, WAL mode, and TRUNCATE mode.

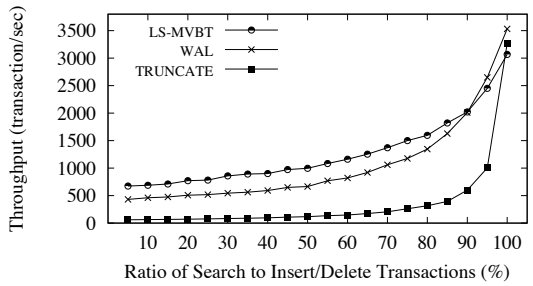


Figure 9: Mixed Workload (Search:Insert) Performance (Avg. of 5 runs)

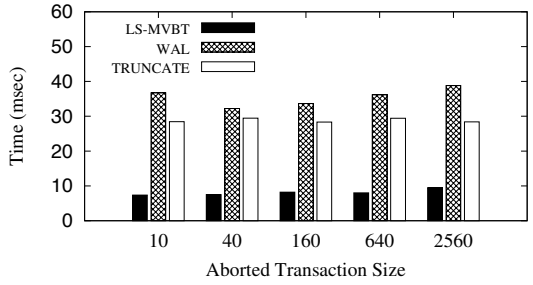
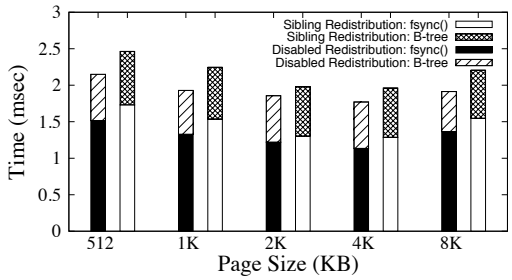
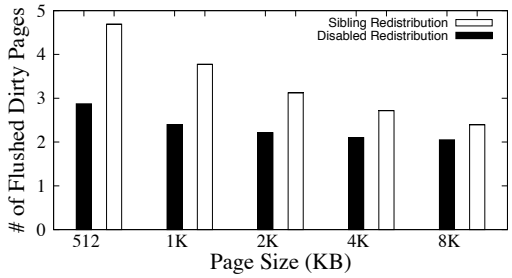


Figure 10: Recovery Time with Varying Size of Aborted Transaction



(a) Insertion Time (With vs. Without Redistribution)



(b) Number of Dirty B-tree Nodes (With vs. Without Redistribution)

Figure 11: *The average elapsed time and the number of flushed dirty nodes per insertion. (Average of 1,000 insertions): Rebalancing data entries hurts write performance when a node splits.*

As we increase the ratio of search transactions, the overall throughput increases because a search operation is much faster than a write operation. As long as at least 7% of the transactions are writes, LS-MVBT outperforms both WAL and TRUNCATE modes. In LS-MVBT, the performance gain on write operations far outweighs the performance penalty on search operations. This is mainly due to asymmetry in latencies of writing and reading a page in NAND flash memory: writing a page may take up to 9 times longer than reading a page [19].

6.5 Recovery Overhead

Recovery latency is one of the key elements that govern the effectiveness of a crash recovery scheme. While WAL mode exhibits superior SQLite performance against the other three journal modes, i.e., DELETE, TRUNCATE, and PERSIST, it suffers from longer recovery latency. This is because in WAL mode, the log records in the WAL file need to be replayed to reconstruct the database. In this section, we examine the recovery latencies of TRUNCATE, WAL, and LS-MVBT under varying number of outstanding (or aborted equivalently) insert statements in an aborted transaction at the time of crash: 10, 40, 160, 640, and 2560.

Figure 10 illustrates the recovery latencies of LS-MVBT, WAL, and TRUNCATE. When the aborted transaction inserts less than 10 records, WAL mode recovery takes about 4∼5 times longer than LS-MVBT. As the transaction size grows from 10 insertions to 2,560 insertions, WAL recovery mode suffers from a larger number of write I/Os and its recovery time increases by 20%. LS-MVBT recovery mode also increases by 28% but from much shorter recovery time. TRUNCATE mode recovery time slightly increases, by only 3%, but its recovery time is already 3.9 times longer than LS-MVBT when the transaction size is just 10. LS-MVBT needs to read the entire B-tree nodes for recovery but it only updates the nodes that should rollback to a consistent version.

6.6 Performance Effect of Optimizations

In order to quantify the performance effect of the optimizations made on MVBT, we first examine the effect of sibling redistribution in SQLite B-tree implementation by enabling and disabling the sibling redistribution. We use the average insertion time and the average number of dirty B-tree nodes for each insertion as performance metrics in Figure 11. We insert 1,000 records of 128 bytes into an empty table, and vary the node sizes of B-tree in SQLite from 512 bytes to 8 KB.

Figure 11(a) shows the average insertion time when sibling redistribution is *enabled* and *disabled*. When sib-

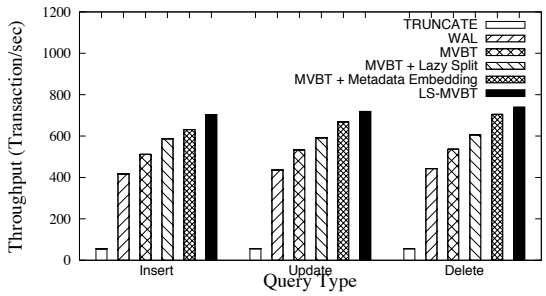


Figure 12: Performance Improvement Quantification (Avg. of 5 runs)

ling redistribution is disabled, insertion time decreases as much as 20%. In the original B-tree, 70% of the insertion time is spent on `fsync()` and most of the improvement comes from the reduction in `fsync()` overhead. Figure 11(b) shows the average number of dirty B-tree nodes per a single `insert` transaction. With 1 KB node size, the number of dirty pages in an `insert` is reduced from 3.7 pages to 2.4 pages if sibling redistribution is disabled. Since metadata embedding can save another dirty page, with disabled sibling redistribution and metadata embedding, the average number of dirty B-tree nodes per a single insertion transaction can drop down to fewer than 2 nodes, i.e., approximately 50% of disk page flush can be saved.

With a larger node size, the number of dirty B-tree nodes decreases because node overflow occurs less often. However, we observe that the elapsed `fsync()` time grows with larger node sizes (4 KB and 8 KB) since the size of nodes that need to be flushed increases, and also the time spent in B-tree insertion code increases because more computation is required for larger tree entries. After examining the effect of B-tree node size on insert performance (Figure 11), we determine that 4 KB node size yields the best performance. In all experiments in this study, B-tree node size is set to 4 KB. [4]

6.7 Putting Everything Together

It is time to put everything together and examine real world implications. In Figure 12, we compare the performance of the multi-version B-trees with different combinations of the optimizations for three different types of SQL queries. The performances are measured in terms of transaction throughput (number of transactions/sec). *MVBT* denotes the multi-version B-Tree with disabled sibling redistribution. *MVBT + Metadata Embedding* denotes the multi-version B-tree with metadata embedding optimization and disabled sibling redistribution. *MVBT + Lazy Split* is the multi-version B-tree with lazy split algorithm and disabled sibling redistribution. Finally, *LS-MVBT* denotes the multi-version B-tree with metadata embedding, lazy split algorithm, and disabled sibling redistribution. All three schemes employ lazy garbage collection and use one reserved buffer cell for lazy split. We compare these variants of multi-version B-trees with TRUNCATE journal mode and WAL mode.

TRUNCATE mode yields the worst performance (60 ins/sec), which is well aligned with previously reported results [4]. Via merely changing the SQLite journal mode to WAL, we increase the query processing throughput (insertions/sec) to 416 ins/sec. Via weaving the crash recovery information into the B-tree, which eliminates the need for a separate journal (or log) file, and via disabling sibling redistribution, we achieve 20% performance gain against WAL mode. Via augmenting metadata embedding in MVBT, we achieve 50% performance gain against WAL mode.

Combining all the optimizations we propose together, (metadata embedding, lazy split, and disabling sibling redistribution), we are able to achieve 70% performance gain in an existing smartphone without any hardware assistance.

7 Conclusion

In this work, we show that lazy split multi-version B-tree (LS-MVBT) can resolve the Journaling of Journal anomaly by avoiding expensive external rollback journal I/O. LS-MVBT minimizes the number of dirty pages and reduces the Android I/O traffic via *lazy split*, *reserved buffer space*, *metadata embedding*, *disabling sibling redistribution*, and *lazy garbage collection* schemes.

The optimizations we propose exploit the unique characteristics of Android I/O subsystem: (i) write is much slower than read in the Flash based storage, (ii) dominant fraction of storage accesses are write, and (iii) there are no concurrent write accesses to database.

By reducing the underlying I/O traffic of SQLite, the lazy split multi-version B-trees (LS-MVBT) consistently outperforms TRUNCATE rollback journal mode and WAL mode in terms of write transaction throughput.

One future direction of this work is to improve LS-MVBT in order to support multiple concurrent write transactions. With the presented versioning scheme, modifications to B-tree nodes should be made in commit order. As multicore chipsets are widely used in recent smartphones, the need for concurrent write transactions would increase and multi-version B-tree should be improved to fully support concurrent write transactions.

[4] With 4 KB of node size, an internal tree page of SQLite can hold at most 292 key-child cells when the key is integer type while the maximum number of entries in leaf node is dependent on the record size.

Acknowledgement

We would like to thank our shepherd Raju Rangaswami and the anonymous reviewers for their insight and suggestions on early drafts of this paper. This research was supported by MKE/KEIT (No.10041608, Embedded System Software for New Memory based Smart Devices).

References

[1] M. Meeker, "KPCB Internet trends year-end update, Kleiner Perkins Caufield & Byers," Dec 2012.

[2] "Market share analysis: Mobile phones, worldwide, 3q13," *http://www.gartner.com/document/2622821*.

[3] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2013.

[4] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," in *Proceedings of the USENIX Annual Technical Conference (ATC 2013)*, 2013.

[5] K. Lee and Y. Won, "Smart layers and dumb result: Io characterization of an android-based smartphone," in *Proceedings of the 12th International Conference on Embedded Software (EMSOFT 2012)*, 2012.

[6] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion B-tree," *VLDB Journal*, vol. 5, no. 4, pp. 264–275, Dec. 1996.

[7] "Sqlite," *http://www.sqlite.org/*.

[8] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems*, vol. 17, no. 1, 1992.

[9] M. C. Easton, "Key-sequence data sets on indelible storage," *IBM Journal of Research and Development*, vol. 30, no. 3, pp. 230–241, May 1986.

[10] D. Lomet and B. Saltzberg, "Access methods for multiversion data," in *Proceedings of 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1989.

[11] P. J. Varman and R. M. Verma, "An efficient multiversion access structure," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 3, pp. 391–409, 1997.

[12] G. Kollios and V. Tsotras, "Hashing methods for temporal data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 4, pp. 902–919, 2002.

[13] T. Haapasalo, I. Jaluta, B. Seeger, S. Sippu, and E. Soisalon-Soininen, "Transactions on the multiversion B+-tree," in *the 12th International Conference on Extending Database Technology (EDBT '09)*, 2009.

[14] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger, "Metadata efficiency in versioning file systems," in *Proceedings of the 2nd USENIX conference on File and Storage Technologies (FAST)*, 2003, pp. 43–58.

[15] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

[16] Y. Li, B. He, Q. Luo, and K. Yi, "Tree indexing on flash disks," in *Proceedings of the 25th International Conference on Data Engineering (ICDE)*, 2009.

[17] G. Graefe, "A survey of B-tree logging and recovery techniques," *ACM Transactions on Database Systems*, vol. 37, no. 1, Feb. 2012.

[18] B. Sowell, W. Golab, and M. A. Shah, "Minuet: A scalable distributed multiversion b-tree," in *Proceedings of the VLDB Endowment, Vol. 5, No. 9*, 2012.

[19] G. Wu and X. He, "Reducing ssd read latency via nand flash program and erase suspension," in *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, 2012.

Journaling of Journal Is (Almost) Free

Kai Shen Stan Park* Meng Zhu
University of Rochester

Abstract

Lightweight databases and key-value stores manage the consistency and reliability of their own data, often through rollback-recovery journaling or write-ahead logging. They further rely on file system journaling to protect the file system structure and metadata. Such *journaling of journal* appears to violate the classic end-to-end argument for optimal database design. In practice, we observe a significant cost (up to 73% slowdown) by adding the Ext4 file system journaling to the SQLite database on a Google Nexus 7 tablet running a Ubuntu Linux installation. The cost of file system journaling is up to 58% on a conventional machine with an Intel 311 SSD.

In this paper, we argue that such cost is largely due to implementation limitations of the existing system. We apply two simple techniques—ensuring a single I/O operation on the synchronous commit path, and adaptively allowing each file to have a custom journaling mode (in particular, whether to journal the file data in addition to the metadata). Compared to SQLite without file system journaling, our enhanced journaling improves the performance or incurs minor (<6%) slowdown on all but one of our 24 test cases (with 14% slowdown in the exceptional case). On average, our enhanced journaling implementation improves the SQLite performance by 7%.

1 Introduction

Ensuring data consistency and durability despite sudden system crashes (due to power/battery outages and software panics) is a critical aspect of computer system dependability. Following such failures, the application and system data should be recovered from durable storage to a consistent state without unexpected data losses. With data-driven applications spreading from servers and desktops to resource-constrained devices and near-client cloudlets [17], the overhead of such protection is also an important concern. Many applications utilize lightweight databases and key-value stores to manage their data. For instance, the SQLite database [19] and Kyoto Cabinet [3] protect the consistency of their data through techniques such as rollback-recovery journaling or write-ahead logging. Transactions are flushed to durable storage at commit time to prevent the loss of data.

These data management applications run on traditional file systems. While file system journaling protects the file system structure and metadata, it lacks knowledge of the application's semantics to protect application data. Such *journaling of journal* appears to violate the classic end-to-end argument that the low-level implementation of a function (transactional data protection in this case) is incomplete and it can hurt the overall system performance [16, 21].

We argue that the cost of additional file system journaling is largely due to implementation limitations of the current system. Simple enhancements can be made following two principles—1) the critical path of a synchronous I/O commit should involve a single sequential I/O operation to the storage device, 2) different application-level log files should be allowed to use customized file system journaling modes that best match their respective access patterns. This paper describes the design and implementation of our techniques on the Ext4 file system, as well as a performance evaluation with the SQLite database on a Google Nexus 7 tablet and a conventional machine with a NAND Flash-based SSD.

We note that the journaling of journal can be avoided if the operating system provides a richer interface that allows application data protection semantics to be exposed to the OS. The OS can then provide unified data protection for both application data and the file system metadata. Examples of such enhanced OS data management interface include I/O transactions [13, 18], software persistent memory [4], and failure-atomic msync() [11]. In contrast to these approaches, we explore performance enhancements of the existing file system journaling with only minor addition to its semantics and usage, which can be more easily deployed in practice.

The issue of journaling of journal has been raised before, notably in the context of Android's employment of the SQLite database on smartphones [6, 7]. While our work was partially motivated by these studies, our own observation in limited experimental setups suggest that the performance of many typical Android smartphone workloads is dominated by network (particularly wide-area network) delay rather than storage I/O. Although we make no argument on the importance of smartphone storage I/O performance in this paper, we do caution that the performance findings of our work only applies to I/O-intensive workloads in server/cloud and client systems.

*Park is currently affiliated with HP Labs.

For clarity, we will call an application-level journal a *log file* in the rest of this paper and the term *journaling* will refer to the file system journaling by default.

2 Fast Journaling of Journal

When an application-level transaction commits, its REDO or UNDO information is synchronously written to a log file. The atomicity of such a commit is often ensured through a checksum code written together with the committed transaction. A rollback-recovery log records transaction UNDO information. Updates on the database file(s) occur after the log write but before the transaction completes. Alternatively, write-ahead logging records transaction REDO information. Since the REDO record contains sufficient information to keep the database up-to-date, updates to the database file(s) do not need to happen before the transaction commits. In fact they can be delayed until they are overwritten by future transactions and therefore never written to durable storage.

Write operations are particularly expensive on NAND Flash-based storage devices [1, 12]. File system journaling protects the file system structure and metadata which can incur additional I/O costs in two ways. First, it may increase the number of I/O operations. For instance, the Ext4 ordered-mode journaling only journals the file system metadata but for consistency, it must write the file data before journaling the metadata. Second, it may increase the write size. For instance, the Ext4 data-mode journaling may write twice as much as the application does (once to the journal and a second time to the main file system structure). Furthermore, the journal data writes can be substantially larger than the original data writes due to the minimal journal record granularity (e.g., a 4 KB page).

However, we show that the cost of file system journaling can be mitigated through simple implementation enhancements. Our techniques do not change the existing journaling semantics of protecting the file system metadata and require very little application modification.

2.1 Single-I/O Data Journaling

Under journaling of journal, the data management application protects the consistency of its own data while file system journaling protects the file system integrity. However, the journaling of both metadata and file data (e.g., Ext4 data-mode journaling) allows a single sequential I/O operation on the critical path of a synchronous I/O commit (a `fsync()`, `fdatasync()`, or `msync()`) and therefore may enhance performance.

To accomplish single-I/O data journaling, the operating system should avoid direct file data or metadata writes on the synchronous I/O commit path. This principle is compatible with data journaling semantics since the journaled content contains sufficient information to recover the committed data in the event of a crash. To ensure it on Linux/Ext4, we examined its I/O traces and inspected its file sync code paths which led us to identify a violation in its current implementation. Specifically, the `ext4_sync_file()` function flushes the dirty file buffer (through `filemap_write_and_wait_range()`) before writing the journal. Such flushes skip buffers ready for data journaling but still write previously journaled dirty data buffers (resultant from earlier transactions). These file buffer writes are unnecessary for the journaling semantics and they slow down the synchronous commit. We make corrections to avoid these writes under data journaling.

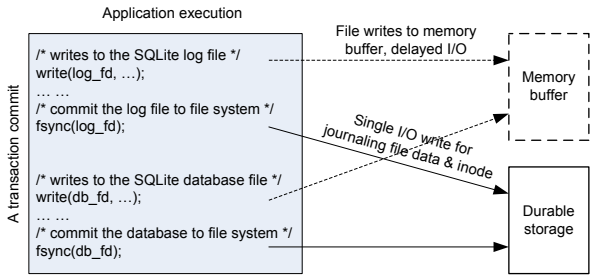


Figure 1: A simplified illustration of the journaling of journal during a transaction commit—file system journaling of the SQLite database that employs rollback-recovery logging.

The single I/O operation in the synchronous commit path only writes the file data and metadata to the file system journal. They would in theory still need to be asynchronously checkpointed to the main file system structure. In practice, the asynchronous checkpoints may never happen for data and log files when they are overwritten or deleted (for some UNDO logs) during continuous transaction processing. For instance, Figure 1 provides a simplified segment of system call trace during a transaction commit for the SQLite database that employs rollback-recovery logging. For the log file, the `fsync()` call would trigger a single I/O operation that commits the file data and metadata to the file system journal. Checkpoints to the SQLite log file itself are delayed and then typically canceled when the log file is deleted after the database transaction commits. Memory buffering and journaling for the database file writes behave similarly, in that the delayed checkpointing writes may be overwritten by a later transaction that writes to the same database location.

We also note that a traditional file system journaling transaction commit involves two write operations—first synchronously writing the transaction content and then writing a transaction commit record—to ensure atom-

icity. Alternatively, single-I/O atomic transaction commit can be accomplished by adding a checksum in the transaction block. In Ext4, this is supported by the `journal_async_commit` option. This option is not used widely in production systems due to a challenge of handling corrupted checksums in journal recovery [22]. There is also a concern in the case of ordered journaling that, with `journal_async_commit`'s removal of the I/O flush before writing the transaction commit record, the ordering between data writes and metadata journaling commits may not be enforced [23].

2.2 File-Adaptive Journaling

While single-I/O journaling of both file data and metadata minimizes the number of I/O operations during a synchronous commit, it may write a larger volume of data than the original write. This is primarily due to the typical journaling implementation's use of whole pages for journaling records (including Ext4/JBD2). Under such an implementation, a small write of a few bytes still requires a 4 KB journal record. This presents an efficiency dilemma between the single-I/O full data/metadata journaling and ordered metadata-only journaling—the former allows a single I/O operation on the commit path but may write significantly more data. Note that the single-I/O data journaling described in Section 2.1 is an important foundation for this dilemma because otherwise the ordered journaling would almost always outperform the standard data journaling.

In earlier work, Prabhakaran et al. [14] have proposed an adaptive journaling approach that selects the best journaling mode for each transaction according to its I/O access pattern. Specifically, a transaction that would perform sequential I/O under ordered journaling mode should utilize ordered journaling. Such a transaction typically writes a contiguous set of data blocks without making any file metadata change. On the other hand, a transaction that contains multiple I/O segments or performs both data and metadata writes should use full data/metadata journaling to gain the efficiency of a single, sequential I/O operation.

However, per-transaction adaptive journaling may not be safe in the case of overwrites. Specifically, consider two transactions T_1 and T_2 (in that order) that overlap in their file data writes. Assume that the adaptive journaling system chooses the full data/metadata journaling for T_1 while it selects the ordered journaling for T_2. A crash-induced journal replay will apply the metadata and file data for T_1 but only the metadata for T_2, and thus incorrectly leaving T_1's file data write as the final state. This effectively reorders writes in the two transactions that would not have happened under the standard (non-adaptive) full data journaling or ordered journaling. This problem may be resolved by a recent technique [2, Section 4.3.6] that journals data buffer overwrites for transactions which utilize ordered journaling.

Another concern for per-transaction adaptive journaling is its implementation challenge. Specifically, since the relevant transaction characteristics for adaptive journaling decision (e.g., sequential I/O or not) is not known at the beginning of a transaction, writes at the early stage of a transaction would have to be done in a way that permits either data or ordered journaling. This presents new challenges to a typical journaling implementation that performs a write differently depending on the journaling mode. For instance, a write under data journaling must prohibit dirty data flushes until the transaction is committed to the journal while a write under ordered journaling flushes dirty data earlier. Furthermore, a write under data journaling records data writes in the journal while a write under ordered journaling does not.

To avoid these safety and implementation complications, we propose a coarse-grained (per-file) adaptive journaling approach. As long as all journal transactions to a given file follow a single journaling mode (either full data/metadata journaling or ordered metadata-only journaling), a crash-induced journal replay should always result in the correct final state for the file data. Furthermore, we let the application to set the journaling mode for a file according to its access characteristics. This approach works well for practical journaling of journal circumstances. Specifically, write-ahead log files are generally written sequentially with little metadata change and therefore more suitable for the ordered file system journaling. In contrast, a rollback-recovery log file deletes or truncates the transaction record at the end of each transaction and therefore triggers substantial metadata changes. Consequently it is more suitable for full data/metadata file system journaling.

The proposed file-adaptive journaling can be easily implemented in practice. Changes to Linux/Ext4 primarily include new file and inode flags set through `ioctl()` that indicate the desired journaling mode for each file. The journaling code will preform appropriate actions for a file according to its inode flag. For safety, changing the journaling mode for a file requires the flushing of all pending journal transactions. In practice, such flushing incurs little overhead if the journaling modes are set once at database open time for files that persist through a long work session (e.g., the write-ahead log files). Our changes to the SQLite database involve adding about 20 lines of C code right after each log file open.

There is one additional correctness concern for per-file adaptive journaling in the case of data reuses between files. Specifically, if a data block is freed from a data-journaled file (after transaction T_1) and then reused by an ordered-journaled file (followed with transaction T_2),

then the block may be journaled in T_1 but not in T_2. Therefore our earlier example of journal replay-induced write re-ordering may again appear. In fact, similar block reuse problems already exist in the standard ordered journaling, when a metadata block is freed from a file and then reused by another file as a data block [24]. The standard solution to such a problem is to use journal revoke records that instruct the journaling replay mechanism to avoid replaying the concerned blocks. This has not yet been implemented in our current prototype system.

3 Experimental Evaluation

We implemented our journaling enhancements in the Linux 3.1.10 kernel and its Ext4 file system. We performed experiments on a Google Nexus 7 tablet with internal eMMC NAND storage. The Nexus 7 has a quad-core dynamic-frequency Tegra3/ARM processor, which was fixed to run at 1.0 GHz during our experiments. Our Nexus 7 is installed with the Ubuntu 12.10 Linux distribution. We also experimented with a conventional machine with an Intel 311 Flash-based SSD and two dual-core 2.0 GHz Intel processors. In each platform, we measured the transaction performance of the SQLite database (version 3.8.0.2).

We configured the system and application software to optimal-performance settings to establish a strong baseline. The Ext4 file system is mounted with `noatime,nodiratime,journal_async_commit` options. We also enable the `barrier=1` option to ensure the journal write ordering. We configured SQLite to use `fdatasync()` (instead of `fsync()`) for I/O commits. Note that `fdatasync()` will still write the dirty inode if the inode change affects the file system structural integrity (e.g., file size change).

3.1 Performance of File System Journaling

We compare the performance of different file system journaling approaches. Note that our file-adaptive journaling builds on our single I/O data journaling as described at the beginning of Section 2.2.

Our workloads run insert, update, and delete operations on a SQLite database table. Each record has an integer key and a 100-character value field. We test two transaction sizes—small, one-operation transactions and large, 1000-operation transactions. We also experiment with two application-level logging approaches in SQLite—write-ahead logging (WAL) and rollback-recovery logging. WAL is faster at transaction commits by issuing fewer `fdatasync()`'s (once vs. four times under the rollback-recovery logging), but it also has a number of practical disadvantages including poor support for reads and requiring periodic checkpoints [20].

Our test runs in rounds. Each round starts with an empty database table, inserts 10,000 records (10,000 one-operation transactions or ten 1000-operation transactions), updates the 10,000 records (in a random order that is likely different from the insert order), and deletes the 10,000 records (again in a random order). For each test case we run at least five rounds and report the average transaction response time.

Figures 2 and 3 illustrate performance results on our two platforms. Compared to no file system journaling, Ext4 ordered journaling has a worst-case slowdown of about 20%. Ext4 data journaling has a worst-case slowdown of about 73%. These indicate substantial costs of journaling of journal systems. Our single-I/O data journaling has enhanced the performance from the original Ext4 data journaling but it still suffers from a worst-case slowdown of 38%.

Our file-adaptive journaling has no more than 6% transaction slowdown compared to no file system journaling in all but one of our 24 test cases. It experiences 14% slowdown under the exceptional case of small (1 op/txn) update transactions with SQLite rollback-recovery logging on Nexus 7. A closer inspection finds that the overhead is due to page-granularity file system journal records and resultant larger write volume under full data/metadata journaling for the SQLite rollback-recovery log file. In one representative journal commit, nine small writes of a total 2,576 bytes turned into nine 4 KB pages (along with inode journaling and commit block, totaling 45,056 bytes) on the file system journal in Ext4/JBD2. We note that the page-granularity record is not a necessary design choice for file system journaling and previous work has shown that fractional-page journaling record is possible [5]. We expect that such an enhancement, if robustly incorporated into file system journaling, should further strengthen our argument that the journaling of journal is (almost) free.

We also find that our single-I/O data journaling and file-adaptive journaling achieve faster transaction responses than no file system journaling in some cases. This is most pronounced for large (1000 ops/txn) insert transactions with SQLite rollback-recovery logging on both machines. This is due to the substantial benefit of coalescing multiple piecemeal writes into a single I/O operation on the synchronous commit path. On average of all our test cases over different workload patterns and SQLite logging modes, our file-adaptive journaling in fact improves the performance of no file system journaling by 4% on Nexus 7, and by 9% on Intel 311.

Error bars in Figures 2 and 3 show the standard deviations of results from multi-round tests in each case. Most tests show stable results. The most deviating performance results do not always reoccur under a given test condition, which hints at the possible effect of occasional

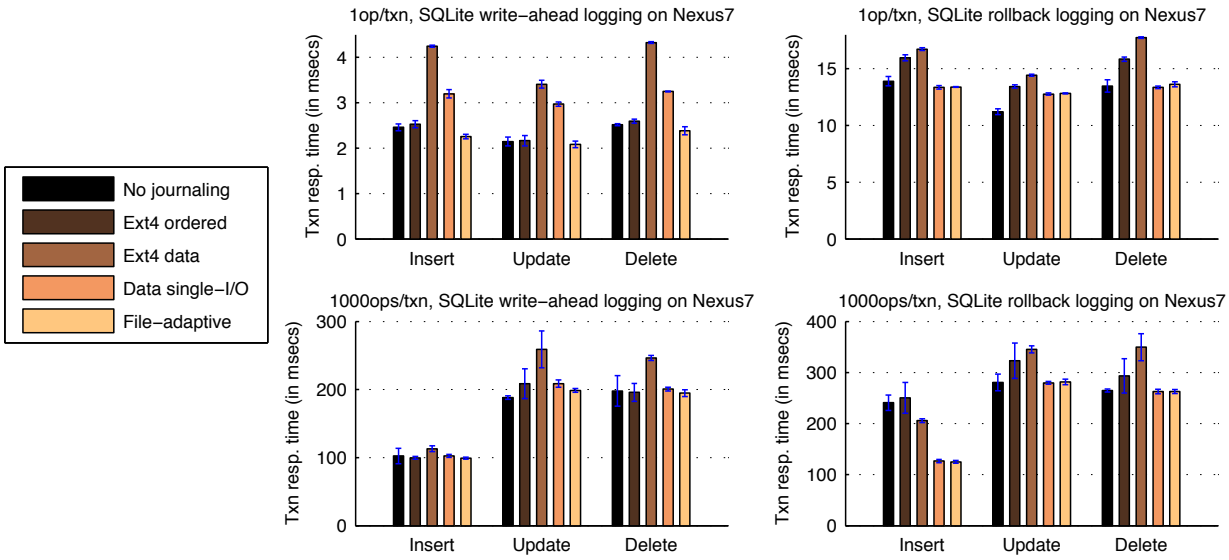


Figure 2: SQLite transaction response time under different file system journaling approaches on Nexus 7. The error bars show the standard deviations of results from multi-round tests in each case. The two columns show results for two SQLite logging modes (write-ahead logging and rollback-recovery logging). The two rows show results on two transaction sizes (1 operation per transaction and 1000 operations per transaction).

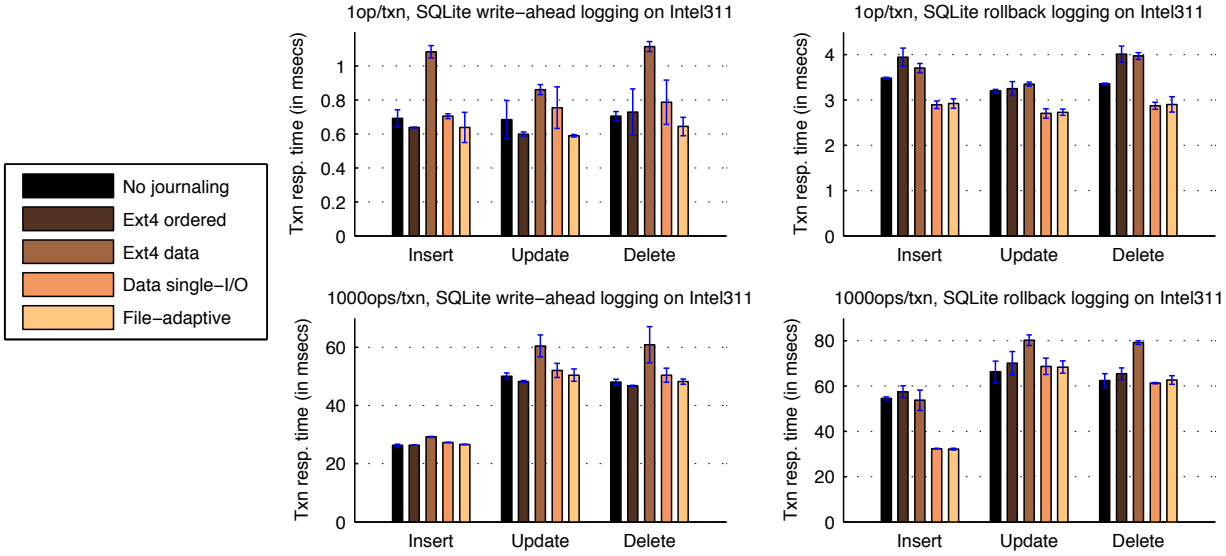


Figure 3: Performance results on a conventional machine with an Intel 311 SSD.

background Flash tasks such as garbage collection.

3.2 Cost of Transactional Data Protection

While we argue that properly adding file system journaling to applications that protect their own data incurs little additional cost, we are not suggesting that transactional data protection (ensuring data consistency and durability) over system failures is free. To reveal the cost of transactional data protection, we compare three system conditions on our experimental platforms—

- *Full protection*: SQLite write-ahead logging (application-level protection) combined with Ext4 file-adaptive journaling (file system protection).
- *Application-level protection*: SQLite write-ahead logging without any file system journaling.
- *No protection*: no SQLite logging or file system journaling; all writes are asynchronous and almost all operations are performed entirely in memory.

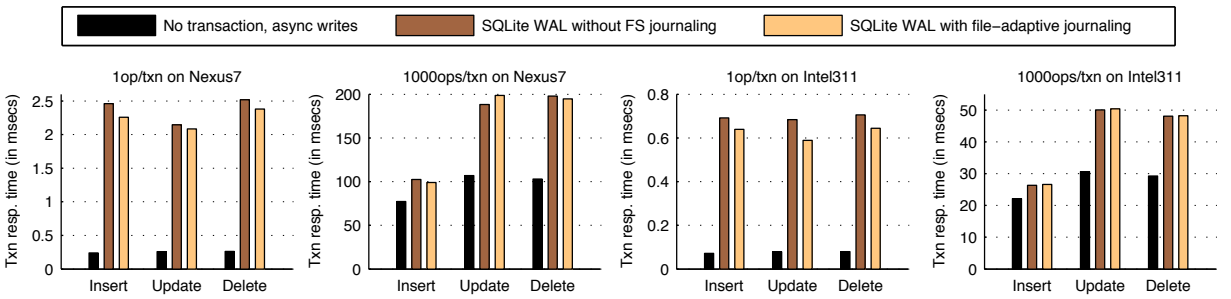


Figure 4: Cost of transactional data protection—performance comparison of application/system-level data protection mechanisms against the situation of no protection (effectively in-memory operations).

Results in Figure 4 show that in all cases, the full data protection with file system journaling adds almost no cost beyond application-level protection. Compared to no data protection with effectively in-memory operations, the transactional data protection and associated atomic, synchronous I/O adds substantial costs for small transactions (almost an order of magnitude slowdown). However, the costs are not excessive for large transactions (less than a factor of two). Larger transactions can be accomplished through careful application development and tuning, or system approaches that intelligently commit only when necessary [10].

4 Conclusion and Discussions

This paper argues that properly adding file system journaling to applications that protect their own data incurs minor additional cost. The large costs for journaling of journal on existing systems are primarily attributable to implementation limitations. We applied simple techniques that ensure a single I/O operation on the synchronous commit path and adaptively allow each file to have its custom file system journaling mode. We performed experiments for SQLite transactions on a Google Nexus 7 tablet and a conventional machine with an Intel 311 SSD. Compared to no file system journaling, our enhanced journaling implementation improves the performance or incurs minor (<6%) slowdown on all but one of our 24 test cases (with 14% slowdown in the exceptional case). On average, our enhanced journaling implementation improves the SQLite performance by 7%.

Our work contributes to the debate between using traditional vs. log-structured file systems [15] on NAND Flash-based systems. While a number of log-structured file systems (including NILFS [9] and F2FS [8]) have emerged recently and promise high performance on NAND storage, many systems have still chosen the Ext4 file system due to the concern of system maturity and robustness for production use. Our work suggests that such a choice does not necessarily carry significant performance costs.

Our proposed file-adaptive journaling is just one form of possible adaptive journaling approaches [14]. It has limited adaptation granularity (all transactions on a file must use a single journaling mode) but is effective for application-level log files that exhibit clear I/O patterns and journaling preferences. A broader use of adaptive journaling may require intelligent learning of the file access pattern and finer-grained control. Beyond adaptive journaling, journaling performance can be further enhanced by relaxing its ordering constraints [2].

Our cost evaluation has targeted the application response time. At the same time, we recognize that the employment of full data and metadata journaling tends to write more to a NAND Flash device and increase its wear. We explained in the paper that such increase of the write volume is primarily due to the page-granularity records in the journaling file system implementation. This can be potentially mitigated by a fractional-page journaling implementation [5].

Experimental work in this paper has focused on systems with NAND Flash-based storage. While much of our design rationale should also apply to mechanical disks, the resulted quantitative benefits might be different. In particular, since seek time dominates the performance of a mechanical disk, our single-I/O data journaling (Section 2.1) may produce larger performance gains than on Flash storage. On the other hand, the benefit of write size reduction under file-adaptive journaling (Section 2.2) might be much smaller on mechanical disks.

Acknowledgments This work was supported in part by the National Science Foundation grants CCF-0937571, CNS-1217372, CNS-1239423, and CCF-1255729, and by a Google Research Award. We thank Ted Ts'o for clarifying relevant issues in the Ext4 file system and its journaling support. We also thank the anonymous FAST reviewers and our shepherd Florentina Popovici for comments that helped improve this paper.

References

[1] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of Flash memory based solid state drives. In *ACM SIGMETRICS*, pages 181–192, Seattle, WA, June 2009.

[2] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *SOSP'13: 24th ACM Symp. on Operating Systems Principles*, pages 228–243, Farmington, PA, Nov. 2013.

[3] FAL Labs. Kyoto Cabinet: a straightforward implementation of DBM. `http://fallabs.com/kyotocabinet/`.

[4] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *USENIX Annual Technical Conf.*, Boston, MA, June 2012.

[5] A. Hatzieleftheriou and S. V. Anastasiadis. Okeanos: Wasteless journaling for fast and reliable multistream storage. In *USENIX Annual Technical Conf.*, Portland, OR, June 2011.

[6] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O stack optimization for smartphones. In *USENIX Annual Technical Conf.*, San Jose, CA, June 2013.

[7] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *FAST'12: 10th USENIX Conf. on File and Storage Technologies*, San Jose, CA, Feb. 2012.

[8] J. Kim. F2FS: Introduce Flash-friendly file system, Oct. 2012. `https://lwn.net/Articles/518718/`.

[9] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, July 2006.

[10] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. *ACM Trans. on Computer Systems*, 26(3), Sept. 2008.

[11] S. Park, T. Kelly, and K. Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *EuroSys'13 Conf.*, Prague, Czech Republic, Apr. 2013.

[12] M. Polte, J. Simsa, and G. Gibson. Comparing performance of solid state devices and mechanical disks. In *3rd Petascale Data Storage Workshop*, Austin, TX, Nov. 2008.

[13] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP'09: 22th ACM Symp. on Operating Systems Principles*, pages 161–176, Big Sky, MT, Oct. 2009.

[14] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conf.*, Anaheim, CA, Apr. 2005.

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

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

[17] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct. 2009.

[18] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI'06: 7th USENIX Symp. on Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006.

[19] SQLite. `http://www.sqlite.org/`.

[20] SQLite Write-Ahead Logging. `http://www.sqlite.org/wal.html`.

[21] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[22] T. Ts'o. What to do when the journal checksum is incorrect, May 2008. `http://lwn.net/Articles/284038/`.

[23] T. Ts'o. Personal communication, Jan. 2014.

[24] S. Tweedie. EXT3, journaling filesystem. In *Ottawa Linux Symposium*, July 2000. `http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`.

Checking the Integrity of Transactional Mechanisms

Daniel Fryer, Dai Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, Ashvin Goel
University of Toronto

Abstract

Data corruption is the most common consequence of file-system bugs, as shown by a recent study. When such corruption occurs, the file system's offline check and recovery tools need to be used, but they are error prone and cause significant downtime. Previous work has shown that a runtime checker for the Ext3 journaling file system can verify that metadata updates within a transaction are mutually consistent, helping detect corruption in metadata blocks at commit time. However, corruption can still be caused when a bug in the file system's transactional mechanism loses, misdirects, or corrupts writes. We show that a runtime checker needs to enforce the atomicity and durability properties of the file system on every write, in addition to checking transactions at commit time, to provide the strong guarantee that every block write will maintain file system consistency.

In this paper, we identify the invariants that need to be enforced on journaling and shadow paging file systems to preserve the integrity of committed transactions. We also describe the key properties that make it feasible to check these invariants for a file system. Based on this characterization, we have implemented runtime checkers for a modified version of the Ext3 file system and for the Btrfs file system. Our evaluation shows that both checkers detect data corruption effectively, and they can be used during normal operation with low overhead.

1 Introduction

File systems contain bugs that are hard to detect even under heavy testing, as shown by researchers [25, 35] and painful real-world experiences [24]. These bugs can result in data corruption, data loss, or persistent application crashes. Today, most techniques that enhance the reliability of storage systems focus on recovery from crash failures, and a variety of storage hardware failures [12]. However, none of these methods address corruption caused by file system or operating system bugs, or random memory corruption [36]. For example, a mirror RAID offers no protection against a buggy file system write, which would be reliably replicated on multiple disks.

A comprehensive study recently showed that 40% of file system bugs have severe consequences, because they lead to in-memory or on-disk data corruption [18]. When a file system bug corrupts file-system metadata, the damage can propagate and thus the entire file system must be checked for possible corruption. This consistency check is typically performed offline, causing significant downtime for large storage systems [14]. Furthermore, repair is an error-prone process [13, 2].

To avoid downtime and data loss, file system corruption must be detected before it propagates to disk. To do so, the file system's write operations must be checked at *runtime*. Unlike a typical offline file system checker, such as fsck, that checks the consistency of the metadata already on disk, the Recon system [10] checks that metadata updates preserve the consistency semantics of the file system at runtime. These semantics are expressed as a set of invariants that are derived from the properties checked by the offline checker. When kernel bugs or memory corruption lead to metadata updates that violate these consistency invariants, a corruption is detected and the updates are prevented from reaching the disk.

Recon takes advantage of transactional methods, such as journaling [34, 7, 32] and shadow paging [15, 5, 26], used by modern file systems for providing crash consistency. In particular, it checks that metadata updates within a transaction are mutually consistent at transaction commit time. This approach is still vulnerable to file system corruption when the transactional mechanism is used incorrectly or has bugs. For example, the Recon checker for the Ext3 journaling file system verifies writes to the journal blocks, but it assumes that 1) all metadata writes first go to the journal, and 2) these writes are then checkpointed correctly. Any bugs that violate these assumptions, e.g., a lost or failed checkpointing write, will cause undetected corruption. In Section 2.2, we show that these bugs manifest in many different ways, such as lost, misdirected, out-of-order and corrupting writes,

making it difficult to detect them. Unfortunately, these types of bugs occur regularly [10], are hard to diagnose [11, 27], and can have serious impact [33].

In this paper, we describe the design and implementation of a runtime checking system that enforces correct usage and implementation of the crash consistency method used by the file system. Our system enforces the atomicity and durability properties of the file system at each block write, in addition to checking consistency at commit time, providing the strong guarantee that every block write will maintain file system consistency.

We express the atomicity and durability properties as invariants, called *location invariants* because they govern which blocks are written to given locations. We describe the location invariants that need to be enforced to preserve the integrity of committed transactions for journaling and shadow paging file systems, and the file system properties that make it feasible to check these invariants efficiently at the block layer.

We have implemented runtime checkers for the Btrfs file system and a slightly modified version of the Linux Ext3 file system by augmenting the Recon system. Our evaluation shows that the runtime checkers for both the file systems detect file-system corruption effectively, preventing any file system metadata inconsistency. We show that the Ext3 checker has low performance overhead, while the Btrfs checker overhead is higher due to increased metadata load. Checking location invariants in both checkers has negligible overheads.

2 Motivation

Our aim is to design a runtime checking system that can reliably detect file system and other operating system software bugs, and memory corruption errors, before they cause on-disk data corruption. Unlike an offline file system checker, a runtime checker does not detect file system corruption caused by I/O hardware failures, such as device controller failures or latent sector errors on disks. Instead, the runtime checker depends on hardware redundancy mechanisms, such as checksums and replication [25], implemented either in the storage system or in the file system [5, 9], to detect and recover from such failures when data is read from disk.

A runtime checking system can be deployed in either a development or a production setting. During development, a runtime checker can serve as a testing tool, catching subtle errors before the file system image becomes inconsistent, making it easier to determine the root cause of a bug. In production, the checker could trigger measures to preserve existing data, recover from the failure [31], or alert administrators to the problem. Our runtime checking system builds on the Recon system [10], and so this section starts by providing an overview of Recon. Then we motivate this work by discussing the types of bugs that Recon will fail to detect, leading to undetected data corruption.

2.1 The Recon System

The Recon system takes advantage of transactional methods, such as journaling and shadow paging, used by modern file systems for providing crash consistency. These transactional methods group writes to disk blocks from one or more operations (such as the creation of a directory and a file write) into transactions. When transactions are committed, the file system believes itself to be consistent. At this point, Recon checks that the contents of the blocks involved in the transaction are mutually consistent, thus detecting the effects of software bugs (or memory errors) that corrupt blocks within the transaction.

The consistency checks in Recon are derived from the consistency properties of the file system. These properties constrain the set of valid file system states that can be generated by an arbitrary sequence of file system operations. Typically, these properties are checked by the offline file system checker. For example, a consistency property in the Btrfs file system is that extents must not overlap. Checking this property requires a full scan of the extent tree, making it infeasible to perform at runtime. Instead, each consistency property is transformed into a local consistency invariant, which is an assertion that must hold for the transaction blocks to preserve consistency. In the Btrfs example, the consistency invariant is that when a new extent item is added to a tree, then the extent must not overlap with the previous or next extents in the extent tree. A runtime checker can enforce this consistency invariant by examining all updated extents and their adjacent extents.

The Recon system interposes at the block layer, and can be implemented in the host operating system, a hypervisor or a storage controller. The benefit of this approach is that the checker only depends on the the format and the consistency properties of the file system, rather than depending on the implementation of the file system, which may be buggy and cannot be trusted. File system formats and their consistency properties tend to be stable over time, even when the implementation changes significantly over time, or there are multiple different implementations of a particular file system.

The consistency invariants are expressed in terms of logical file-system data structures, such as the extent information in the Btrfs example. Since Recon interposes at the block layer, it uses an introspection approach, similar to semantically smart disks [30], to infer the types of blocks as they are accessed, and then interprets the block contents to derive the logical file-system data structures.

2.2 Problematic Bugs

Recon ensures that the blocks in a transaction are consistent, but it depends on the transaction mechanism being both implemented and used correctly. Below, we describe four classes of bugs that break these assumptions, and provide some examples of recent bugs in the Ext3, Ext4 and the Btrfs file system code deployed in "stable" Linux kernel releases.

Overwrite bugs: A write occurs to a location when it shouldn't have happened at all, either due to improper writing or flushing of buffers, or some other failure that causes a misdirected write. For example, Ext4 stores file system quota information as data in special quota files. The contents of these files are metadata, similar to directories, but they were overwritten in place without first writing to the journal, when the file system was used with certain mount options [16]. Recon's consistency invariants would not detect this problem because the journal would appear to be consistent. Similarly, a high profile bug was recently introduced in the Ext4 file system, in which the inode bitmap was modified without updating the journal, which could lead to occasional corruption [27]. Interestingly, after the corruption issue was reported, the developers at first mistakenly thought that the root cause was an incorrect update to the journal superblock [33]. This suggests that understanding, using and implementing the transactional mechanism is challenging and bug prone. In this case, if the file system is allowed to continue running, the transaction that was missing the inode bitmap update in the journal would commit, and the checkpoint of that transaction would bring everything back to a consistent state, with no one the wiser. Consistency problems only occur when an ill-timed crash forces recovery from the incomplete journal entries. When Recon is used in production, things actually become worse. Recon would detect that the journal contents are inconsistent, because the inode bitmap updates are missing (e.g., unallocated inodes would appear to change), and then discard the transaction and stop the file system. The inode bitmap, overwritten in place, would cause the file system to become inconsistent.

Lost write bugs: A write that should happen doesn't occur. For example, in a journaling file system, a lost checkpointing or recovery write will cause file system inconsistency even though the journal is consistent [12].

Write ordering bugs: The file system needs to enforce ordering of writes to disk at certain times. While the block layer may observe writes in the correct order, unless the correct disk barrier commands are sent, the disk or its controller may reorder writes, causing inconsistency of the on-disk state on a power loss. For example, Linux JBD2 journaling code maintains a pointer to the journal tail in a journal superblock. When the tail was updated, the journal superblock was not being flushed to disk before new transactions could reuse the newly freed journal space. On a power loss, the recovery code could replay old transactions containing blocks potentially overwritten in the journal by new transactions [17], including blocks from uncommitted transactions. Similarly, the Btrfs file system in multi-device setups (e.g., mirroring) would send barriers in the wrong order and not wait for all the barriers before writing the commit block [21]. These write ordering bugs would not be detected by Recon but they can cause serious file system inconsistencies.

Corrupting Write bugs: A write occurs to the correct location but its contents are corrupt. For example, the Ext3 journaling code modifies (escapes) its data blocks when they start with a magic code that identifies journal metadata blocks, to distinguish between the two types of blocks, similar to bit stuffing [1]. When Ext3 was used in data journaling mode, the recovery code had a bug that would unescape the wrong buffers, causing corruption of both the block that remains escaped, and the block that is wrongly unescaped [11]. This bug would not be caught by Recon's consistency invariants because the journal itself is not corrupt. However, blocks from committed transactions would be corrupt on disk following recovery.

3 Location Invariants

File systems that use transactional mechanisms for crash consistency provide atomicity and durability properties. Atomicity properties ensure that the file system will be able to roll back to a consistent state on a crash. Durability properties ensure that if a new version of a block is committed, it does not get rolled back or overwritten, except atomically as part of a subsequent transaction.

The problematic bugs described in Section 2.2 can cause corruption because they lead to violations of these properties. For example, a metadata overwrite that is not first committed to the journal violates atomicity, since we cannot roll back to the previous correct version of the block. Durability can be violated by either an omitted checkpoint write, or a write that corrupts a committed transaction in the journal, since updates that were successfully committed to the journal never reach the file system. Finally, in both journaling or shadow paging systems, a misdirected write that overwrites an allocated metadata block (e.g., a data block write that overwrites a metadata block) violates both atomicity and durability.

In this section, we first describe what is needed to detect violations of these properties, and then present the location invariants for journaling and shadow paging transactional mechanisms.

3.1 Enforcing Atomicity and Durability

The Recon runtime checker depends on the correctness of the file system's transactional mechanism to properly enforce the atomicity and durability of the metadata updates that it is checking. Unfortunately, in spite of Recon's distrust of buggy file systems, it assumes that the transactions themselves are implemented and used correctly. This assumption can be violated by several classes of bugs, as shown in Section 2.2. To detect these bugs, a runtime checker needs to enforce atomicity and durability invariants, in addition to consistency invariants. Consistency invariants apply to the contents of updated blocks; they need to be checked at transaction commit points because the file system does not guarantee that the updates are consistent until the commit. In contrast, the atomicity and durability invariants need to be checked on each block write, because they govern whether the write is permitted to the given location. Hence, we call them *location* invariants collectively. Rather than being derived from the offline checking tool, the location invariants are derived from the semantics of the transactional mechanism itself. In particular, they concern *overwrites* to the blocks, and the *ordering* of block write operations.

It is possible to enforce both atomicity and durability invariants on each write because they only depend on the correctness of committed metadata, which has already been checked using consistency invariants. Transactional techniques like journaling or shadow paging must first write metadata to unallocated blocks – for journaling, these are free blocks in the journal area, which must later be checkpointed back to the file system, while for shadow paging these may be any free blocks, which become part of the file system atomically at the commit point. To check that these properties are maintained, location invariants depend on information about block allocation and block type (data vs. metadata). The block allocation information must be based on committed metadata, since uncommitted changes to the allocation state may be rolled back following a crash. In particular, we must not permit a write to a block that has been freed in an uncommitted transaction, since we would not be able to recover the previous version of the block if the deallocation operation were rolled back.

As can be seen, correct checking of consistency and location invariants is interdependent. We begin from the assumption that the file system state on disk is consistent. Initially, this is the result of correct file system initialization, as is done by mkfs. Thereafter, each block write prior to a transaction commit is checked by the location invariants using the old, consistent, committed allocation and block type information. These checks ensure that the committed state is not corrupted. At the transaction commit point, the contents of the transaction are checked by the consistency invariants to ensure that the new file system state will be consistent. The location invariants then govern the write of the commit block itself, and the subsequent checkpoint writes to the file system, as well as the writes of blocks in the next transaction. By enforcing both consistency and location invariants, the runtime checker can provide the strong guarantee that the file system meets its consistency specification on every block write.[1]

As we will see in the next subsection, there are significant differences between the specific location invariants that apply to journaling and shadow paging mechanisms. However, both require the ability to infer block allocation information and the ability to distinguish between metadata and data blocks at the block layer.

3.2 Journaling Invariants

Journaling file systems use write-ahead logging to support failure atomicity. First, they write a consistent set of blocks and their final location information to a designated journal area. When all these blocks are durable in the journal, an atomic journal write signals a commit. After commit, the contents of the journal are flushed to their final locations. This flush to the final file system locations is called checkpointing in the Linux ext3/jbd terminology.

The journal area must be known to the runtime checker so that, on each write, it can distinguish between journal and non-journal writes. This distinction is necessary so that the correctness of both the journal writes and the checkpointing writes can be verified. Checkpointing of committed transactions occurs concurrently with new journal writes, but checkpointing writes must be directed to the non-journal area. Note that although we expect the journal to be a circular buffer, with writes occurring sequentially, at the block layer there is no guarantee of any particular ordering within a transaction.

The following four location invariants ensure that the journaling and checkpointing operations of the file system are correct:

1. Log invariant: A write to the journal area must be to a free block in the journal. A free journal block becomes allocated when it is written and free again when it has been checkpointed (see Checkpoint invariant below). This invariant checks that the allocated journal blocks are not overwritten.

2. Commit invariant: A write of a commit block, which marks a transaction as committed, is allowed to the journal area only after (1) all the blocks in

[1] While the checker implementation may have bugs that generate false alarms, it is unlikely that the checker will fail to detect file system corruption, unless its bugs are correlated with file system bugs [10].

the transaction are allocated in the journal, and (2) a barrier is issued to flush these transaction blocks to the disk. The transaction is considered to be committed (and hence, to be durable) only after the commit block is flushed to disk. When journal checksums are included in the commit block, as in IRON file systems [25], the write of the commit block can be concurrent with the writes of the transaction blocks, but a barrier is still needed to ensure that all these blocks are on disk before the transaction is deemed to be committed.

3. Flush invariant: A write to an allocated, non-journal location is permitted only when (1) the committed part of the journal contains a block that is destined for the same final location, and (2) the contents of this block in the journal matches the contents of the block being written. In other words, overwrites of allocated non-journal blocks are disallowed if the new content was not first committed to the journal. If the block exists only in the uncommitted portion of the journal, or the block does not appear in the journal at all, both atomicity and durability violations can occur. Atomicity is violated by writing new content into the file system ahead of the commit of the transaction that should contain it. Durability is violated by the loss of previously committed content that has been overwritten.

4. Checkpoint invariant: A write of a checkpoint record (e.g., in the journal superblock), which indicates that a set of blocks in the journal area are now free, is permitted only after all the journal blocks for the associated transaction have been either (1) flushed (see Flush invariant), or (2) superseded by a newer version of the corresponding block in a later committed transaction. If a newer version of a block exists in a later committed transaction in the journal, then this version does not need to be flushed before being freed. The affected journal blocks can only be considered free after the checkpoint record has been flushed to disk.

Metadata-only Journaling Since writing to the journal potentially doubles the total write traffic to disk, many file systems allow journaling only metadata blocks to reduce write traffic. The main complication with metadata-only journaling is that data writes are non-atomic, and while these writes must be allowed at any time, they must not overwrite metadata blocks. To accommodate non-journaled data writes, we refine the journaling flush invariant with an exception:

1. Data-flush exception: Any non-journal write that violates the flush invariant must be to a non-metadata (data or free) block location. The type of a block (metadata or not) is determined by the committed file-system state. The consequence of this exception is that data writes can overwrite data blocks unimpeded. Unfortunately, there is no way to tell if data writes are misdirected among each other.

The challenge with allowing this exception is that it must be possible to distinguish metadata blocks from non-metadata blocks on each write, but a file system may not provide this information easily. For example, the Ext3 file system uses allocation bitmaps that allow distinguishing between allocated blocks (which may be data or metadata) and free blocks. However, the file system does not provide an easy way to distinguish between dynamically allocated metadata (e.g., for directories and indirect blocks) and data blocks, other than by traversing the entire file system. We discuss this issue further in Section 4.3.

3.3 Shadow Paging Invariants

Compared to journaling, it is simpler to enforce location invariants for shadow paging systems because blocks are updated once per transaction and all these updates occur before commit. In a file system that uses shadow paging for all blocks, there are two atomicity invariants:

1. Flush invariant: All writes, other than to special non-shadow paged blocks, such as the super block, must be to unallocated blocks. This invariant follows from the basic copy-on-write properties of shadow paging systems. To enforce this invariant, the file system must provide an efficient method for determining the allocation status of a block. For example, the Btrfs file system maintains an extent allocation tree.

2. Commit invariant: The write of the commit block (usually a tree root) is flushed to disk only after both (1) all blocks referenced by the new tree have been updated, and (2) a barrier is issued to flush these blocks to disk. That is, there must be no dangling pointers to potentially uninitialized blocks, before the commit block is flushed.

Durability (e.g., a lost or corrupting update) is checked in modern shadow paging file systems using methods such as block checksums (ZFS) or generation numbers (Btrfs). This information is embedded in metadata blocks, and hence our Btrfs runtime checker uses consistency invariants to check the consistency of block headers and generation numbers for ensuring durability.

Metadata-only Shadow Paging Shadow paging can lead to fragmentation because the updated blocks are placed in new, possibly distant, physical locations. Fragmentation can be reduced with metadata-only shadow paging, with data writes being performed in place. To accommodate non-atomic data writes, we refine the flush invariant with an exception:

1. Data-flush exception: Any write that violates the flush invariant must be to a non-metadata (data or free) block location.

This exception requires being able to distinguish metadata and non-metadata blocks. Btrfs tracks whether an extent has metadata or data in its allocation tree, making it easy to enforce this invariant. Also, the default behavior of Btrfs is to separate metadata and data regions, making this identification even easier and more efficient.

4 Implementation

As explained in Section 3.1, location and consistency invariants are interdependent, and they need to be checked together. Hence, we have implemented location invariant checking for the Linux Ext3 (journal invariants) and Btrfs (shadow paging invariants) file systems by augmenting the Recon consistency checking system. Recon uses the block-layer Linux device mapper framework to interpose on block I/O, allowing location invariants to be checked on all writes. The block-layer approach ensures the independence of the checker and the file-system implementations. Next, we describe the requirements for implementing a runtime checker, and then discuss how these requirements are met in our implementation.

4.1 Runtime Checker Requirements

File system design impacts the capabilities and performance of a runtime checking system. In this section, we present the four types of information needed by a checker. The challenge is to obtain this information correctly and efficiently at the block layer. The more file system state that must be examined to do so, the higher the overhead of the checker.

Consistency Points: Runtime checking at the block layer requires being able to get a consistent picture of the file system state from outside the file system. Consistency points provide both a point in time to check consistency invariants and a consistent view of the file system when checking location invariants on each write.

Allocation Information: A checker needs to distinguish between allocated and unallocated blocks, particularly on the write path, to protect against accidental overwrites. Overwriting an unallocated block is harmless, but location invariants constrain when allocated metadata blocks can be overwritten.

Separate Metadata: The checker also needs to distinguish between metadata and data blocks on both the read and the write paths. Metadata blocks are cached to improve checker performance, since recently accessed metadata is likely to be relevant to invariant checking, while data blocks are ignored because they are not interpreted. Additionally, the location invariants may permit or forbid a write depending on whether the destination is a data or metadata block.

Block Identity: Finally, interpreting a metadata block requires knowing the *identity* of the block. The block identity determines the logical contents of the block in the file system. For example, suppose that the checker knows that some block is an inode block, and it identifies the block as the fourth inode block in the file system. If it knows that inode blocks contain 32 inodes, then it can determine that this block contains inodes with numbers 97-128. A runtime checker can then correlate these inodes with directory entries that reference them, with inode bitmaps that allocate them, and with the indirect blocks to which they point. Without knowing their specific identities, it would not be possible to make the associations between the data structures that are needed for enforcement of invariants.

4.2 Block-Layer Metadata Interpretation

In this section, we discuss two complementary approaches for determining block identity. The following sections describe how we apply them to interpret metadata in the Ext3 and Btrfs checkers.

Forward Pointers: File systems are tree structures or directed acyclic graph structures, with parent blocks containing some form of a pointer to child blocks. Thus, the easiest way to identify a block is if we are already traversing the parent block. For example, if the checker (or the file system) is looking up some specific metadata, starting from the root of the tree, it can traverse intermediate blocks to locate the desired block.

Back References: A back reference for a block is metadata that maps the block's physical location to blocks that reference the block [20], providing an efficient method for locating parent blocks. Back references are used for various tasks such as defragmentation and bad block replacement, in which the parent block containing the reference must be efficiently located and updated. The parent block has information to help type and identify the child block, and hence back references greatly simplify metadata interpretation. However, looking up a back reference may incur additional I/O operations.

4.3 Ext3 Implementation

Ext3 uses static block allocation bitmaps, making it easy for the checker to determine the allocation status of blocks. However, Ext3 does not provide any efficient method for distinguishing metadata blocks from other blocks, either on block writes or on block reads that violate pointer-before-block traversal. One option is for the checker to also create in-memory back references for all data blocks when the parent metadata blocks are traversed. This approach would greatly inflate the memory overhead of the checker. Instead, we have retrofitted the Ext3 file system with a metadata allocation bitmap which records whether a given block is metadata. The new metadata bitmap is stored alongside the block allocation bitmap. Using the metadata bitmap,[2] the checker can ensure that data blocks are never cached on either a read or a write, and the data flush exception, described in Section 3.2, can be implemented easily.

4.3.1 Interpreting Metadata

The Ext3 file system does not provide back references. Instead, we use the file system's forward pointer traversal to create in-memory back references dynamically. The file system needs to read the parent of a block at least once before it accesses the child block, which we call pointer-before-block traversal. When the parent block is read the first time, we create a back reference for each of the child blocks to which it points. For example, when an inode block is read by Ext3, we copy the block into the read cache, parse the inodes in the block, and then create back references for all child metadata blocks (e.g., indirect blocks) directly pointed to by the inodes. The back reference contains the block type, and for an indirect block, it contains the inode number and an offset that locates the indirect block. When the indirect block is read, its back reference will exist, and hence the block can be typed and identified. These back references are bootstrapped using the superblock, which exists at a known location.

The main drawback of in-memory back references is that they cannot be evicted because the file system may cache information from the parent block indefinitely, allowing it to access the child block directly at any time in the future. However, the in-memory references could be persisted by leveraging the backpointer-based consistency techniques developed in NoFS [6] and ffsck [19].

4.3.2 Location Invariants

The Ext3 location invariants require tracking the state of the journal. The checker maintains three data structures: a list of transactions currently present in the journal, an array containing information about the status of each block in the journal, including block checksums, and a hash table mapping from physical block numbers to versions of that block in the journal. A block in the journal can be in one of four states: logged, committed, flushed, and free. These four states correspond to the four journaling invariants described in Section 3.2. Note that a block stays allocated (as explained in the Log invariant) during the logged, committed, and flushed states.

Based on writes to different types of journal blocks (i.e., the descriptor blocks, metadata blocks, commit blocks, and the journal superblock) and non-journal blocks, the checker updates its data structures and the block states, and enforces the journaling invariants described in Section 3.2.

During a commit, if a new metadata pointer is found without a corresponding new metadata block in the journal, we detect a violation of the commit invariant (that all blocks should have been written before commit).

One complication with metadata-only journaling is that Ext3 uses *revoke* records to indicate that a metadata block has been freed, and could be reused as a data block that is updated non-atomically. As a result, any versions of this block in previous transactions should no longer be checkpointed or else the data block could be overwritten. The checker handles such revoked blocks by marking their status as checkpointed, so that the Checkpoint Invariant does not fail if the containing transaction is freed without seeing a write to that block.

4.4 Btrfs Implementation

Btrfs provides various features such as extent-based allocation (which allows a single allocation record to cover multiple blocks), back references (which help tasks like online defragmentation) and writable snapshots (which are isolated from the original version using copy-on-write semantics). Btrfs uses shadow paging for ensuring crash consistency, similar to the WAFL file system [15].

Btrfs uses multiple B-trees to store its metadata. A root B-tree contains pointers to the roots of other B-trees, including the main file system tree, snapshot trees, and an extent tree that records allocation information. Each B-tree consists of internal nodes and leaves. Internal nodes contain an array of key/block-pointer pairs, with the key representing the smallest key stored in the pointed-to node or leaf, and the block pointer helping locate the child node or leaf on disk. All Btrfs metadata blocks begin with a header that has a block checksum, a generation number, and the id of the tree containing the block.

We found that Btrfs can issue writes from concurrent transactions. For example, blocks from the next transac-

[2] Note that the consistency checker implements additional invariants for checking metadata bitmap consistency.

tion can be written to disk before the current transaction commit, but as expected, the next transaction blocks are unreachable from the current transaction. As a result, the Btrfs checker assumes that unreachable blocks belong to a future transaction and delays processing them.

4.4.1 Interpreting Metadata

Btrfs uses shadow paging, so that when a leaf node is updated, all its ancestor nodes are also updated. Because of this property, the checker can use forward pointer traversal on commit, starting from the superblock.

Btrfs uses an extent B-tree to store allocation information, which the checker also uses to determine the allocation status of blocks. Similarly, separating data and metadata blocks on both the read and write paths is relatively easy because Btrfs allocates separate large contiguous regions for data and metadata. However, if Btrfs is operating in a "mixed" region mode (not a common configuration), data extents can be distinguished from metadata extents by traversing the extent allocation tree and examining the per-extent flags.

Btrfs uses typed and self-identifying metadata blocks. Each metadata block has a header that stores the type (node or leaf) and level of the block in the tree, and the first key in the block is its identity, helping locate the block in the tree. Btrfs also supports back references to multiple snapshots, storing them with the allocation information in the extent tree.

Both back references and self-identifying metadata blocks can be used independently to type and identify blocks. We initially decided to implement a Btrfs runtime checker because we thought that both of these properties would be useful for the runtime checker. However, neither are necessary due to the forward pointer traversal enabled by shadow paging.

4.4.2 Location Invariants

The checker ensures atomicity and durability by checking that allocated blocks are never overwritten, which requires looking up the extent allocation tree on each write. For metadata-only shadow paging, a metadata flag in the extent record is checked to implement the data-flush exception. While checking a transaction for consistency, an invariant is tripped if a pointer to an unwritten block is encountered within the updated tree.

5 Evaluation

We evaluate our runtime checker in terms of its ability to detect violations of the location invariants, listed in Section 3, and the performance impact of checking location invariants in addition to consistency invariants for the Ext3 and Btrfs file systems. We have implemented the runtime checkers within the Linux kernel using the Recon framework, based on the approach described in the previous sections. Our Btrfs implementation is based on the Linux 2.6.35 kernel, which does not support passing disk barrier and flush requests through the device mapper, and so we cannot check for them. Recon for Ext3 is implemented and tested on Linux 3.8.11.

5.1 Correctness

We evaluate the ability of our runtime checker to detect the types of bugs described in Section 2.2. Specifically, we inject errors into write operations issued to the block layer that result in lost, misdirected, or corrupted writes. We refer to these injected errors as corruptions. If writes are correctly ordered, and no writes are lost, misdirected, or corrupt, then the transaction mechanism is working correctly. By deliberately altering writes to violate these properties, we can evaluate whether the location invariants can successfully protect the file system.

Our corruptor sits between the file system and the checking system, and has the opportunity to act before each write is visible to the checker. The actions the corruptor can take are: 1) discard a write (lost), 2) alter the destination of the write (misdirect), or 3) alter a range of bytes within the block being written (content). Because the location invariants distinguish between several different types of blocks, we perform corruption in a type-specific manner to increase our coverage of possible scenarios and to help explain any uncaught corruption. The type of a block is determined by its destination, and in the case of certain journal blocks, by the journal header stored at the beginning of the block.

5.1.1 Corrupting Ext3

The corruptor can target one of four types of journal blocks (journal metadata such as a descriptor block, revoke, and commit blocks, and journaled file system metadata block), or the two types of non-journal blocks (file system metadata and data). To misdirect writes, it must distinguish between free and allocated journal space, and data and metadata locations outside the journal. When the corruptor targets a non-journal metadata block write, it is emulating a bug that corrupts the checkpoint write of that metadata block. When the corruptor targets a data block write, it always misdirects the write to a non-journal metadata block. Lost write and content corruption types are not applied to data block writes.

Some corruptions may not violate location invariants immediately. Instead, they may lead to a future operation causing metadata corruption. For example, a lost write to the journal cannot be detected when it is dropped, and

Target Block Type	Corruption Type		
Journal Blocks	Lost	Misdirect	Content
Descriptor	10	10	8
Commit	10	10	4
Revoke	10	10	4
Metadata	10	10	2
Non-Journal Blocks			
Metadata	3	10	10
Data	N/A	10	N/A

Table 1: Corruptions detected by location invariants.

the resulting transaction may still be consistent, but the problem should be detected when the checkpoint write targets a metadata location that has not been committed to the journal. There are four distinct points in time when a corruption may be detected: during the corrupted write, at the next commit point, during the checkpoint of a corrupted transaction, and during transaction free. Any corruption that occurred in the past must be caught before a write harms the atomicity, durability, or consistency of metadata on disk.

There are a total of 16 combinations of target block types and corruption types, as shown in Table 1. We perform 10 corruptions per combination. Out of 160 corruptions, 131 were detected by the location invariants and 7 were detected by the consistency invariants (all were content corruptions of metadata blocks in the journal). We analyzed the remaining 22 corruptions that did not trigger any invariant violations. There are two situations in which we miss corruption events, but the "corruptions" do not affect file system integrity. In the case of random content corruptions to journal metadata, much of the space in the block is unused and corruptions to the unused area have no effect on the block semantics. Together, these cases account for 14 of the missed corruptions. Similarly, when unused space in a journaled metadata block is corrupted, which occurred in one case, no invariants are violated. We verified that the corrupted space was unused by logging the range of bytes corrupted and examining the target blocks. The final 7 missed corruptions were all lost checkpoint writes. In each case, we verified that these writes were safe to omit because there was already a newer version of the block committed to the journal. In all the 22 cases where we didn't catch the corruption, the e2fsck offline checker also reported that the file system was consistent.

5.1.2 Corrupting Btrfs

Testing the Btrfs location invariants is less involved, since the invariants are simpler, as described in Section 3.3. A buggy write in Btrfs can be redirected to overwrite an existing data or metadata block, lost, or redirected to the wrong free block. We simulated a metadata block being misdirected by the file system by changing the block's header to match the new, incorrect location, and updating the block checksum accordingly. Our checker always detected misdirections that cause overwrites of allocated data or metadata. Lost writes or writes that are misdirected to an incorrect free block are always detected by Recon during transaction processing, when a new pointer is found to a block that is missing from its write cache [10].

5.2 Performance

For benchmarking, we select three workload profiles with different behaviors from the Filebench workload generator. The varmail profile performs many small, synchronous writes. The webserver profile reads many small files concurrently (100 threads) in a large directory hierarchy (250,000 files), while appending to a log. The ms_nfs profile simulates a network file server, operating on a file system with a file size distribution from a study of Windows desktops [23].

All benchmarks were run on a dual-core 3.0 Ghz Xeon server with 4GB of RAM. The target disk for the benchmarks was a 250GB 7200rpm SATA drive. We allocated 256 MB of memory to the Recon caches [10]. The performance results account for Recon's memory usage because Linux implements a shared page cache, and so with Recon, this memory is not available to the file system cache.

Figures 1 and 2 show the benchmark throughput, and the time to initialize the benchmark's file system tree (setup time), averaged over 5 runs, for the Ext3 and the Btrfs file systems. Each graph shows the performance of the native file system, the file system with consistency checking enabled, and the file system with consistency and location checking enabled. These figures show that the overhead of checking location invariants is minimal compared to the existing overhead of checking consistency invariants in Recon. Even though the location invariants require a check on every write, this check is usually quick and takes advantage of the cached metadata.

Figure 1 shows that runtime consistency checking has relatively low overhead for the Ext3 file system. Figure 2 shows that the checking overhead is slightly higher for the Btrfs file system. The total amount of metadata is higher in Btrfs, due to its increased internal redundancy and larger data structures, putting more pressure on the Recon caches. The rapid reallocation of metadata blocks in a copy-on-write system makes it important to promptly evict blocks in the Recon caches that are no longer referenced.

The varmail and ms_nfs profiles show minimal overheads. Surprisingly, the most significant impact on per-

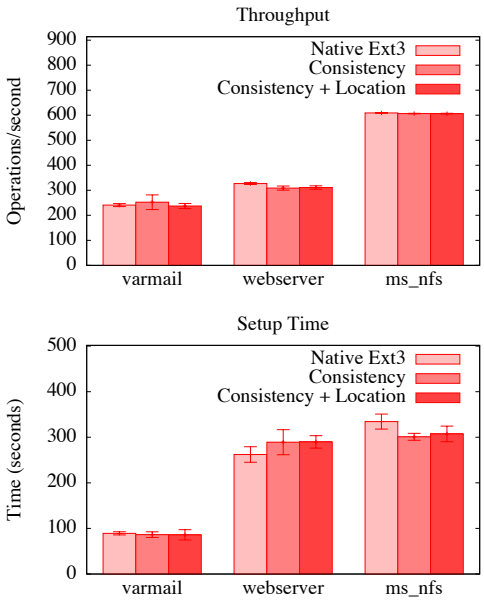


Figure 1: Performance on FileBench workloads for Ext3

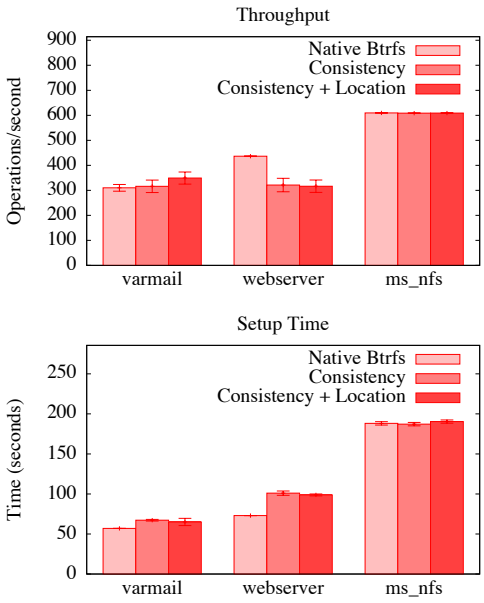


Figure 2: Performance on FileBench workloads for Btrfs

formance occurs for the webserver profile, which is a read-heavy benchmark. On closer analysis, we found that it generates significant metadata write traffic due to timestamp updates, which affects read performance. With the `noatime` mount option enabled, performance reaches roughly 90% of native performance.

6 Designing Checkable File Systems

Section 4.1 describes the four requirements of a runtime checker that make it feasible to check invariants efficiently at the block layer: 1) well-defined consistency points, 2) easily accessible allocation information, 3) easily distinguishable data versus metadata blocks, and 4) easily available block identity information. In this section, we describe how well various file systems meet these requirements. Table 2 provides a summary of our analysis. Then we recommend features that make file systems easily checkable at runtime.

6.1 Analysis of File System Design

No-Ordering FS: NoFS [6] aims to provide file system consistency in the face of poorly-behaved hardware that ignores ordering constraints and flush commands. They propose a novel commit-less approach to providing crash consistency by adding a backpointer to every block by using the out-of-band bytes provided by some devices, enabling atomic write of the block and its backpointer together. The backpointer makes it possible to identify the contents of blocks. NoFS performs block allocation based on an in-memory bitmap, thus avoiding any consistency issues between pointers and a persistent bitmap. Determining the allocation status at the block layer is expensive because it requires reading the block and its parent block to determine if a bidirectional pointer relationship exists between them. Unfortunately, NoFS does not provide any ordering guarantees by design, and thus lacks consistency points, and any consistency or location invariants. As a result, it is not possible to check any invariants in NoFS. Bugs in NoFS that cause data corruption would not be easily detectable by an offline checker as well.

FFS with Soft Updates: The soft updates mechanism provides crash consistency in an update-in-place file system without requiring journaling. Soft updates impose a partial order on writes and prevent cyclic dependencies between blocks by using a temporary in-memory rollback mechanism. Blocks and inodes can "leak" after a crash, but this problem is much less severe than blocks or inodes being overwritten while still in use. The ordering of writes allows some invariants to be checked (for example, you can't write a pointer to a newly-allocated block before you initialize the block). However, soft updates are not transactional and thus lack consistency points, and so most file system invariants cannot be checked because data might always be in flight.

Ext3: We have described the Ext3 file system properties in detail in Section 4.3. Ext3 provides consistency points and allocation information, but it mixes dynamically allocated metadata (directory data and indirect blocks) with data, thus requiring a full file system scan to distin-

	Consistency Points	Allocation Information	Separate Metadata	Block Identity
NoFS				X
Soft Updates		X		
Ext3	X	X		
REXt3	X	X	X	X
Btrfs	X	X	X	X

Table 2: Designing Checkable File Systems

guish data from metadata at the block layer. Instead, we distinguish the two types of blocks by retrofitting Ext3 with a metadata bitmap, as described in Section 4.3. In addition, the dynamically allocated metadata blocks cannot be easily identified, because they do not contain type information or information about the inode that points to them. We solve this problem by using in-memory back references in the checker.

REXt3: REXt3 [19] is a variant of ext3 that is optimized for a fast, offline file system checker called ffsck. Speeding up offline fsck involved two changes to the file system format, the co-location of metadata within metadata regions, and the addition of backpointers associating dynamically allocated metadata with their corresponding inodes. The separation of metadata and data into two regions makes it possible to distinguish between them with low overhead. With the addition of backpointers, the runtime checker for REXt3 will not need to use in-memory back references, thus reducing the memory overhead of the checker.

Btrfs: We have described the Btrfs properties in detail in Section 4.4. Btrfs provides consistency points, and it uses a separate extent tree to store allocation information. The extent records specify whether an allocated extent is data or metadata, and also record backpointers for the extent. Since Btrfs allows snapshots, some extents (both data and metadata) may have multiple parent blocks which point to them. A runtime checking system can identify metadata by its placement in a designated area, or by looking up the metadata flag in the extent tree. Furthermore, the contents of a metadata block can be identified based on the header structure shared by all metadata blocks. The shadow paging location invariants are easier to verify than their journaling equivalents because there is less state that needs to be tracked.

6.2 Design Recommendations

Based on our analysis of these file systems, we now suggest design features that enable efficient runtime checking of file systems. We expect that these same features will help implementing other file-system aware storage applications, such as differentiated storage services [22]. Consistency points are essential for runtime checking. While new file systems, possibly running on new hardware, may avoid providing consistency points, the resulting loss in protection is a serious issue. Easily accessible allocation information at the block layer, such as in bitmaps in fixed locations, allows enforcing location invariants efficiently. Other applications, like scrubbers and secure delete utilities, can also benefit from knowing the allocation status of a block. Separating data from metadata in well-defined regions allows distinguishing between them with low overhead because there is no need to lookup this information in bitmaps or trees. This approach also allows other policies, such as replication and placement, to be applied to contiguous metadata regions with ease. Fortunately, the mixing of metadata and data for performance reasons has been obsoleted by large disk caches [19]. Finally, backpointer information helps identify blocks at the block layer efficiently. This information is especially useful for dynamically allocated metadata in update-in-place file systems, because the checker may need to interpret an arbitrary block without knowing its position in the file system tree.

7 Related Work

We describe closely related work in the areas of runtime and offline file system consistency checking, and smart disk interfaces. Static bug finding tools [35] can reveal scores of bugs in file systems, but they can suffer from typical scalability issues, necessitating runtime checking. ZFS [5] uses a checksum-based runtime consistency checker for detecting and repairing file system corruption caused by storage hardware, e.g., latent sector errors, but it may not detect corruption caused by software bugs. Based on several requests, a check for location and some consistency invariants was added to Btrfs as a debugging tool [4]. These checks catch common errors, but they are embedded within the file system code itself, and so, for example, a file system bug could disable them. EnvyFS [3] uses N-version programming for detecting file system bugs at runtime. It uses the common VFS interface to pass each VFS-layer file system request to three child file systems and uses voting when returning results. The runtime overheads of this approach are high and subtle differences in file system semantics can make it hard

to compare results. HARDFS [8] detects software bugs in the Hadoop distributed file system (HDFS) at runtime by interposing on network messages and I/O, and verifies that the HDFS implementation behaves according to its operational specification. The verification state is compressed using bloom filters, significantly reducing the memory overhead. HARDFS can check certain end-to-end properties that a consistency checker cannot, such as whether a request was performed, but HARDFS does not attempt to catch all failures or guarantee that it will not raise false alarms.

Once a bug is detected at runtime, Membrane [31] proposes tolerating bugs by transparently restarting a failed file system. It assumes that file system bugs will lead to detectable, fail-stop crash failures. However, inconsistencies may have propagated to the on-disk metadata by the time the crash occurs. Our approach is complementary to Membrane, rather than waiting for the file system to crash, a restart could be initiated when a runtime checker detects an invariant violation.

Recently, there has been significant interest in improving the performance and robustness of offline consistency checkers. The rext3 file system [19] uses backpointers and collocates its metadata blocks, allowing its ffsck checker to scan the file system at rates close to the sequential bandwidth of the drive. Chunkfs [14] reduces the time to check consistency by breaking the file system into chunks that can be checked independent of each other. The SQCK offline consistency checker [13] expresses file system consistency properties declaratively, demonstrating that file system checks and repairs are more easily understood when expressed as SQL queries. It improves upon the repairs made by e2fsck by correcting the order in which certain repairs are performed and by using redundant information already provided by the file system. The SWIFT tool [2] tests the correctness of offline file system checker recovery code by leveraging the file system checker itself or by comparing the outputs of multiple checkers.

Our checker leverages ideas from semantically-smart disks [30], which use probing to gather detailed knowledge of file system behavior, allowing functionality or performance to be enhanced transparently at the block layer. Sivathanu et al. [29] provide a logic of file systems that helps reason about the correctness of smart disks. I/O shepherding [12] builds on smart disks, allows a file system developer to write reliability policies to detect and recover from a wide range of storage system failures. Unlike smart disks, a type-safe disk extends the disk interface by exposing primitives for block allocation [28], which helps enforce invariants such as preventing accesses to unallocated blocks.

8 Conclusion

We have presented the design of runtime file system checkers that can reliably detect file system bugs before they cause file system inconsistency. We show that the runtime checker needs to check location invariants on every write. These invariants enforce the atomicity and durability properties of the file system, helping preserve the integrity of committed transactions. Together with checking consistency properties on commit, the checker can provide the strong guarantee that every block write will preserve file system consistency.

We have implemented runtime checkers for the Ext3 journaling file system and the Btrfs copy-on-write file system. Our experimental results show that while consistency checking imposes some performance overhead, checking location invariants has almost no additional overhead. The Ext3 file system checker has low overhead but the Btrfs checker has higher overhead due to a higher metadata load. We are currently working on improving the Btrfs checker performance with better caching policies. Btrfs keeps a log to enable fast sync operations. We plan to implement our journaling invariants for this log. We expect that the checker overhead will be higher on faster storage devices, such as flash. We plan to evaluate this overhead in detail in the future.

We have shown that four file system features ease the design of runtime checkers, and enable checking invariants efficiently: 1) consistency points at which the file system is expected to be consistent on disk, 2) easily accessible allocation information at the block level, 3) distinguishable data versus metadata blocks at the block layer, and 4) backpointers for block typing and identification. We expect that these file system features will benefit other file-system aware storage applications as well.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Remzi Arpaci-Dusseau, for their detailed comments on this work. We also thank Andrei Soltan for designing and implementing the metadata bitmap for the Ext3 file system. We had many discussions and received feedback about this work from several members of the Computer Systems and Networking group and the SSRG group at the University of Toronto. Ali Hashemi provided invaluable system administration support. This research was supported by NSERC through the Discovery Grants and Graduate Scholarships programs.

References

[1] Bit stuffing. http://en.wikipedia.org/wiki/Bit_stuffing.

[2] Jo ao Carlos Menezes Carreira, Rodrigo Rodrigues, George Candea, and Rupak Majumdar. Scalable testing of file system checkers. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems (Eurosys)*, pages 239–252, 2012.

[3] Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Tolerating file-system mistakes with envyfs. In *Proceedings of the USENIX Technical Conference*, June 2009.

[4] Stephen Behrens. Btrfs: runtime integrity check tool, November 2011. http://lwn.net/Articles/466493.

[5] J. Bonwick and B. Moore. ZFS - The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

[6] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, February 2012.

[7] Helen Custer. *Inside the Windows NT File System*. Microsoft Press, 1994.

[8] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. HARDFS: Hardening HDFS with selective and lightweight versioning. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, February 2013.

[9] Chris Mason et al. Btrfs. http://btrfs.wiki.kernel.org.

[10] 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*, 8(4):15:1–15:29, December 2012.

[11] Duane Griffin. jbd: correctly unescape journal data blocks, March 2008. http://kerneltrap.org/mailarchive/git-commits-head/2008/3/20/1206404/thread.

[12] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving file system reliability with I/O shepherding. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 293–306, 2007.

[13] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A declarative file system checker. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, December 2008.

[14] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the Workshop on Hot Topics in System Dependability (HotDep)*, 2006.

[15] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Technical Conference*, 1994.

[16] Jan Kara. ext4: Always journal quota file modifications, June 2010. http://www.kerneltrap.org/mailarchive/linux-ext4/2010/6/2/6884775.

[17] Jan Kara. jbd: Write journal superblock with WRITE_FUA after checkpointing, April 2012. https://git.kernel.org/cgit/linux/kernel/git/tytso/ext4.git/commit/?id=fd2cbd4dfa3db477dd6226d387d3f1911d36a6a9.

[18] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the USENIX Conference on File and Storage Technolo-gies (FAST)*, February 2013.

[19] Ao Ma, Charlotte Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ffsck: The fast file system checker. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, February 2013.

[20] Peter Macko, Margo Seltzer, and Keith A. Smith. Tracking back references in a write-anywhere file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[21] Chris Mason, November 2011. https://git.kernel.org/cgit/linux/kernel/git/tytso/ext4.git/commit/?id=387125fc722a8ed432066b85a552917343bdafca.

[22] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 57–70, 2011.

[23] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[24] Rich Miller. Joyent Services Back After 8 Day Outage, January 2008. `http://www.datacenterknowledge.com/archives/2008/01/21/joyent-services-back-after-8-day-outage/`.

[25] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 206–220, 2005.

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

[27] Eric Sandeen. ext4: fix unjournaled inode bitmap modification, October 2012. `https://lwn.net/Articles/521819/`.

[28] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. Type-safe disks. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 15–28, 2006.

[29] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A logic of file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[30] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 73–88, 2003.

[31] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-dusseau, Remzi H. Arpaci-dusseau, and Michael M. Swift. Membrane: Operating system support for restartable file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[32] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the USENIX Technical Conference*, pages 1–14, 1996.

[33] Theodore Ts'o. Re: Apparent serious progressive ext4 data corruption bug in 3.6.3, October 2012. `https://lkml.org/lkml/2012/10/23/690`.

[34] Stephen C. Tweedie. Journalling the ext2fs filesystem. In *Proceedings of the 4th Annual Linux Expo*, May 1998.

[35] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, 2006.

[36] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: a ZFS case study. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express

Dejan Vučinić,[1] Qingbo Wang,[1] Cyril Guyot,[1] Robert Mateescu,[1] Filip Blagojević,[1] Luiz Franca-Neto,[1] Damien Le Moal,[1] Trevor Bunker,[2] Jian Xu,[2] Steven Swanson[2] and Zvonimir Bandić[1]
[1]*HGST San Jose Research Center,* [2]*University of California, San Diego*

Abstract

Phase Change Memory (PCM) presents an architectural challenge: writing to it is slow enough to make attaching it to a CPU's main memory controller impractical, yet reading from it is so fast that using it in a peripheral storage device would leave much of its performance potential untapped at low command queue depths, throttled by the high latencies of the common peripheral buses and existing device protocols.

Here we explore the limits of communication latency with a PCM-based storage device over PCI Express. We devised a communication protocol, dubbed DC Express, where the device continuously polls read command queues in host memory without waiting for host-driven initiation, and completion signals are eliminated in favor of a novel completion detection procedure that marks receive buffers in host memory with incomplete tags and monitors their disappearance. By eliminating superfluous PCI Express packets and context switches in this manner we are able to exceed 700,000 IOPS on small random reads at queue depth 1.

1 Introduction

The development of NAND flash and the market adoption of flash-based storage peripherals has exposed the limitations of a prior generation of device interfaces (SATA, SAS), prompting the creation of NVM Express [1] (NVMe), a simplified protocol for Non-Volatile Memory (NVM) storage attached to PCI Express. In the course of researching the capabilities of several novel memory technologies vying to displace flash, we set out to build NVMe-compliant prototypes as technology demonstrators. We found, however, that the maximal performance permitted by NVMe throttles the potential of many emerging memory cell technologies.

Phase Change Memory, one of the most promising contenders, achieves non-volatility by re-melting a material with two distinguishable solid phases to store two or more different bit values. Discovered in 1968 [2], this effect is today widely used in DVD-RW media, and is now making inroads into lithographed memory devices thanks to its favorable device size and scaling properties [3], high endurance [4] and very fast readout.

The most dramatic advantage PCM has over NAND flash is that its readout latency is shorter by more than two orders of magnitude. While its write latency is about fifty times longer than reads at current lithographic limits, it is already comparable with NAND flash and is expected to improve further with advances in lithography [5]. This makes PCM a very attractive alternative in the settings where the workload is dominated by reads.

The main motivation for the work we present in this paper is the desire to build a block storage device that takes advantage of the fast readout of PCM to achieve the greatest number of input-output operations per second (IOPS) permitted by the low physical latency of the memory medium. While spectacular numbers [6] of IOPS are touted for flash-based devices, such performance is only possible at impractically high queue depths. The fact remains that most practical data center usage patterns revolve around low queue depths [7, 8], especially under completion latency bounds [9]. The most critical metric of device performance in many settings is the round-trip latency to the storage device as opposed to the total bandwidth achievable: the latter scales easily with device bus width and speed, unlike the former. Under this more stringent criterion, modern flash-based SSDs top out around 13 kIOPS for small random reads at queue depth 1, limited by over 70 μs of readout latency of the memory medium (our measurements).

Here we describe how, starting from NVMe as the state of the art, we proceeded to slim down the read-side protocol by eliminating unnecessary packet exchanges over PCI Express and by avoiding mode and context switching. In this manner we were able to reduce the average round-trip protocol latency to just over 1 μs, a tenfold improvement over our current implementation of NVMe-compliant interface protocol. The resulting protocol, DC Express, exceeds 700 kIOPS at queue depth 1 on a simple benchmark with 512 B reads from PCM across a 4-lane 5 GT/s PCI Express interface, with modest impact on the total power consumption of the system.

We believe one cannot go much faster without retooling the physical link to the storage device.

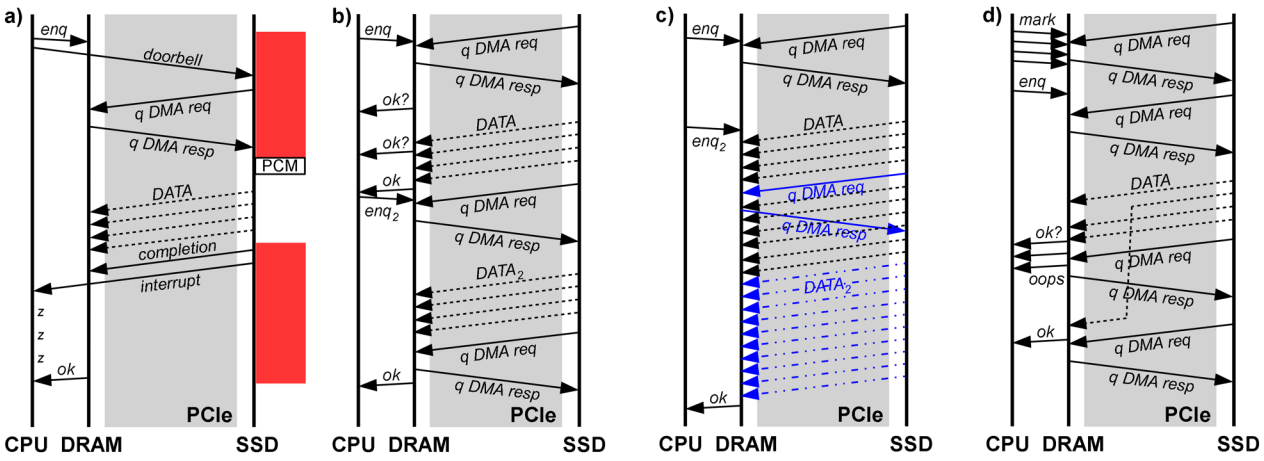


Figure 1: Timing diagrams illustrating the NVM Express and DC Express protocols. Time flows down, drawings are not to scale. a) NVMe: host CPU enqueues (***enq***) a command and rings doorbell; the device sends DMA request for the queue entry; the DMA response arrives; the command is parsed and data packets sent to the host DRAM, followed by completion queue entry and interrupt assertion; the host CPU thread handles interrupt. Red bars at right mark irreducible protocol latencies; rectangle illustrates the time when PCM is actually read. b) DC Express protocol at queue depth 1. There are no distinct doorbell nor completion signals. Device sends out DMA requests for new commands continuously. c) DC Express at higher queue depths. Subsequent DMA requests (***blue***) for new commands are launched prior to completing data transmission for the previous command (***black***) to take advantage of full-duplex nature of PCI Express and allow for seamless transmission. d) DC Express checks completion by marking each TLP-sized chunk of the receive buffer with an incomplete tag (***mark***) then monitoring for their disappearance. In case of out-of-order arrival the incomplete tag is found in one of the chunks (***oops***) prompting a longer wait for all the data to settle.

2 Endpoint-initiated queue processing

Prompted by the observation that the latency of one PCI Express packet exchange exceeds the time required to transfer a kilobyte of data, the approach we took to try to maximize the performance of a small read operation was to eliminate all unnecessary packet exchanges. At a minimum, even the leanest protocol requires a way to initiate and complete a transaction. In this section we describe endpoint-driven queue polling as an alternative to "doorbell" signals traditionally used for initiation; in the next section we discuss a minimalist way of signaling completion.

A rough outline of one NVMe-compliant read operation is shown in Figure 1a. The protocol for reading one block from the storage device begins with the host CPU preparing a read command in host DRAM and initiating the transaction by sending a "doorbell" packet over PCI Express. This is a signal to the device that there is a new read command waiting, hence it is to initiate a direct memory access (DMA) request—another PCI Express packet—to pick up that command from the queue in host DRAM.

Since every round trip over PCI Express incurs well over 0.6 μs latency on today's fastest hardware, with such a protocol we waste a microsecond of back-and-forth over the bus before the device can even commence the actual reading of data from the non-volatile storage medium. In the past, with the fundamental read latency of NAND flash between 25 and 80 μs, this extra request latency was but a small fraction of total transaction time and so was deemed negligible. In contrast, the fundamental latency to first byte read from a modern PCM chip is 110 ns, so now the protocol becomes severely limiting when trying to maximize the overall performance of the storage device for small random reads at queue depth 1.

The most important regime we strive to optimize for is at high load. In this case there will almost always be a new command waiting in the queue should the device ask for one, making the sending of a doorbell signal for every small read superfluous. In the quest for best performance under these conditions latency becomes the key factor, and a given fraction of "no news" transfers we treat as an acceptable overhead.

With this scenario in mind, and taking advantage of the full-duplex nature of PCI Express, we present the first key ingredient of DC Express in Figures 1b and 1c. The device keeps sending out requests for one or more commands in the read queue in host DRAM without waiting for doorbell signals, so that there is almost always a request "in flight." Further, since we can probe the actual round-trip latency for one DMA request to complete on given hardware, we can send anticipatory

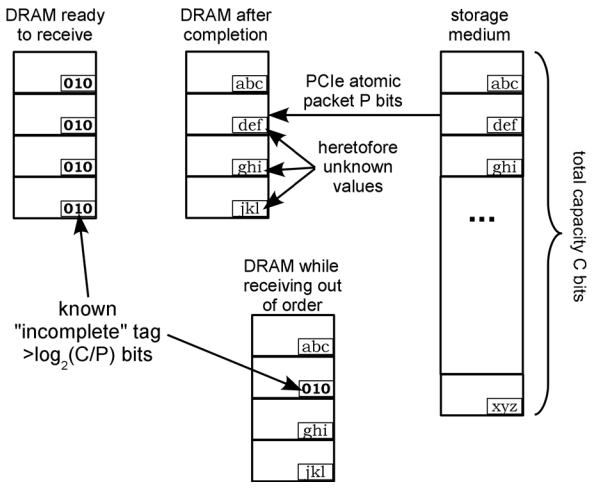


Figure 2: Detecting completion by pre-populating locations of packet trailing bits in the receive buffer with incomplete tags. The disappearance of all incomplete tags is a robust signal that the entire data transfer has completed.

queue read requests prior to sending all data packets for a previous request so that the next commands, if available, arrive at the device just in time when the device is able to service another command (Figure 1c).

3 Tagging receive buffers as incomplete

To notify the host process that a read operation from NVM has completed, an NVMe-compliant PCI Express endpoint writes an entry into a "completion" queue in host DRAM with a DMA transaction, followed by the assertion of an interrupt signal to wake up the sleeping thread (*cf*. Figure 1a). This protocol has two adverse performance implications in addition to the bandwidth consumed by the completion signal itself.

First, PCI Express allows for out-of-order arrival of transaction-level packets (TLPs), meaning that the possibility exists for the completion packet to settle into DRAM prior to all its data having arrived—which would open a window for data corruption of random duration (*cf*. Figure 1d). To ensure that all the data packets have reached host DRAM prior to issuing the completion signal, the endpoint must declare "strict packet ordering" for that traffic class by setting a particular bit in the TLP header. Since PCI Express flow control works by prior exchange of "transaction credits," one subtle negative effect of strict ordering is that any delayed data packet and all its successors, including the corresponding completion packet, will be holding up the credits available until all the transactions complete in turn, which can slow down the rest of PCI Express traffic.

Second, the context switching [10] and mode switching overhead of interrupt-based completion signaling can easily exceed the latency of a small PCM read operation by two orders of magnitude. On a modern x86 processor running Linux, two context switches between processes on the same core take no less than 1.1 μs, so it is imprudent to relinquish the time slice if the read from the storage device is likely to complete in less time. Even if the interrupt signal is ignored by a polling host CPU, the act of asserting it entails transmitting a packet over the PCI Express link—which again results in a small penalty on the maximal payload bandwidth remaining.

To avoid these performance penalties, we reasoned that the lowest latency test of payload's arrival into DRAM would be to simply poll the content of the trailing bits in the receive buffer from a CPU thread: seeing them change would be the signal that the read operation has completed. This spin-wait would not necessarily increase CPU utilization since the cycles spent waiting for request completions would otherwise be spent on context switching and interrupt handling.

There are two obstacles to implementing this simple protocol. As already mentioned, the individual TLPs that comprise one read operation may arrive into DRAM out of order, so the last word does not guarantee the arrival of the entire buffer. And, the CPU does not know what final bits to look for until they have already been read from the device.

The solution, and the second key ingredient of DC Express, we elaborate in Figures 1d and 2. Since the granularity of TLPs on a given PCI Express link is known, in addition to checking the trailing bits of the entire receive buffer the protocol also checks the trailing bits of every TLP-sized chunk of host DRAM. In the event of out-of-order packet reception, such checking will reveal a chunk that has not yet settled, as shown in the bottom panel of Figure 2.

Instead of looking for particular bit patterns to arrive into the trailing bits of every atomic transfer, we choose an "incomplete tag," a pre-selected bit pattern that does **not** appear in the data that is about to arrive from the device. The protocol then writes this known tag to the receive buffer prior to initiating the read operation, and looks for its disappearance from the location of every packet's trailing bits as the robust completion signal. In this way we are using the fast link from CPU to DRAM to avoid sending any extraneous bits or packets over the much slower PCI Express link.

3.1 Strategy for choosing incomplete tag value

Obviously, the bit pattern used for the incomplete tag in our scheme must be different from the trailing bits of every TLP arriving. If we choose a pattern of length greater than $\log_2(C/P)$ bits (*cf*. Figure 2), where C is the total capacity of the storage device and P the size of one TLP, then in principle we can always select a pattern such that no TLP arriving from that particular storage device at that time will have trailing bits that match our choice of pattern. Note that this characteristic of storage device interfaces is different from, for instance, network interface protocols, where we are not privy to the content of arriving data even in principle.

One very simple strategy for choosing the pattern for the incomplete tag is to pick it at random. Although probabilistic, this method is adequate for the vast majority of computing applications that have no hard real time latency bounds.

Let's illustrate for the case of a device with 128 GiB of PCM and 128 B size of TLP payload. Dividing device capacity by the TLP size,[1] there are 2^{30} possible values at the trailing end of any one TLP-sized transfer. If we set the size for the incomplete tag at 32 bits, a randomly generated 32-bit pattern will then have at most $2^{30}/2^{32}$ = 25% chance of being repeated somewhere on the storage device—the worst case scenario where every one of the 2^{30} possible patterns is present on the device. If the random choice was unlucky and the generated pattern is present on the device, that read operation will get stuck since the arrival of that packet will go unnoticed, *i.e*. there will be a "collision."

The strategy, then, is to pick the length of the tag such that we can declare the probability of collision to be low enough. If it encounters a collision, the protocol simply times out the stuck read operation and chooses a new tag at random before retrying. The time to timeout we set to the product of maximum queue depth and maximum latency to complete one read operation.

For applications that do have hard real time latency bounds it is possible to devise more complex strategies such that the incomplete tag value is always chosen so no collision is possible. This would be done at the storage device at first power-up and whenever a write to the device invalidates the existing choice of pattern. One such strategy would be for the device to pick values at random and compare internally with the current contents of the device. This would incur no communication overhead as the storage medium accesses would be confined to the PCM controller on the device. If even that much latency cannot be tolerated, additional computing resources can be provided in the device to monitor writes to keep track of intervals of values not present in the currently stored data so that the selection of a new tag can always complete in constant time.

[1] We assume only block-aligned reads are allowed.

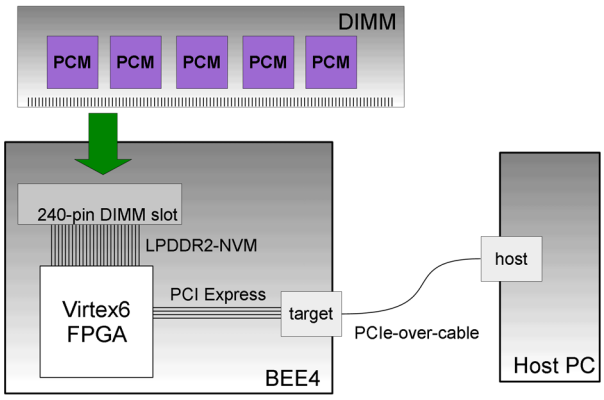


Figure 3: Diagram of our prototype system.

4 Performance

To implement DC Express we built a prototype NVM storage device (Figure 3) using a BEE4 FPGA platform (BEEcube, Inc., Fremont, CA) equipped with a custom-built DIMM card containing 5 Gib of Phase Change Memory (Micron NFR0A2B0D125C50). The NVM device exposed a 4-lane 5 GT/s ("gen2") PCI Express link from a Virtex6 FPGA running a custom memory controller that communicated with the PCM chips over the LPDDR2-NVM bus. The host systems used for testing included a Dell R720 server with an Intel Xeon E5-2690 CPU (Sandy Bridge-EP, TurboBoost to 3.8 GHz) and a Z77 Extreme4-M motherboard with an Intel i7-2600 CPU (Sandy Bridge, TurboBoost to 3.4 GHz). The NVM device was normally connected to the PCI Express lanes on the CPU dies. Alternatively, on the Z77 host we could use the lanes connecting to the Z77 chipset to measure the impact of the retransmission. All measurements were done on

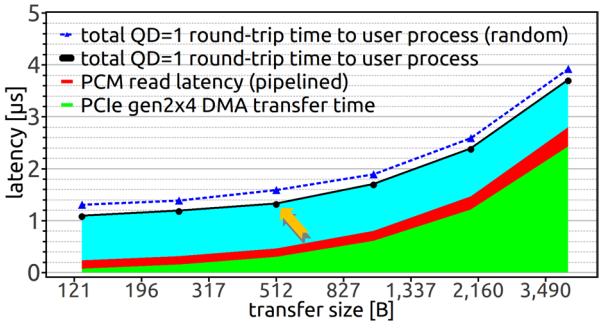


Figure 4: Average latency of a small random read operation when using the DC Express protocol at queue depth 1.

component	latency [µs]	kIOPS
data transfer (4 kiB)	2.432	
PCM read	0.368	
protocol + command parsing	0.863	273(1)
block driver	0.99	
read() call (kernel entry/exit)	1.17	
fio	0.506	158(2)

Table 5: Breakdown of contributions to average round-trip latency of DC Express for 4 kiB random reads at queue depth 1. IOPS were measured from a user space process (1) or linux block device driver (2). The total latency to a given layer is the sum of all latencies above it.

Linux kernel version 3.5 (Ubuntu and Fedora distributions).

We first exercised the bare protocol from a user space process by mmap()-ing a kernel buffer where the queues and receive buffer locations were pre-allocated. This allowed measurement of raw performance without mode or context switching overhead. The results are shown in Figure 4 for different transfer sizes. We designed the NVM device so that the bandwidth of data retrieval from PCM matches that of PCI Express transmission. Therefore, only the initial PCM row activation and local LPDDR2-NVM memory bus overhead (*red*) contribute to the irreducible protocol latency; the remainder is pipelined with PCI Express transfer (*green*). The remaining (*cyan*) component consists of PCI Express packet handling and command parsing, in addition to the polling from both ends of the link.

When we exercise the protocol in a tight loop, or with predictable timing in general, we can adjust the endpoint polling to anticipate the times of arrival of new commands into the read queue so that a new command gets picked up by the queue DMA request soon after its arrival into the queue. The total round-trip latency for this use case (shown by the solid black line in Figure 4) we measured as the inverse of the total number of read operations executed in a tight loop. For traditional 512 B blocks (arrow in Figure 4) the total latency seen by a user-space process averages 1.4 μs, over 700,000 IOPS.

If we fully randomize read command arrival times so that no predictive optimization of endpoint-driven queue polling is possible, there is additional latency incurred by the average delay between the arrival of a read command into the queue and the time when the next queue DMA hits. For this use case we measured the completion latencies using Intel CPU's time stamp counter (dashed blue line in Figure 4).

Next we constructed a lightweight block device driver to measure the impact of kernel entry and exit. We derived our driver from the Linux ramdisk device example. The read block size was limited to 4 kiB. We list the additional latencies in Table 5. One memory-to-memory copy of the retrieved block accounts for a small fraction of the time spent inside the block driver. Note that the tool used for measuring the latency of the block device, fio, contributes a significant amount of its own latency to these measurements. For comparison, our current implementation of NVMe-compliant device accessed through the Linux NVMe device driver under similar conditions reaches 78 kIOPS at queue depth 1, nearly 13 μs per 4 kiB read operation.

The latencies measured on the i7 system were comparable to those on the E5 server system when our device was connected to CPU lanes. Routing the packets through the Z77 chipset resulted in about two microseconds of additional latency per PCI Express round trip.

4.1 Power and congestion considerations

One concern with a protocol that continuously queries the host DRAM for new commands is the waste of resources at idle. To better understand the magnitude of this component relative to the baseline idle consumption of a modern server configuration, for this exercise we disabled all but one core on the single socket populated by the E5-2690 on the Dell R720 server equipped with 16 GiB of DDR3-1600 DRAM.

In Figure 6 we show the dependence of DC Express protocol performance and system power usage on the clock frequency of the CPU core doing the spin-wait. As expected, higher polling frequency reduces the average round-trip latency. Surprisingly, the optimal operating point, as defined by the Joules-per-IOP measure, is not at the lowest core frequency. Dominated by the

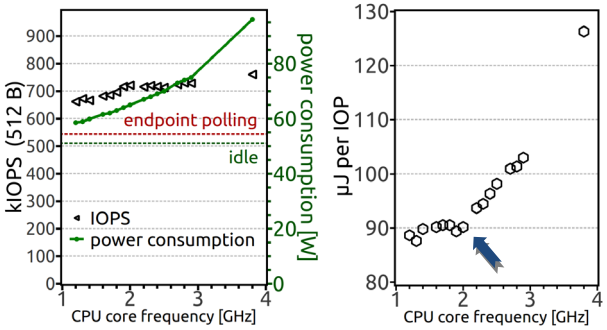


Figure 6: DC Express protocol performance for 512 B packets and total server power consumption as a function of the E5-2690 CPU core frequency.

significant idle power consumption of the entire server, the energy cost of one read operation stays relatively flat at low clock settings, suggesting a cost-optimal operating point near 2 GHz for this configuration (arrow in Figure 6) before hardware depreciation is taken into account.

Note that the overall impact of constant polling from the PCI Express endpoint is modest, about six percent of idle power consumption of the server. This is the worst case scenario where there is always a DMA request in flight, *i.e.* at queue depth 1 every other read of the command queue is guaranteed to be wasted. In this regime, fetching one 64 B command at a time would tie up less than six percent of the upstream PCI Express bandwidth.

5 Discussion

In this paper we described our attempts to wring the last drop of performance out of the widely adopted PCI Express interface, driven by the possibility of much higher performance frontiers uncovered by Phase Change Memory and other emerging non-volatile storage technologies. By eliminating unnecessary packet exchanges and avoiding context and mode switching we were able to surpass 700,000 IOPS at queue depth 1 when reading from a PCM storage device on commodity hardware. The performance increases further for smaller transfers to just under a million reads per second, the hard limit set by bus and protocol latency. By increasing the number of PCI Express lanes or the per-lane bandwidth it will be possible in the future to asymptotically approach this limit with larger transfers, but going even faster will require a fundamental change to the bus.

The unsolicited polling of DRAM from the endpoint to check for presence of new read commands results in a reduction in average protocol latency, but at the expense of slightly higher idle power consumption. We have shown that the worst-case impact is modest, both on power consumption and on the remaining PCI Express bandwidth. In settings with high load variability this component of overall power usage can be greatly mitigated ever further by, for instance, making the switch to DC Express at a given load threshold while reverting to the traditional "doorbell" mode of operation at times of low load.

Our focus was exclusively on small random reads, as that is the most interesting regime where PCM greatly outperforms the cheaper NAND flash. Write latency of the current generation of PCM is 55 times higher than read latency, so we did not attempt to modify the write-side protocol as the performance benefit would be small. For new memory technologies with much lower write latencies, *e.g.* STT-MRAM [11], a similar treatment of the write-side protocol could result in similarly large round-trip latency improvements, and will be the subject of future work.

Prior work on accessing low-latency NVMs over PCI Express has elaborated the advantages of polling over interrupts [12]. Our work goes two steps further: we introduce polling from both ends of the latency-limiting link, and we do away with the separate completion signal in favor of low-latency polling on all atomic components of a compound transfer.

While the advance in read performance we report is quite dramatic, it is important to note the high cost of using our protocol through kernel facilities. To maximize the read performance of PCM storage we resorted to a user-space library which did not provide security. To take advantage of the low latency while still enjoying safety guarantees from the operating system one must implement an additional protocol layer of negotiation through the kernel, such as *Moneta Direct* [13].

Our work casts PCM-based peripheral storage in a new light. Rather than using it in the traditional fashion, just like spinning disk of yore, we envision a new storage tier that fills a niche between DRAM and NAND flash. Using our FAST protocol will enable exposing very large non-volatile memory spaces that can still be read in-context with intermediate read latencies but without the several Watts per gigabyte penalty of DRAM refresh. On the other hand, treating PCM as block storage alleviates the need to rethink the cache hierarchy of contemporary CPUs, which would be necessary to achieve reasonable write performance in architectures where PCM is the main and only memory.

Beyond our work, almost an order of magnitude of further improvement in small random read latency is possible in principle before we hit the limits of the underlying physics of phase change materials. At this time, such advances would require either the use of parallel main memory buses together with deep changes to the cache hierarchy, or the use of fundamentally different high speed serial buses, such as HMCC [14], with shorter minimal transaction latencies. The latter, while promising, is still in the future, and is geared toward devices soldered onto motherboards as opposed to field-replaceable peripheral cards. It therefore appears that the niche for low read latency PCI Express peripheral storage based on Phase Change Memory is likely to persist until the arrival of future generations of peripheral buses and CPUs.

References

[1] http://nvmexpress.org/wp-content/uploads/2013/05/NVM_Express_1_1.pdf

[2] Ovshinsky, Stanford R. "Reversible electrical switching phenomena in disordered structures." *Physical Review Letters* **20**: 1450–1453, 1968.

[3] Servalli, G. "A 45nm generation phase change memory technology." *Electron Devices Meeting (IEDM), 2009 IEEE International*. IEEE, 2009.

[4] Goux, L. *et al*., "Degradation of the Reset Switching During Endurance Testing of a Phase-Change Line Cell." *IEEE Transactions on Electron Devices* vol.56(2), pp.354–358, 2009.

[5] Loke, D., et al. "Breaking the speed limits of phase-change memory." *Science* 336.6088: 1566–1569, 2012.

[6] Fusion-io: http://www.fusionio.com/overviews/9m-iops-technology-showcase/

[7] Seltzer, M., Chen, P. and Ousterhout, J. "Disk scheduling revisited." *Proceedings of the Winter 1990 USENIX Technical Conference*. USENIX Association, 1990.

[8] Personal communications with customers.

[9] Stanovich, Mark J., Baker, Theodore P., and Wang, An-I A. "Throttling on-disk schedulers to meet soft-real-time requirements." *Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2008.

[10] Li, C., Ding, C., and Shen, K. "Quantifying the cost of context switch." *ACM Workshop on Experimental Computer Science*. ACM, 2007.

[11] Huai, Yiming, et al. "Observation of spin-transfer switching in deep submicron-sized and low-resistance magnetic tunnel junctions." *Applied Physics Letters* 84.16: 3118-3120, 2004.

[12] Yang, J., Minturn, D. B., and Hady, F. "When poll is better than interrupt." *Proceedings of the 10th USENIX conference on File and Storage Technologies*. USENIX Association, 2012.

[13] Caulfield, Adrian M., et al. "Providing safe, user space access to fast, solid state disks." *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2012.

[14] http://www.hybridmemorycube.org/

MultiLanes: Providing Virtualized Storage for OS-level Virtualization on Many Cores

Junbin Kang, Benlong Zhang, Tianyu Wo, Chunming Hu, and Jinpeng Huai
Beihang University, Beijing, China
{*kangjb, woty, hucm*}*@act.buaa.edu.cn, zblgeqian@gmail.com, huaijp@buaa.edu.cn*

Abstract

OS-level virtualization is an efficient method for server consolidation. However, the sharing of kernel services among the co-located *virtualized environments* (VEs) incurs performance interference between each other. Especially, interference effects within the shared I/O stack would lead to severe performance degradations on many-core platforms incorporating fast storage technologies (e.g., non-volatile memories).

This paper presents MultiLanes, a virtualized storage system for OS-level virtualization on many cores. MultiLanes builds an isolated I/O stack on top of a virtualized storage device for each VE to eliminate contention on kernel data structures and locks between them, thus scaling them to many cores. Moreover, the overhead of storage device virtualization is tuned to be negligible so that MultiLanes can deliver competitive performance against Linux. Apart from scalability, MultiLanes also delivers flexibility and security to all the VEs, as the virtualized storage device allows each VE to run its own guest file system.

The evaluation of our prototype system built for Linux container (LXC) on a 16-core machine with a RAM disk demonstrates MultiLanes outperforms Linux by up to 11.32X and 11.75X in micro- and macro-benchmarks, and exhibits nearly linear scalability.

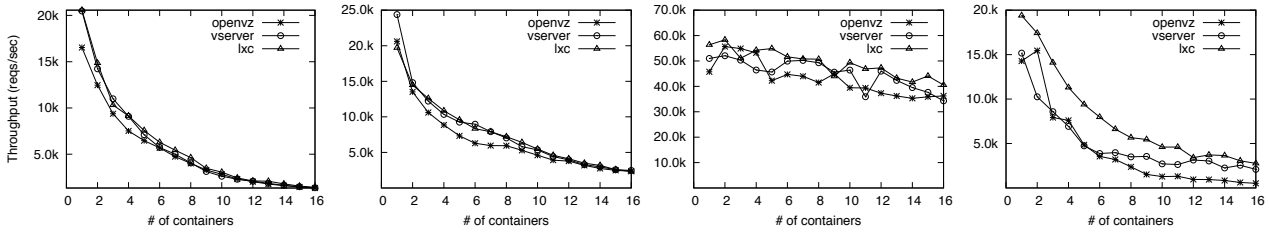
1 Introduction

As many-core architectures exhibit powerful computing capacity, independent workloads can be consolidated in a single node of data centers for high efficiency. Operating system level virtualization (e.g., VServer [32], OpenVZ [6], Zap [29], and LXC [4]) is an efficient method to run multiple virtualized environments (VEs) for server consolidation, as it comes with significantly lower overhead than hypervisors [32, 29]. Thus, each independent workload can be hosted in a VE for both good isolation and high efficiency [32]. Previous work on OS-level virtualization mainly focuses on how to efficiently space partition or time multiplex the hardware resources (e.g., CPU, memory and disk).

However, the advent of non-volatile memory technologies (e.g., NAND flash, phase change memories and memristors) creates challenges for system software. Specially, emerging fast storage devices built with non-volatile memories deliver low access latency and enormous data bandwidth, thus enabling high degree of application-level parallelism [16, 31]. This advance has shifted the performance bottleneck of the storage system from poor hardware performance to system software inefficiencies. Especially, the sharing of the I/O stack would incur performance interference between the co-located VEs on many cores, as the legacy storage stack scales poorly on many-core platforms [13]. A few scalability bottlenecks exist in the Virtual File System (VFS) [23] and the underlying file systems. As a consequence, the overall performance of the storage system suffers significant degradations when running multiple VEs with I/O intensive workloads. The number of concurrently running VEs may be limited by the software bottlenecks instead of the capacity of hardware resources, thus degrading the utilization of the hardware.

This paper presents MultiLanes, a storage system for operating system level virtualization on many cores. MultiLanes eliminates contention on shared kernel data structures and locks between co-located VEs by providing an isolated I/O stack for each VE. As a consequence, it effectively eliminates the interference between the VEs and scales them well to many cores. The isolated I/O stack design consists of two components: the virtualized block device and the partitioned VFS.

The virtualized block device. MultiLanes creates a file-based virtualized block device for each VE to run a guest file system instance atop it. This approach avoids contention on shared data structures within the file system layer by providing an isolated file system stack for



(a) Ext3 (b) Ext4 (c) XFS (d) Btrfs

Figure 1: **VE Scalability Evaluation.** *This figure shows the average throughput of each container performing sequential buffered writes on different file systems. We choose the latest OpenVZ, Linux-VServer and LXC that are based on Linux kernel 2.6.32, 3.7.10, and 3.8.2 respectively. The details of experimental setup is to be presented in Section 5.*

Ext3				**Ext4**			
#	lock	contention bounces	total wait time	#	lock	contention bounces	total wait time
1	zone->wait_table	5216186	36574149.95	1	journal->j_list_lock	2085109	138146411.03
2	journal->j_state_lock	1581931	56979588.44	2	zone->wait_table	147386	384074.06
3	journal->j_list_lock	382055	20804351.46	3	journal->j_state_lock	46138	541419.08
XFS				**Btrfs**			
#	lock	contention bounces	total wait time	#	lock	contention bounces	total wait time
1	zone->wait_table	22185	36190.48	1	found->lock	778055	44325371.60
2	rq->lock	6798	9382.04	2	btrfs-log-02	387846	1124781.19
3	key#3	4869	13463.40	3	btrfs-log-01	230158	1050066.24

Table 1: **The Top 3 Hottest Locks.** *This table shows the contention bounces and total wait time of the top 3 hottest locks when running 16 LXC containers with buffered writes. The total wait time is in us.*

each VE. The key challenges to the design of the virtualized block device are (1) how to tune the overhead induced by the virtualized block device to be negligible, and (2) how to achieve good scalability with the number of virtualized block devices on the host file system which itself scales poorly on many cores.

Hence, we propose a set of techniques to address these challenges. First, MultiLanes uses a synchronous bypass strategy to complete block I/O requests of the virtualized block device. In particular, it translates a block I/O request from the guest file system into a list of requests of the host block device using block mapping information got from the host file system. Then the new requests will be directly delivered to the host device driver without the involvement of the host file system. Second, MultiLanes constrains the work threads interacting with the host file system for block mapping to a small set of cores to avoid severe contention on the host, as well as adopts a prefetching mechanism to reduce the communication costs between the virtualized devices and the work threads.

Another alternative for block device virtualization is to give VEs direct accesses to physical devices or logical volumes for native performance. However, there are several benefits in adopting plain files on the host as the back-end storage for virtualization environments [24]. First, using files allows storage space overcommitment as most modern file systems support sparse files (e.g., Ext3/4 and XFS). Second, it also eases the management of VE images as we can leverage many existing file-based storage management tools. Third, snapshotting an image using copy-on-write is simpler at the file level than the block device level.

The partitioned VFS. In Unix-like operating systems, VFS provides a standard file system interface for applications to access different types of concrete file systems. As it needs to maintain a consistent file system view, the inevitable use of global data structures (e.g., the inode cache and dentry cache) as well as the corresponding locks might result in scalability bottlenecks on many cores. Rather than iteratively eliminating or mitigating the scalability bottlenecks of the VFS [13], MultiLanes in turn adopts a straightforward strategy that partitions the VFS data structures to completely eliminate contention between co-located VEs, as well as to achieve improved locality of the VFS data structures on many cores. The partitioned VFS is referred to as the pVFS in the rest of the paper.

The remainder of the paper is organized as follows. Section 2 highlights the storage stack bottlenecks in existing OS-level virtualization approaches for further motivation. Then we present the design and implementation of MultiLanes in Section 3 and Section 4 respectively. Section 5 evaluates its performance and scalability with micro- and macro-benchmarks. We discuss the related works in Section 6 and conclude in Section 7. A virtualized environment is referred to as a *container* in the following sections also.

2 Motivation

In this section, we create a simple microbenchmark to highlight the storage stack bottlenecks of existing OS-level virtualization approaches on many-core platforms incorporating fast storage technologies. The benchmark performs 4KB sequential writes to a 256MB file. We run the benchmark program inside each container in parallel and vary the number of containers. Figure 1 shows the average throughput of containers running the benchmark on a variety of file systems (i.e., Ext3/4, XFS and Btrfs). The results show that the throughput on all the file systems except XFS decreases dramatically with the increasing number of containers in the three OS-level virtualization environments (i.e., OpenVZ, VServer and LXC). The kernel lock usage statistics in Table 1 presents the lock bounces and total wait time during the benchmarking, which results in the decreased performance. XFS delivers much better scalability than the other three as much less contention occurred for buffered writes. Nevertheless it would also suffer from scalability bottlenecks under other workloads, which will be described in Section 5.

The poor scalability of the storage system is mainly caused by the concurrent accesses to shared data structures and the use of synchronization primitives. The use of shared data structures modified by multiple cores would cause frequent transfers of the data structures and the protecting locks among the cores. As the access latency of remote caches is much larger than that of local caches on modern shared-memory multicore processors [12], the overhead of frequent remote accesses would significantly decrease the overall system performance, leading to severe scalability bottlenecks. Especially, the large traffic of non-scalable locks generated by cache coherence protocols on the interconnect will exacerbate system performance. Previous studies show that the time taken to acquire a lock will be proportional to the number of contending cores [13, 12].

3 MultiLanes Design

MultiLanes is a storage system for OS level virtualization that addresses the I/O performance interference between the co-located VEs on many cores. In this section, we present the designing goals, concepts and components of MultiLanes.

3.1 Design Goals

Existing OS-level virtualization approaches simply leverage *chroot* to realize file system virtualization [32, 6, 29]. The containers co-located share the same I/O stack, which not only leads to severe performance interference between them but also suppresses flexibility.

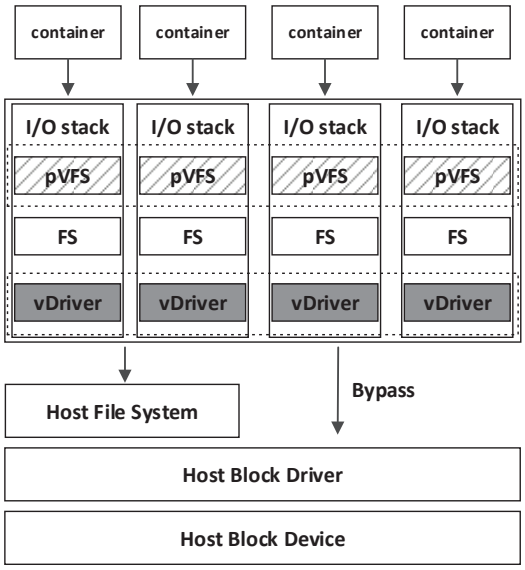


Figure 2: **MultiLanes Architecture.** *This figure depicts the architecture of MultiLanes. The virtualized storage is mapped as a plain file on the host file system and is left out in the figure.*

MultiLanes is designed to eliminate storage system interference between containers to provide good scalability on many cores. We aim to meet three design goals: (1) it should be conceptually simple, self-contained, and transparent to applications and to various file systems; (2) it should achieve good scalability with the number of containers on the host; (3) it should minimize the virtulization overhead on fast storage media so as to offer near-native performance.

3.2 Architectural Overview

MultiLanes is composed of two key design modules: the virtualized storage device and the pVFS. Figure 2 illustrates the architecture and the overall primary abstractions of the design. We have left out other kernel components to better focus on the I/O subsystem.

At the top of the architecture we host multiple containers. A container is actually a group of processes which are completely constrained to execute inside it. Each container accesses its guest file system through the partitioned VFS that provides POSIX APIs. The partitioned VFS offers a private kernel abstraction to each container to eliminate contention within the VFS layer. Under each pVFS there lies the specific guest file system of the container. The pVFS remains transparent to the underlying file system by providing the same standard interfaces with the VFS.

Between the guest file system and the host are the virtualized block device and the corresponding customized

block device driver. MultiLanes maps regular files in the host file system as virtualized storage devices to containers, which provides the fundamental basis for running multiple guest file systems. This storage virtualization approach not only eliminates performance interference between containers in the file system layer, but also allows each container to use a different file system from each other, which enables flexibility both between the host and a single guest, and between the guests. The virtualized device driver is customized for each virtualized device, which provides the standard interfaces to the Linux generic block layer. Meanwhile, MultiLanes adopts a proposed synchronous bypass mechanism to avoid most of the overhead induced by the virtualization layer.

3.3 Design Components

MultiLanes provides an isolated I/O stack to each container to eliminate performance interference between containers, which consists of the virtualized storage device, the virtualized block device driver, and the partitioned VFS.

3.3.1 Virtualized Storage

Compared to full virtualization and para-virtualization that provide virtualized storage devices for virtual machines (VMs), OS-level virtualization stores the VMs' data directly on the host file system for I/O efficiency. However, the virtualized storage has inborn advantage over shared storage in performance isolation because each VM has an isolated I/O stack.

As described in Section 2, the throughput of each LXC container will fall dramatically with the increasing number of containers due to the severe contention on shared data structures and locks within the shared I/O stack. The interference is masked by the high latency of the sluggish mechanical disk in traditional disk-based storage. But it has to be reconsidered in the context of next generation storage technologies due to the shift that system software becomes the main bottleneck on fast storage devices.

In order to eliminate storage system performance interference between containers on many cores, we provide lightweight virtualized storage for each container. We map a regular file as a virtualized block device for each container, and then build the guest file system on top of it. Note that as most modern file systems support sparse files for disk space efficiency, the host doesn't preallocate all blocks in accordance with the file size when the file system is built on the back-end file. The challenge is to balance performance gain achieved by performance isolation against the overhead incurred by storage virtualization. However, scalability and competitive performance can both be achieved when the virtualized storage architecture is efficiently devised.

3.3.2 Driver Model

Like any other virtualization approaches adopted in other fields, the most important work for virtualization is to establish the mapping between the virtualized resources and the physical ones. This is done by the virtualized block device driver in MultiLanes. As shown in Figure 2, each virtualized block device driver receives block I/O requests from the guest file system through the Linux generic block layer and maps them to requests of the host block device.

A block I/O request is composed of several segments, which are contiguous on the block device, but are not necessarily contiguous in physical memory, depicting a mapping between a block device sector region and a list of individual memory segments. On the block device side, it specifies the data transfer start sector and the block I/O size. On the buffer side, the segments are organized as a group of I/O vectors. Each I/O vector is an abstraction of a segment that is in a memory page, which specifies the physical page on which it lies, offset relative to the start of the page, and the length of the segment starting from the offset. The data residing in the block device sector region would be transmitted to/from the buffer in sequence according to the data transfer direction given in the request.

For the virtualized block device of MultiLanes, the sector region specified in the request is actually a data section of the back-end file. The virtualized driver should translate logical blocks of the back-end file to physical blocks on the host, and then map each I/O request to the requests of the host block device according to the translation. It is composed of two major components: the block translation and block handling.

Block Translation. Almost all modern file systems have devised a mapping routine to map a logical block of a file to the physical block on the host device, which returns the physical block information to the caller at last. If the block is not mapped, the mapping process involves the block allocation of the file system. MultiLanes achieves block translation with the help of this routine.

As shown in Figure 3, the block translation unit of each virtualized driver consists of a cache table, a job queue and a translation thread. The cache table maintains the mapping between logical blocks and physical blocks. The virtulized driver will first look up the table with the logical block number of the back-end file for block translation when a container thread submits an I/O request to it. Note that the driver actually executes in the context of the container thread as we adopt a synchronous model

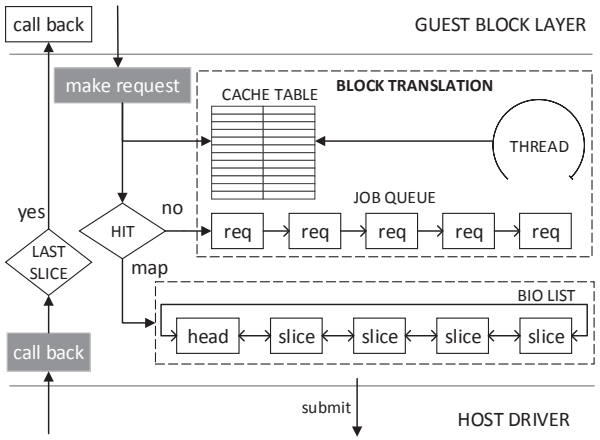


Figure 3: **Driver Structure.** *This figure presents the structure of the virtualized storage driver, which comprises the block translation unit and the request handling unit.*

of I/O request processing. If the target block is hit in the cache table the driver directly gets the target mapping physical block number. Otherwise it starts a cache miss event and then puts the container thread to sleep. A cache miss event delivers a translation job to the job queue and wakes up the translation thread. The translation thread then invokes the interface of the mapping routine exported by the host file system to get the target physical block number, stores a new mapping entry in the cache table, and wakes up the container thread at last. The cache table is initialized as empty when the virtualized device is mounted.

Block translation will be extremely inefficient if the translation thread is woken up to only map a single cache miss block every time. The driver will suffer from frequent cache misses and thread context switches, which would waste CPU cycles and cause considerable communication overhead. Hence we adopt a prefetching approach similar to that of handling CPU cache misses. The translation thread maps a predefined number of continuous block region starting from the missed block for each request in the job queue.

On the other hand, as the block mapping of the host file system usually involves file system journaling, the mapping process may cause severe contention within the host on many cores when cache misses of multiple virtulized drivers occur concurrently, thus scaling poorly with the number of virtualized devices on many cores. We address this issue by constraining all translation threads to work on a small set of cores to reduce contention [18] and improve data locality on the host file system. Our current prototype binds all translation threads to a set of cores inside a processor, due to the observation that sharing data within a processor is much less expensive than that crossing processors [12].

Request Handling. Since the continuous data region of the back-end file may not be necessarily continuous on the host block device, a single block I/O request of the virtualized block device may be remapped to several new requests according to the continuity of the requested blocks on the host block device.

There are two mapping involved when handling the block I/O requests of the virtualized block device. The mapping between the memory segments and the virtualized block device sector region is specified in a scatter-gather manner. The mapping between the virtualized block device and the host block device gives the physical block number of a logical block of the back-end file. For simplicity, the block size of the virtualized block device should be the same with that of the host block device in our current prototype. For each segment of the block I/O request, the virtulized device driver first gets the logical block number of it, then translates the logical block number to the physical block number with the support of the block translation unit. When all the segments of a request are remapped, we have to check whether they are contiguous on the host block device. The virtualized device driver combines the segments which are contiguous on the host block device as a whole and allocates a new block I/O request of the host block device for them. Then it creates a new block I/O request for each of the remaining segments. Thus a single block I/O request of the virtualized block device might be remapped to several requests of the host block device. Figure 4 illustrates such an example, which will be described in Section 4.1.

A new block I/O request is referred to as a slice of the original request. We organize the slices in a doubly-linked list and allocate a head to keep track of them. When the list is prepared, each slice would be submitted to the host block device driver in sequence. The host driver will handle the data transmission requirements of each slice in the same manner with regular I/O requests.

I/O completion should be carefully handled for the virtualized device driver. As the original request is split into several slices, the host block device driver will initiate a completion procedure for each slice. But the original request should not be terminated until all the slices have been finished. Hence we offer an I/O completion callback, in which we keep track of the finished slices, to the host driver to invoke when it tries to terminate each slice. The host driver will terminate the original block I/O request of the virtualized block device driver only when it finds out that it has completed the last slice.

Thus a block I/O request of MultiLanes is remapped to multiple slices of the host block device and is completed by the host device driver. The most important feature of the virtualized driver is that it stays transparent to the guest file system and the host block device driver, and only requires minor modification to the host file system to export the mapping routine interface.

3.3.3 Partitioned VFS

The virtual file system in Linux provides a generic file system interface for applications to access different types of concrete file systems in a uniform way. Although MultiLanes allows each container to run its own guest file system independently, there still exists performance interference within the VFS layer. Hence, we propose the partitioned VFS that provides a private VFS abstraction to each container, eliminating the contention for shared data structures within the VFS layer between containers.

#	Hot VFS Locks	Hot Invoking Functions
1	inode_hash_lock	insert_inode_locked() __remove_inode_hash()
2	dcache_lru_lock	dput() dentry_lru_prune()
3	inode_sb_list_lock	evict() inode_sb_list_add()
4	rename_lock	write_seqlock()

Table 2: **Hot VFS Locks.** *The table shows the hot locks and the corresponding invoking functions in VFS when running the metadata intensive microbenchmark ocrd in Linux kernel 3.8.2.*

Table 2 shows the top four hottest locks in VFS when conducting the metadata-intensive microbenchmark *ocrd*, which will be described in Section 5. VFS maintains an inode hash table to speed up inode lookup and uses the *inode_hash_lock* to protect the list. Inodes that belong to different super blocks are hashed together into the hash table. Meanwhile, each super block has a list that links all the inodes that belong to it. Although this list is independently managed by each super block, the kernel uses the global *inode_sb_list_lock* to protect accesses to all lists, which would introduce unnecessary contention between multiple file system instances.

For the purpose of path resolution speedup, VFS uses a hash table to cache directory entries, which allows concurrent read accesses to it without serialization by using Read-Copy-Update (RCU) locks [27]. The *rename_lock* is a sequence lock that is indispensable for the hash table in this context because a rename operation may involve the edition of two hash buckets which might cause false lookup results. It is also inappropriate that the VFS protects the LRU dentry lists of all file system instances with the global *dcache_lru_lock*.

Rather than iteratively fixing or mitigating the lock bottlenecks in the VFS, we in turn adopts a straightforward approach that partitions the VFS data structures and corresponding locks to eliminate contention, as well as to improve locality of the VFS data structures. In particular, MultiLanes allocates an inode hash table and a dentry hash table for each container to eliminate the performance interference within the VFS layer. Along with the separation of the two hash tables from each other, *inode_hash_lock* and *rename_lock* are also separated. Meanwhile, each guest file system has its own *inode_sb_list_lock* and *dcache_lru_lock* also.

By partitioning the resources that would cause contention in the VFS, the VFS data structures and locks become localized within each partitioned domain. Supposing there are *n* virtualized block devices built on the host file system, the original VFS domain now is split into *n+1* independent domains: each guest file system domain and the host domain that serves the host file system along with special file systems (e.g., procfs and debugfs). We refer the partitioned VFS to as the pVFS. The pVFS is an important complementary part of the isolated I/O stack.

4 Implementation

We choose to implement the prototype of MultiLanes for Linux Container (LXC) out of OpenVZ and Linux-VServer due to that both OpenVZ and Linux-VServer need customized kernel adaptations while LXC is always supported by the latest Linux kernel. We implemented MultiLanes in the Linux 3.8.2 kernel, which consists of a virtualized block device driver module and adaptations to the VFS.

4.1 Driver Implementation

We realize the virtualized block device driver based on the Linux loop device driver that provides the basic functionality of mapping a plain file as a storage device on the host.

Different from traditional block device drivers that usually adopt a request queue based asynchronous model, the virtualized device driver of MultiLanes adopts a synchronous bypass strategy. In the routine *make_request_fn*, which is the standard interface for delivering block I/O requests, our driver finishes request mapping and redirects the slices to the host driver via the standard *submit_bio* interface.

When a virtualized block device is mounted, MultiLanes creates a translation thread for it. And we export the *xxx_get_block* function into the *inode_operations* structure for Ext3, Ext4, Btrfs, Reiserfs and JFS so that the translation thread can invoke it for block mapping via the inode of the back-end file.

The *multilanes_bio_end* function is implemented for I/O completion notification, which will be called each time the host block device driver completes a slice. We store global information such as the total slice number, finished slice count and error flags in the list head, and update the statistics every time it is called. The original request will be terminated by the host driver by calling the *bi_end_io* method of the original *bio* when the last slice is completed.

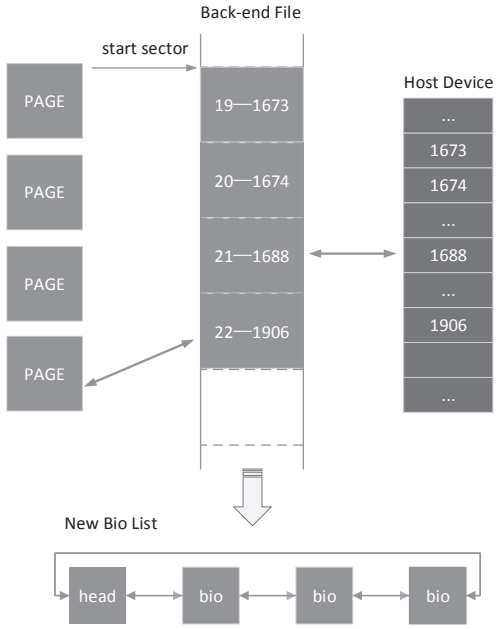


Figure 4: **Request Mapping.** *This figure shows the mapping from a single block I/O request of the virtualized block device to a request list on the host block device.*

Figure 4 shows an example of block request mapping. We assume the page size is 4096 bytes and the block size of the host block device and the virtualized storage device are both 4096 bytes. As shown in the figure, a block I/O request delivered to the virtualized driver consists of four segments. The start sector of the request is 152 and the block I/O size is 16KB. The *bio* contains four individual memory segments, which lie in four physical pages. After all the logical blocks of the request are mapped by the block translation unit, we can see that only the logical block 19 and 20 are contiguous on the host. MultiLanes allocates a new *bio* structure for the two contiguous blocks and two new ones for the remaining two blocks, and then delivers the new *bio*s to the host driver in sequence.

4.2 pVFS Implementation

The partitioned VFS data structures and locks are organized in the super block of the file system. We allocate SLAB caches *ihtable_cachep* and *dhtable_cachep* for inode and dentry hash table allocation when initializing the VFS at boot time. MultiLanes adds the *dentry_hashtable* pointer, the *inode_hashtable* pointer, and the corresponding locks (i.e., *inode_hash_lock* and *rename_lock*) to the super block. Meanwhile, each super block has its own LRU dentry list, and inode list along with the separated *dcache_lru_lock* and *inode_sb_list_lock*. We also add a flag field to the *superblock* structure to distinguish guest file systems on virtualized storage devices from other host file systems. For each guest file system, MultiLanes will allocate a dentry hash table and an inode hash table from the corresponding SLAB cache when the virtualized block device is mounted, both of which are predefined to have 65536 buckets.

Then we modify the kernel control flows that access the hash tables, lists and corresponding locks to allow each container to access its private VFS abstraction. We first find out all the code spots where the hash tables, lists and locks are accessed. Then, a multiplexer is embedded in each code spot to do the branching. Accesses to each guest file system are redirected to its private VFS data structures and locks while other accesses keep going through the original VFS. This work takes much efforts to finish all the code spots. But this is non-complicated work since the idea behind all modifications is the same.

5 Evaluation

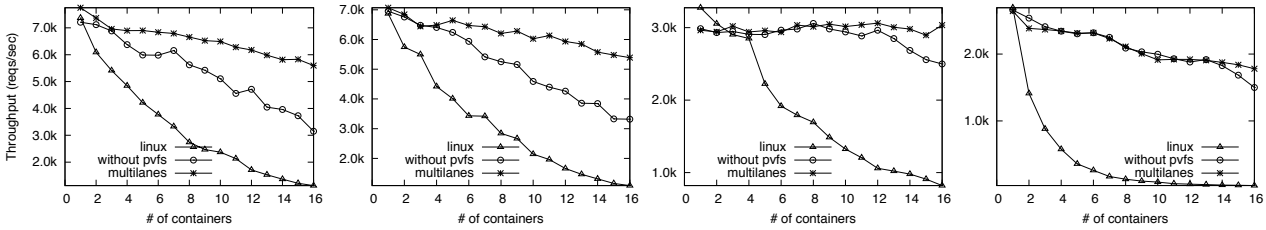
Fast storage devices mainly include prevailing NAND flash-based SSDs, and SSDs based on next-generation technologies (e.g., Phase Change Memory), which promise to further boost the performance. Unfortunately when the evaluation was conducted we did not have a high performance SSD at hand. So we used a RAM disk to emulate a PCM-based SSD since phase change memory is expected to have bandwidth and latency characteristics similar to DRAM [25]. The emulation is appropriate as Multilanes does not concern about the underlying specific storage media, as long as it is fast enough. Moreover, using a RAM disk could rule out any effect from SSDs (e.g., global locks adopted in their corresponding drivers) so as to measure the maximum scalability benefits of MultiLanes.

In this section, we experimentally answer the following questions: (1) Does MultiLanes achieve good scalability with the number of containers on many cores ? (2) Are all of MultiLanes's design components necessary to achieve such good scalability? (3) Does the overhead induced by MultiLanes contribute marginally to the performance under most workloads?

5.1 Experimental Setup

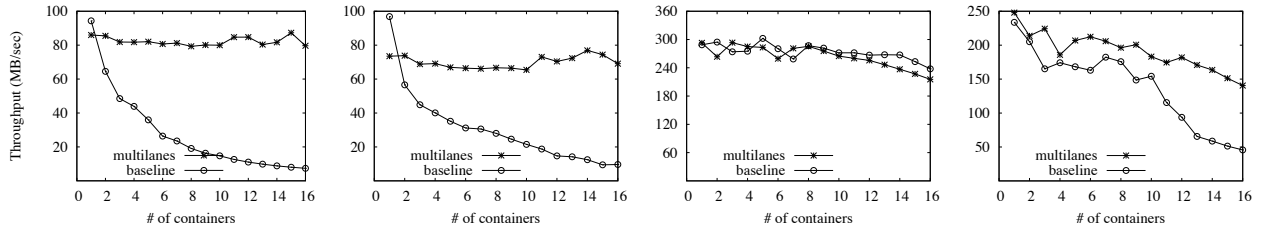
All experiments were carried out on an Intel 16-core machine with four Intel Xeon(R) E7520 processors and 64GB memory. Each processor has four physical cores clocked at 1.87GHZ. Each core has 32KB of L1 data cache, 32KB of L1 instruction cache and 256KB of L2 cache. Each processor has a shared 18MB L3 cache. The hyperthreading capability is turned off.

We turn on RAM block device support as a kernel module and set the RAM disk size to 40GB. Lock usage

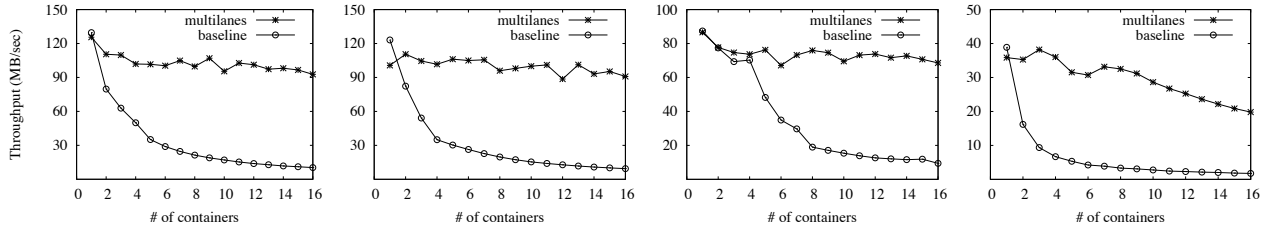


(a) Ocrd on Ext3 (b) Ocrd on Ext4 (c) Ocrd on XFS (d) Ocrd on Btrfs

Figure 5: **Scalability Evaluation with the Metadata-intensive Benchmark *Ocrd*.** *The figure shows the average throughput of the containers on different file systems when varying the number of LXC containers with ocrd. Inside each container we run a single instance of the benchmark program.*

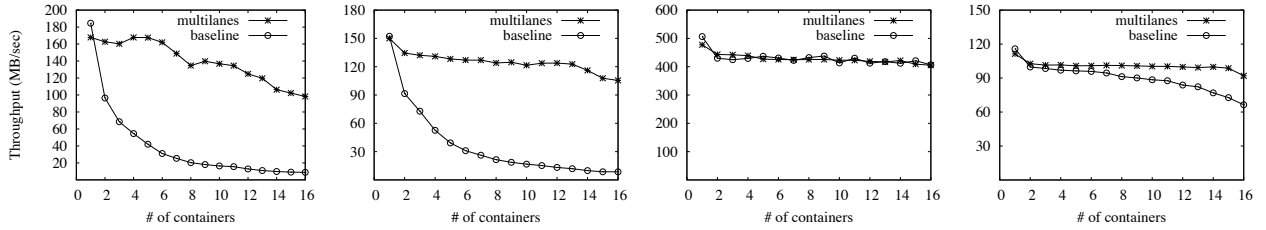


(a) Buffered write on Ext3 (b) Buffered write on Ext4 (c) Buffered write on XFS (d) Buffered write on Btrfs



(e) Direct write on Ext3 (f) Direct write on Ext4 (g) Direct write on XFS (h) Direct write on Btrfs

Figure 6: **Scalability Evaluation with IOzone (Sequential Workloads).** *The figure shows the container average throughput on different file systems when varying the number of LXC containers with IOzone. Inside each container we run an IOzone process performing sequential writes in buffered mode and direct I/O mode respectively.*



(a) Buffered write on Ext3 (b) Buffered write on Ext4 (c) Buffered write on XFS (d) Buffered write on Btrfs

Figure 7: **Scalability Evaluation with IOzone (Random Workloads).** *The figure shows the container average throughput on different file systems when varying the number of LXC containers with IOzone. Inside each container we run an IOzone process performing random writes in buffered mode.*

statistics is enabled to identify the heavily contended kernel locks during the evaluation. In this section, we evaluate MuliLanes against canonical Linux as the baseline. For the baseline groups, we have a RAM disk formatted with each target file system in turn and build 16 LXC containers atop it. For MultiLanes, we have the host RAM disk formatted with Ext3 and mounted in *ordered* mode, then build 16 LXC containers over 16 virtualized devices which are mapped as sixteen 2500MB regular files formatted with each target file system in turn. In all the experiments, the guest file system Ext3 and Ext4 are all mounted in *journal* mode unless otherwise specified.

5.2 Performance Results

The performance evaluation consists of both a collection of micro-benchmarks and a set of application-level macrobenchmarks.

5.2.1 Microbenckmarks

The purpose of the microbenchmarks is two-fold. First, these microbenchmarks give us the opportunity to measure an upper-bound on performance, as they effectively rule out any complex effects from application-specific behaviors. Second, microbenchmarks allow us to verify the effectiveness of each design component of MultiLanes as they stress differently. The benchmarks consist of the metadata-intensive benchmark *ocrd* developed from scratch, and IOzone [3] which is a representative storage system benchmark.

Ocrd. The ocrd benchmark runs 65536 transactions, and each transaction creates a new file, renames the file and at last deletes the file. It is set up for the purpose of illuminating the performance contributions of each individual design component of MultiLanes because the metadata-intensive workload could cause heavy contention on both the hot locks in the VFS, as mentioned in Table 2, and those in the underlying file systems.

Figure 5 presents the average throughput of each container running the *ocrd* benchmark for three situations: Linux, MultiLanes disabling pVFS and complete MultiLanes. As shown in the figure, the average throughput suffers severe degradation with the increasing number of containers on all four file systems in Linux. Lock usage statistics show it is caused by severe lock contention within both the underlying file system and the VFS. Contention bounces between cores can reach as many as several million times for the hot locks. MultiLanes without pVFS achieves great performance gains and much better scalability as the isolation via virtualized devices has eliminated contention in the file system layer. The average throughput on complete MultiLanes is further improved owing to the pVFS, exhibits marginal degradation with the increasing number of containers, and achieves nearly linear scalability. The results have demonstrated that each design component of MultiLanes is essential for scaling containers on many cores. Table 3 presents the contention details on the hot locks of the VFS that rise during the benchmark on MultiLanes without the pVFS. These locks are all eliminated by the pVFS.

It is interesting to note that the throughput of complete MultiLanes marginally outperforms that of Linux at one container on Ext3 and Ext4. This phenomenon is also observed in the below Varmail benchmark on Ext3, Ext4 and XFS. This might be because that the use of private VFS data structures provided by the pVFS speeds up the lookup in the dentry hash table as there are much less directory entries in each pVFS than in the global VFS.

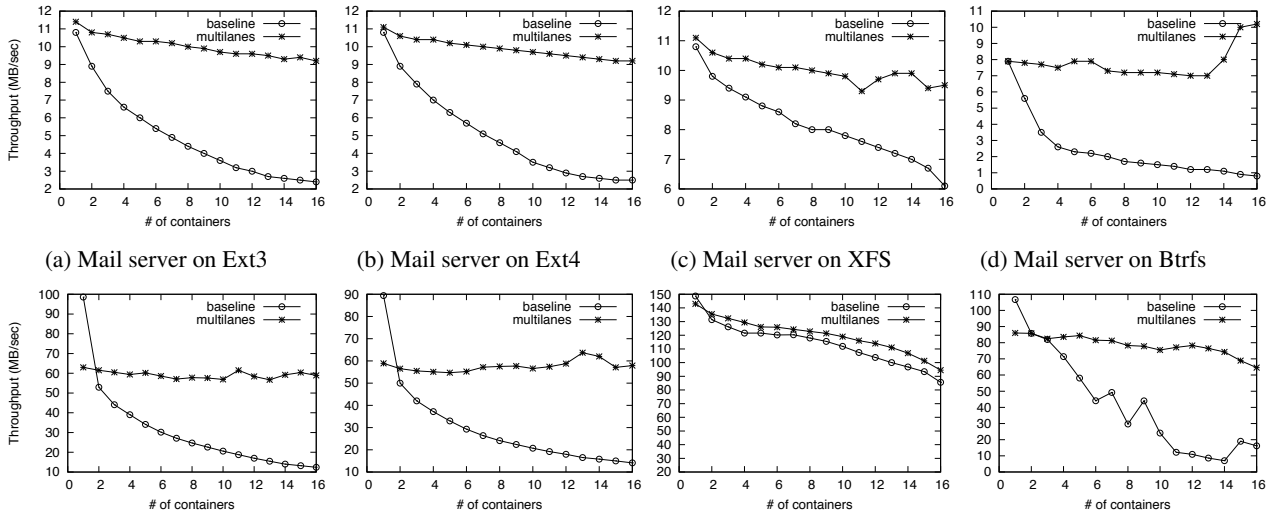
IOzone. We use the IOzone benchmark to evaluate the performance and scalability of MultiLanes for data-intensive workloads, including sequential and random workloads. Figure 6 shows the average throughput of each container performing sequential writes in buffered mode and direct I/O mode respectively. We run a single IOzone process inside each container in parallel and vary the number of containers. Sequential writes with 4KB I/O size are to a file that ends up with 256MB size. Note that Ext3 and Ext4 are mounted in ordered journaling mode for direct I/O writes as the data journaling mode does not support direct I/O.

Lock	Ext3	Ext4	XFS	Btrfs
inode_hash_lock	1092k	960k	114k	228k
dcache_lru_lock	1023k	797k	583k	5k
inode_sb_list_lock	239k	237k	144k	106k
rename_lock	541k	618k	446k	252k

Table 3: **Contention Bounces.** *The table shows the contention bounces using MultiLanes without pVFS.*

As shown in the figure, the average throughput of MultiLanes outperforms that of Linux in all cases except for buffered writes on XFS. MultiLanes outperforms Linux by 9.78X, 6.17X and 2.07X on Ext3, Ext4 and Btrfs for buffered writes respectively. For direct writes, the throughput improvement of MultiLanes over Linux is 7.98X, 8.67X, 6.29X and 10.32X on the four file systems respectively. XFS scales well for buffered writes owing to its own performance optimizations. Specially, XFS delays block allocation and associated metadata journaling until the dirty pages are to be flushed to disk. Delayed allocation avoids the contention induced by metadata journaling so as to scale well for buffered writes.

Figure 7 presents the results of random writes in buffered mode. Random writes with 4KB I/O size are to a 256MB file except for Btrfs. For Btrfs, we set each file size to 24MB due to the observation that when the writing data files occupy a certain proportion of the storage space Btrfs generates many work threads during the benchmark even for single-threaded random writes, which causes heavy contention and leads to sharply dropped throughput. Nevertheless, MultiLanes exhibits much better scalability and significantly outperforms the baseline at 16 containers for random writes to a 256MB file. However, in order to fairly evaluate the normal performance of both MultiLanes and Linux, we experimentally set a proper data file size for Btrfs. As shown in the figure, the throughput of MultiLanes outperforms that of Linux by 10.04X, 11.32X and 39% on Ext3, Ext4 and Btrfs respectively. As XFS scales well for buffered writes, MultiLanes exhibits competitive performance with it.



(a) Mail server on Ext3 (b) Mail server on Ext4 (c) Mail server on XFS (d) Mail server on Btrfs

(e) File server on Ext3 (f) File server on Ext4 (g) File server on XFS (h) File server on Btrfs

Figure 8: **Scalability Evaluation with Filebench Fileserver and Varmail.** *The figure shows the average throughput of the containers on different file systems when varying the number of LXC containers, with Filebench mail server and file server workload respectively.*

5.2.2 Macrobenchmarks

We choose Filebench [2] and MySQL [5] to evaluation performance and scalability of MultiLanes for application-level workloads.

Filebench. Filebench is a file system and storage benchmark that allows to generate a variety of workloads. Of all the workloads it supports, we choose the Varmail and Fileserver benchmarks as they are write-intensive workloads that would cause severe contention within the I/O stack.

The Varmail workload emulates a mail server, performing a sequence of create-append-sync, read-append-sync, reads and deletes. The Fileserver workload performs a sequence of creates, deletes, appends, reads, and writes. The specific parameters of the two workloads are listed in Table 4. We run a single instance of Filebench inside each container. The thread number of each instance is configured as 1 to avoid CPU overload when increasing the number of containers from 1 to 16. Each workload was run for 60 seconds.

Workload	# of Files	File Size	I/O Size	Append Size
Varmail	1000	16KB	1MB	16KB
Fileserver	2000	128KB	1MB	16KB

Table 4: **Workload Specification.** *This table specifies the parameters configured for Filebench Varmail and Fileserver workloads.*

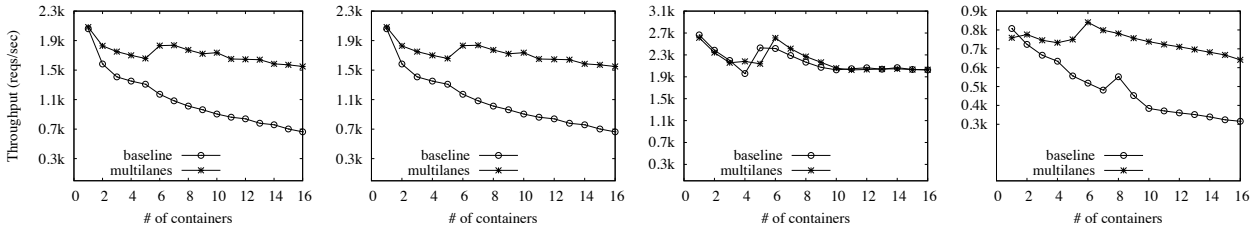
Figure 8 shows the average throughput of multiple concurrent Filebench instances on MultiLanes compared to Linux. For the Varmail workload, the average throughput degrades significantly with the increasing number of containers on the four file systems in Linux. MultiLanes exhibits little overhead when there is only one container, and marginal performance loss when the number of containers increases. The throughput of MultiLanes outperforms that of Linux by 2.83X, 2.68X, 56% and 11.75X on Ext3, Ext4, XFS and Btrfs respectively.

For the Fileserver workload, although the throughput of MultiLanes is worse than that of Linux at one single container, especially for Ext3 and Ext4, it scales well to 16 containers and outperforms that of Linux when the number of containers exceeds 2. In particular, MultiLanes achieves a speedup of 4.75X, 4.11X, 1.10X and 3.99X over the baseline Linux on the four file systems at 16 containers respectively. It is impressive that the throughput of MultiLanes at 16 containers even exceeds that at one single container on Btrfs. The phenomenon might relate to the design of Btrfs which is under actively development and does not become mature.

MySQL. MySQL is an open source relational database management system that runs as a server providing multi-user accesses to databases. It is widely used for data storage and management in web applications.

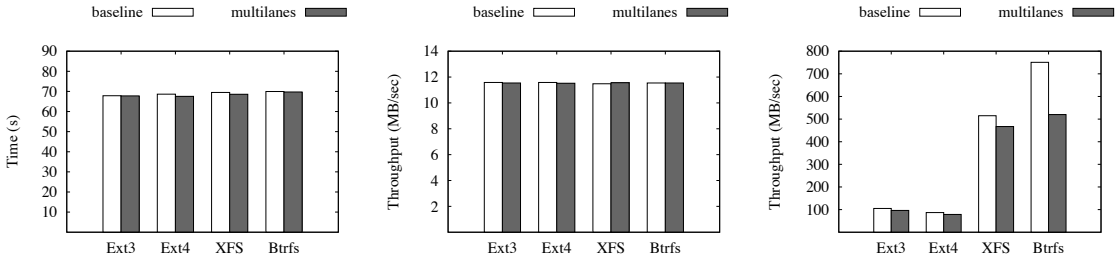
We install mysql-server-5.1 for each container and start the service for each of them. The virtualized MySQL servers are configured to allow remote accesses and we generate requests with Sysbench [7] on another identical machine that resides in the same LAN with the experimental server. The evaluation is conducted in non-transaction mode that specializes update_key operations as the transaction mode provided by Sysbench is dominated by read operations. Each table is initialized with 10k records at the prepare stage. We use 1 thread to generate 20k requests for each MySQL server.

As Figure 9 shows, MultiLanes improves the throughput by 87%, 1.34X and 1.03X on Ext3, Ext4, and Btrfs



(a) MySQL on Ext3 (b) MySQL on Ext4 (c) MySQL on XFS (d) MySQL on Btrfs

Figure 9: **Scalability Evaluation with MySQL.** *This figure shows the average throughput of the containers when varying the number of LXC containers on different file systems with MySQL. The requests are generated with Sysbench on another identical machine in the same LAN.*



(a) Apache Build (b) Webserver (c) Streamwrite

Figure 10: **Overhead Evaluation.** *This figure shows the overhead of MultiLanes relative to Linux, running Apache build, Filebench Webserver and Filebench single-stream write inside a single container respectively.*

respectively. And once again we have come to see that XFS scales well on many cores, and MultiLanes shows competitive performance with it. The throughput of MultiLanes exhibits nearly linear scalability with the increasing number of containers on the four file systems.

5.3 Overhead Analysis

We also measure the potential overhead of MultiLanes's approach to eliminating contention in OS-level virtualization by using an extensive set of benchmarks: Apache Build, Webserver and Streamwrite, which is file I/O less intensive, read intensive and write intensive respectively.

Apache Build. The Apache Build benchmark, which overlaps computation with file I/O, unzips the Apache source tree, does a complete build in parallel with 16 threads, and then removes all files. Figure 10a shows the execution time of the benchmark on MultiLanes over Linux. We can see that MultiLanes exhibits almost equivalent performance against Linux. The result demonstrates that the overhead of MultiLanes would not affect the performance of workloads which are not dominated by file I/O.

Webserver. We choose the Filebench Webserver workload to evaluate the overhead of MultiLanes under read-intensive workloads. The parameters of the benchmark is configured as default. Figure 10b presents the throughput of MultiLanes against Linux. The result shows that the virtualization layer of MultiLanes contributes marginally to the performance under the Webserver workload.

Streamwrite. The single-stream write benchmark performs 1MB sequential writes to a file that ends up with about 1GB size. Figure 10c shows the throughput of benchmark on MultiLanes over Linux. As the sequential stream writes cause frequent block allocation of the back-end file, MultiLanes incurs some overheads of block mapping cache misses. The overhead of MultiLanes compared to Linux is 9.0%, 10.5%, 10.2% and 44.7% for Ext3, Ext4, XFS and Btrfs respectively.

6 Related Work

This section relates MultiLanes to other work done in performance isolation, kernel scalability and device virtualization.

Performance Isolation. Most work on performance isolation mainly focuses on minimizing performance interference by space partitioning or time multiplexing hardware resources (e.g., CPU, memory, disk and network bandwidth) between the co-located containers. VServer [32] enforces resource isolation by carefully allocating and scheduling physical resources. Resource containers [10] provides explicit and fine-grained control over resource consumption in all levels in the system. Eclipse [14] introduces a new operating system ab-

straction to enable explicit control over the provisioning of the system resources among applications. Software Performance Units [35] enforces performance isolation by restricting the resource consumption of each group of processes. Cgroup [1], which is used in LXC to provide resource isolation between co-located containers, is a Linux kernel feature to limit, account and isolate the resource usage of process groups. Argon [36] mainly focuses on the I/O schedule algorithms and the file system cache partition mechanisms to provide storage performance isolation.

In contrast, MultiLanes aims to eliminate contention on shared kernel data structures and locks in the software to reduce storage performance interference between the VEs. Hence our work is complementary and orthogonal to previous studies on performance isolation.

Kernel Scalability. Improving the scalability of operating systems has been a longstanding goal of system researchers. Some work investigates new OS structures to scale operating systems by partitioning the hardware and distributing replicated kernels among the partitioned hardware. Hive [17] structures the operating system as an internal distributed system of independent kernels to provide reliability and scalability. Barrelfish [11] tries to scale applications on multicore systems using a multi-kernel model, which maintains the operating system consistency by message-passing instead of shared-memory. Corey [12] is an exokernel based operating system that allows applications to control the sharing of kernel resources. K42 [9] (and its relative Tornado [20]) are designed to reduce contention and improve locality on NUMA systems. Other work partitions hardware resources by running a virtualization layer to allow the concurrent execution of multiple commodity operating systems. For instance, Diso [15] (and its relative Cellular Diso [22]) runs multiple virtual machines to create a virtual cluster on large-scale shared-memory multiprocessors to provide reliability and scalability. Cerberus [33] scales shared-memory applications with POSIX-APIs on many cores by running multiple clustered operating systems atop VMM on a single machine. MultiLanes is influenced by the philosophy and wisdoms of these work but strongly foucses on the scalability of I/O stack on fast storage devices.

Other studies aim to address the scalability problem by iteratively eliminating the bottlenecks. MCS lock [28], RCU [27] and local runqueues [8] are strategies proposed to reduce contention on shared data structures.

Device Virtualization. Traditionally, hardware abstraction virtualization adopts three approaches to virtualize devices. First, device emulation [34] is used to emulate familiar devices such as common network cards and SCSI devices. Second, para-virtualization [30] customizes the virtualized device driver to enable the guest OS to explicitly cooperate with the hypervisor for performance improvements. Such examples include KVM's VirtIO driver, Xen's para-virtualized driver, and VMware's guest tools. Third, direct device assignment [21, 19, 26] gives the guest direct accesses to physical devices to achieve near-native hardware performance.

MultiLanes maps a regular file as the virtualized device of a VE rather than giving it direct accesses to a physical device or a logical volume. The use of backend files eases the management of the storage images [24]. Our virtualized block device approach is more efficient when compared to device emulation and para-virtualization as it comes with little overhead by adopting a bypass strategy.

7 Conclusions

The advent of fast storage technologies has shifted the I/O bottlenecks from the storage devices to system software. The co-located containers in OS-level virtualization will suffer from severe storage performance interference on many cores due to the fact that they share the same I/O stack. In this work, we propose MultiLanes, which consists of the virtualized storage device, and the partitioned VFS, to provide an isolate I/O stack to each container on many cores. The evaluation demonstrates that MultiLanes effectively addresses the I/O performance interference between the VEs on many cores and exhibits significant performance improvement compared to Linux for most workloads.

As we try to eliminate contention on shared data structures and locks within the file system layer with the virtualized storage device, the effectiveness of our approach is based on the premise that multiple file system instances work independently and share almost nothing. For those file systems in which the instances share the same worker thread pool (e.g., JFS), there might still exist performance interference between containers.

8 Acknowledgements

We would like to thank our shepherd Anand Sivasubramaniam and the anonymous reviewers for their excellent feedback and suggestions. This work was funded by China 973 Program (No.2011CB302602), China 863 Program (No.2011AA01A202, 2013AA01A213), HGJ Program (2010ZX01045-001-002-4) and Projects from NSFC (No.61170294, 91118008). Tianyu Wo and Chunming Hu are the corresponding authors of this paper.

References

[1] Cgroup. `https://www.kernel.org/doc/Documentation/cgroups`.

[2] Filebench. http://sourceforge.net/projects/filebench/.

[3] IOzone. http://www.iozone.org/.

[4] LXC. http://en.wikipedia.org/wiki/LXC.

[5] MySQL. http://www.mysql.com/.

[6] OpenVZ. http://en.wikipedia.org/wiki/OpenVZ.

[7] Sysbench. http://sysbench.sourceforge.net/.

[8] AAS, J. Understanding the Linux 2.6.8.1 CPU scheduler. http://josh.trancesoftware.com/linux/.

[9] APPAVOO, J., SILVA, D. D., KRIEGER, O., AUSLANDER, M. A., OSTROWSKI, M., ROSENBURG, B. S., WATERLAND, A., WISNIEWSKI, R. W., XENIDIS, J., STUMM, M., AND SOARES, L. Experience distributing objects in an SMMP OS. *ACM Trans. Comput. Syst. 25*, 3 (2007).

[10] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource Containers: A new facility for resource management in server systems. In *OSDI* (1999).

[11] BAUMANN, A., BARHAM, P., DAGAND, P.-É., HARRIS, T. L., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP* (2009).

[12] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, M. F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., HUA DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *OSDI* (2008).

[13] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of Linux scalability to many cores. In *OSDI* (2010).

[14] BRUNO, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. The Eclipse operating system: Providing quality of service via reservation domains. In *USENIX Annual Technical Conference* (1998).

[15] BUGNION, E., DEVINE, S., AND ROSENBLUM, M. Disco: Running commodity operating systems on scalable multiprocessors. In *SOSP* (1997).

[16] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO* (2010).

[17] CHAPIN, J., ROSENBLUM, M., DEVINE, S., LAHIRI, T., TEODOSIU, D., AND GUPTA, A. Hive: Fault containment for shared-memory multiprocessors. In *SOSP* (1995).

[18] CUI, Y., WANG, Y., CHEN, Y., AND SHI, Y. Lock-contention-aware scheduler: A scalable and energy-efficient method for addressing scalability collapse on multicore systems. *TACO 9*, 4 (2013), 44.

[19] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMSON, M. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)* (2004).

[20] GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI* (1999).

[21] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: bare-metal performance for I/O virtualization. In *ASPLOS* (2012).

[22] GOVIL, K., TEODOSIU, D., HUANG, Y., AND ROSENBLUM, M. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *SOSP* (1999).

[23] KLEIMAN, S. R. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Summer* (1986).

[24] LE, D., HUANG, H., AND WANG, H. Understanding performance implications of nested file systems in a virtualized environment. In *FAST* (2012).

[25] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable DRAM alternative. In *ISCA* (2009).

[26] MANSLEY, K., LAW, G., RIDDOCH, D., BARZINI, G., TURTON, N., AND POPE, S. Getting 10 Gb/s from Xen: Safe and fast device access from unprivileged domains. In *Euro-Par Workshops* (2007).

[27] MCKENNEY, P. E., SARMA, D., ARCANGELI, A., KLEEN, A., KRIEGER, O., AND RUSSELL, R. Read-copy update. In *Linux Symposium* (2002).

[28] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 9*, 1 (1991), 21–65.

[29] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The design and implementation of Zap: A system for migrating computing environments. In *OSDI* (2002).

[30] RUSSELL, R. virtio: towards a de-facto standard for virtual I/O devices. *Operating Systems Review 42*, 5 (2008), 95–103.

[31] SEPPANEN, E., O'KEEFE, M. T., AND LILJA, D. J. High performance solid state storage under Linux. In *MSST* (2010).

[32] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A. C., AND PETERSON, L. L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *EuroSys* (2007).

[33] SONG, X., CHEN, H., CHEN, R., WANG, Y., AND ZANG, B. A case for scaling applications to many-core with OS clustering. In *EuroSys* (2011).

[34] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track* (2001).

[35] VERGHESE, B., GUPTA, A., AND ROSENBLUM, M. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *ASPLOS* (1998).

[36] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance insulation for shared storage servers. In *FAST* (2007).